VxWorks® OS Libraries

5.5

API REFERENCE

WindRiver®

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 OS Libraries API Reference, 5.5

6 Jun 02
Part #: DOC-14604-ND-00

Contents

1: Libraries

This volume provides reference entries for VxWorks OS 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 OS libraries documented in section 1.

Keyword Index

This section is a "permuted index" of keywords found in the NAME line of each reference entry. The keyword for each index item is left-aligned in column 2. The remaining words in column 1 and 2 show the context for the keyword.

1
Libraries

aioPxLib

NAME

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

ROUTINES

aioPxLibInit( ) - initialize the asynchronous I/O (AIO) library
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_error( ) - retrieve error status of asynchronous I/O operation (POSIX)
aio_return( ) - retrieve return status of asynchronous I/O operation (POSIX)

DESCRIPTION

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

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

AIO LIBRARY

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

AIO COMMANDS

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

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

aio_read( )
initiates an asynchronous read

aio_write( )
initiates an asynchronous write

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

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

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

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

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

AIO CONTROL BLOCK

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

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

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

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

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

aio_fildes
: file descriptor for I/O.

aio_offset
: offset from the beginning of the file where the AIO takes place. Note that performing AIO on the file does not cause the offset location to automatically increase as in read and write; the caller must therefore keep track of the location of reads and writes made to the file (see *POSIX COMPLIANCE* below).

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

aio_nbytes
number of bytes to read or write.

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

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

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

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

EXAMPLES

A writer could be implemented as follows:

```
if ((pAioWrite = calloc (1, sizeof (struct aiocb))) == NULL)
    {
    printf ("calloc failed\n");
    return (ERROR);
    }
pAioWrite->aio_fildes = fd;
pAioWrite->aio_buf = buffer;
pAioWrite->aio_offset = 0;
strcpy (pAioWrite->aio_buf, "test string");
pAioWrite->aio_nbytes  = strlen ("test string");
pAioWrite->aio_sigevent.sigev_notify = SIGEV_NONE;
aio_write (pAioWrite);
/*  .
    .
    do other work
    .
    .
*/
/* now wait until I/O finishes */
while (aio_error (pAioWrite) == EINPROGRESS)
    taskDelay (1);
```

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

A reader could be implemented as follows:

```
    /* initialize signal handler */

    action1.sa_sigaction = sigHandler;
    action1.sa_flags   = SA_SIGINFO;
    sigemptyset(&action1.sa_mask);
    sigaction (TEST_RT_SIG1, &action1, NULL);
    if ((pAioRead = calloc (1, sizeof (struct aiocb))) == NULL)
        {
        printf ("calloc failed\n");
        return (ERROR);
        }

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

    aio_read (pAioRead);
    /*
        .
        .
        do other work
        .
        .
    */
```

The signal handler might look like the following:

```
void sigHandler
    (
    int                 sig,
    struct siginfo      info,
    void *              pContext
    )
    {
    struct aiocb *      pAioDone;
    pAioDone = (struct aiocb *) info.si_value.sival_ptr;
    aio_return (pAioDone);
    free (pAioDone);
    }
```

A

POSIX COMPLIANCE

Currently VxWorks does not support the **O_APPEND** flag in the open call. Therefore, the user must keep track of the offset in the file that the asynchronous writes occur (as in the case of reads). The **aio_offset** field is used to specify that file position.

In addition, VxWorks does not currently support synchronized I/O.

INCLUDE FILES **aio.h**

SEE ALSO POSIX 1003.1b document

aioPxShow

NAME **aioPxShow** – asynchronous I/O (AIO) show library

ROUTINES **aioShow( )** - show AIO requests

DESCRIPTION This library implements the show routine for **aioPxLib**.

aioSysDrv

NAME **aioSysDrv** – AIO system driver

ROUTINES **aioSysInit( )** - initialize the AIO system driver

DESCRIPTION This library is the AIO system driver. The system driver implements asynchronous I/O with system AIO tasks performing the AIO requests in a synchronous manner. It is installed as the default driver for AIO.

SEE ALSO POSIX 1003.1b document

ansiAssert

NAME **ansiAssert** – ANSI **assert** documentation

ROUTINES **assert( )** - put diagnostics into programs (ANSI)

DESCRIPTION

The header **assert.h** defines the **assert( )** macro and refers to another macro, NDEBUG, which is not defined by **assert.h**. If NDEBUG is defined as a macro at the point in the source file where **assert.h** is included, the **assert( )** macro is defined simply as:

```
#define assert(ignore) ((void)0)
```

ANSI specifies that **assert( )** should be implemented as a macro, not as a routine. If the macro definition is suppressed in order to access an actual routine, the behavior is undefined.

INCLUDE FILES

stdio.h, **stdlib.h**, **assert.h**

SEE ALSO

American National Standard X3.159-1989

ansiCtype

NAME

ansiCtype – ANSI **ctype** documentation

ROUTINES

isalnum( ) - test whether a character is alphanumeric (ANSI)
isalpha( ) - test whether a character is a letter (ANSI)
iscntrl( ) - test whether a character is a control character (ANSI)
isdigit( ) - test whether a character is a decimal digit (ANSI)
isgraph( ) - test whether a character is a printing, non-white-space character (ANSI)
islower( ) - test whether a character is a lower-case letter (ANSI)
isprint( ) - test whether a character is printable, including the space character (ANSI)
ispunct( ) - test whether a character is punctuation (ANSI)
isspace( ) - test whether a character is a white-space character (ANSI)
isupper( ) - test whether a character is an upper-case letter (ANSI)
isxdigit( ) - test whether a character is a hexadecimal digit (ANSI)
tolower( ) - convert an upper-case letter to its lower-case equivalent (ANSI)
toupper( ) - convert a lower-case letter to its upper-case equivalent (ANSI)

DESCRIPTION

The header **ctype.h** declares several functions useful for testing and mapping characters. In all cases, the argument is an **int**, the value of which is representable as an **unsigned char** or is equal to the value of the macro **EOF**. If the argument has any other value, the behavior is undefined.

The behavior of the **ctype** functions is affected by the current locale. VxWorks supports only the “C” locale.

The term “printing character” refers to a member of an implementation-defined set of characters, each of which occupies one printing position on a display device; the term “control character” refers to a member of an implementation-defined set of characters that are not printing characters.

INCLUDE FILES **ctype.h**

SEE ALSO American National Standard X3.159-1989

ansiLocale

NAME **ansiLocale** – ANSI **locale** documentation

ROUTINES
localeconv( ) - set the components of an object with type **lconv** (ANSI)
setlocale( ) - set the appropriate locale (ANSI)

DESCRIPTION The header **locale.h** declares two functions and one type, and defines several macros. The type is:

struct lconv
contains members related to the formatting of numeric values. The structure should contain at least the members defined in **locale.h**, in any order.

SEE ALSO **localeconv( )**, **setlocale( )**, American National Standard X3.159-1989

ansiMath

NAME **ansiMath** – ANSI **math** documentation

ROUTINES
asin( ) - compute an arc sine (ANSI)
acos( ) - compute an arc cosine (ANSI)
atan( ) - compute an arc tangent (ANSI)
atan2( ) - compute the arc tangent of y/x (ANSI)
ceil( ) - compute the smallest integer greater than or equal to a specified value (ANSI)
cosh( ) - compute a hyperbolic cosine (ANSI)
exp( ) - compute an exponential value (ANSI)
fabs( ) - compute an absolute value (ANSI)
floor( ) - compute the largest integer less than or equal to a specified value (ANSI)
fmod( ) - compute the remainder of x/y (ANSI)
frexp( ) - break a floating-point number into a normalized fraction and power of 2 (ANSI)
ldexp( ) - multiply a number by an integral power of 2 (ANSI)
log( ) - compute a natural logarithm (ANSI)
log10( ) - compute a base-10 logarithm (ANSI)

modf( ) - separate a floating-point number into integer and fraction parts (ANSI)
pow( ) - compute the value of a number raised to a specified power (ANSI)
sin( ) - compute a sine (ANSI)
cos( ) - compute a cosine (ANSI)
sinh( ) - compute a hyperbolic sine (ANSI)
sqrt( ) - compute a non-negative square root (ANSI)
tan( ) - compute a tangent (ANSI)
tanh( ) - compute a hyperbolic tangent (ANSI)

DESCRIPTION

The header **math.h** declares several mathematical functions and defines one macro. The functions take double arguments and return double values.

The macro defined is:

HUGE_VAL
expands to a positive double expression, not necessarily representable as a float.

The behavior of each of these functions is defined for all representable values of their input arguments. Each function executes as if it were a single operation, without generating any externally visible exceptions.

For all functions, a domain error occurs if an input argument is outside the domain over which the mathematical function is defined. The description of each function lists any applicable domain errors. On a domain error, the function returns an implementation-defined value; the value **EDOM** is stored in **errno**.

Similarly, a range error occurs if the result of the function cannot be represented as a double value. If the result overflows (the magnitude of the result is so large that it cannot be represented in an object of the specified type), the function returns the value **HUGE_VAL**, with the same sign (except for the **tan( )** function) as the correct value of the function; the value **ERANGE** is stored in **errno**. If the result underflows (the type), the function returns zero; whether the integer expression **errno** acquires the value **ERANGE** is implementation defined.

INCLUDE FILES

math.h

SEE ALSO

mathALib, American National Standard X3.159-1989

ansiSetjmp

NAME

ansiSetjmp – ANSI **setjmp** documentation

ROUTINES

setjmp( ) - save the calling environment in a **jmp_buf** argument (ANSI)
longjmp( ) - perform non-local goto by restoring saved environment (ANSI)

A

DESCRIPTION

The header **setjmp.h** defines functions and one type for bypassing the normal function call and return discipline.

The type declared is:

jmp_buf
an array type suitable for holding the information needed to restore a calling environment.

The ANSI C standard does not specify whether **setjmp( )** is a subroutine or a macro.

SEE ALSO

American National Standard X3.159-1989

ansiStdarg

NAME

ansiStdarg – ANSI **stdarg** documentation

ROUTINES

va_start( ) - initialize a **va_list** object for use by **va_arg( )** and **va_end( )**
va_arg( ) - expand to an expression having the type and value of the call's next argument
va_end( ) - facilitate a normal return from a routine using a **va_list** object

DESCRIPTION

The header **stdarg.h** declares a type and defines three macros for advancing through a list of arguments whose number and types are not known to the called function when it is translated.

A function may be called with a variable number of arguments of varying types. The rightmost parameter plays a special role in the access mechanism, and is designated *parmN* in this description.

The type declared is:

va_list
a type suitable for holding information needed by the macros **va_start( )**, **va_arg( )**, and **va_end( )**.

To access the varying arguments, the called function shall declare an object having type **va_list**. The object (referred to here as *ap*) may be passed as an argument to another function; if that function invokes the **va_arg( )** macro with parameter *ap*, the value of *ap* in the calling function is indeterminate and is passed to the **va_end( )** macro prior to any further reference to *ap*.

va_start( ) and **va_arg( )** have been implemented as macros, not as functions. The **va_start( )** and **va_end( )** macros should be invoked in the function accepting a varying number of arguments, if access to the varying arguments is desired.

The use of these macros is documented here as if they were architecture-generic. However, depending on the compilation environment, different macro versions are included by **vxWorks.h**.

SEE ALSO American National Standard X3.159-1989

ansiStdio

NAME **ansiStdio** – ANSI **stdio** documentation

ROUTINES

clearerr( ) - clear end-of-file and error flags for a stream (ANSI)
fclose( ) - close a stream (ANSI)
fdopen( ) - open a file specified by a file descriptor (POSIX)
feof( ) - test the end-of-file indicator for a stream (ANSI)
ferror( ) - test the error indicator for a file pointer (ANSI)
fflush( ) - flush a stream (ANSI)
fgetc( ) - return the next character from a stream (ANSI)
fgetpos( ) - store the current value of the file position indicator for a stream (ANSI)
fgets( ) - read a specified number of characters from a stream (ANSI)
fileno( ) - return the file descriptor for a stream (POSIX)
fopen( ) - open a file specified by name (ANSI)
fprintf( ) - write a formatted string to a stream (ANSI)
fputc( ) - write a character to a stream (ANSI)
fputs( ) - write a string to a stream (ANSI)
fread( ) - read data into an array (ANSI)
freopen( ) - open a file specified by name (ANSI)
fscanf( ) - read and convert characters from a stream (ANSI)
fseek( ) - set the file position indicator for a stream (ANSI)
fsetpos( ) - set the file position indicator for a stream (ANSI)
ftell( ) - return the current value of the file position indicator for a stream (ANSI)
fwrite( ) - write from a specified array (ANSI)
getc( ) - return the next character from a stream (ANSI)
getchar( ) - return the next character from the standard input stream (ANSI)
gets( ) - read characters from the standard input stream (ANSI)
getw( ) - read the next word (32-bit integer) from a stream
perror( ) - map an error number in **errno** to an error message (ANSI)
putc( ) - write a character to a stream (ANSI)
putchar( ) - write a character to the standard output stream (ANSI)
puts( ) - write a string to the standard output stream (ANSI)
putw( ) - write a word (32-bit integer) to a stream
rewind( ) - set the file position indicator to the beginning of a file (ANSI)
scanf( ) - read and convert characters from the standard input stream (ANSI)

setbuf( ) - specify the buffering for a stream (ANSI)
setbuffer( ) - specify buffering for a stream
setlinebuf( ) - set line buffering for standard output or standard error
setvbuf( ) - specify buffering for a stream (ANSI)
stdioInit( ) - initialize standard I/O support
stdioFp( ) - return the standard input/output/error FILE of the current task
stdioShowInit( ) - initialize the standard I/O show facility
stdioShow( ) - display file pointer internals
tmpfile( ) - create a temporary binary file (Unimplemented) (ANSI)
tmpnam( ) - generate a temporary file name (ANSI)
ungetc( ) - push a character back into an input stream (ANSI)
vfprintf( ) - write a formatted string to a stream (ANSI)

DESCRIPTION

The header **stdio.h** declares three types, several macros, and many functions for performing input and output.

Types

The types declared are **size_t** and:

FILE
object type capable of recording all the information needed to control a stream, including its file position indicator, a pointer to its associated buffer (if any), an error indicator that records whether a read/write error has occurred, and an end-of-file indicator that records whether the end of the file has been reached.

fpos_t
object type capable of recording all the information needed to specify uniquely every position within a file.

Macros

The macros are **NULL** and:

_IOFBF, _IOLBF, _IONBF
expand to integral constant expressions with distinct values, suitable for use as the third argument to **setvbuf( )**.

BUFSIZ
expands to an integral constant expression that is the size of the buffer used by **setbuf( )**.

EOF
expands to a negative integral constant expression that is returned by several functions to indicate **end-of-file**, that is, no more input from a stream.

FOPEN_MAX
expands to an integral constant expression that is the minimum number of the files that the system guarantees can be open simultaneously.

FILENAME_MAX
expands to an integral constant expression that is the size needed for an array of **char** large enough to hold the longest file name string that can be used.

L_tmpnam
expands to an integral constant expression that is the size needed for an array of **char** large enough to hold a temporary file name string generated by **tmpnam( )**.

SEEK_CUR, SEEK_END, SEEK_SET
expand to integral constant expressions with distinct values suitable for use as the third argument to **fseek( )**.

TMP_MAX
expands to an integral constant expression that is the minimum number of file names generated by **tmpnam( )** that will be unique.

'stderr, stdin, stdout'
expressions of type "pointer to FILE" that point to the FILE objects associated, respectively, with the standard error, input, and output streams.

STREAMS

Input and output, whether to or from physical devices such as terminals and tape drives, or whether to or from files supported on structured storage devices, are mapped into logical data streams, whose properties are more uniform than their various inputs and outputs. Two forms of mapping are supported: for text streams and for binary streams.

A text stream is an ordered sequence of characters composed into lines, each line consisting of zero or more characters plus a terminating new-line character. Characters may have to be added, altered, or deleted on input and output to conform to differing conventions for representing text in the host environment. Thus, there is no need for a one-to-one correspondence between the characters in a stream and those in the external representation. Data read in from a text stream will necessarily compare equal to the data that were earlier written out to that stream only if: the data consists only of printable characters and the control characters horizontal tab and new-line; no new-line character is immediately preceded by space characters; and the last character is a new-line character. Space characters are written out immediately before a new-line character appears.

A binary stream is an ordered sequence of characters that can transparently record internal data. Data read in from a binary stream should compare equal to the data that was earlier written out to that stream, under the same implementation. However, such a stream may have a number of null characters appended to the end of the stream.

Environmental Limits

VxWorks supports text files with lines containing at least 254 characters, including the terminating new-line character. The value of the macro **BUFSIZ** is 1024.

FILES

A stream is associated with an external file (which may be a physical device) by opening a file, which may involve creating a new file. Creating an existing file causes its former contents to be discarded, if necessary. If a file can support positioning requests (such as a

A

disk file, as opposed to a terminal), then a file position indicator associated with the stream is positioned at the start (character number zero) of the file. The file position indicator is maintained by subsequent reads, writes, and positioning requests, to facilitate an orderly progression through the file. All input takes place as if characters were read by successive calls to **fgetc( )**; all output takes place as if characters were written by successive calls to **fputc( )**.

Binary files are not truncated, except as defined in **fopen( )** documentation.

When a stream is unbuffered, characters are intended to appear from the source or at the destination as soon as possible. Otherwise characters may be accumulated and transmitted to or from the host environment as a block. When a stream is fully buffered, characters are intended to be transmitted to or from the host environment as a block when the buffer is filled. When a stream is line buffered, characters are intended to be transmitted to or from the host environment as a block when a new-line character is encountered. Furthermore, characters are intended to be transmitted as a block to the host environment when a buffer is filled, when input is requested on an unbuffered stream, or when input is requested on a line-buffered stream that requires the transmission of characters from the host environment. VxWorks supports these characteristics via the **setbuf( )** and **setvbuf( )** functions.

A file may be disassociated from a controlling stream by closing the file. Output streams are flushed (any unwritten buffer contents are transmitted to the host environment) before the stream is disassociated from the file. The value of a pointer to a **FILE** object is indeterminate after the associated file is closed (including the standard text streams).

The file may be subsequently reopened, by the same or another program execution, and its contents reclaimed or modified (if it can be repositioned at its start).

TASK TERMINATION

ANSI specifies that if the main function returns to its original caller or if **exit( )** is called, all open files are closed (and hence all output streams are flushed) before program termination. This does **not** happen in VxWorks. The **exit( )** function does not close all files opened for that task. A file opened by one task may be used and closed by another. Unlike in UNIX, when a VxWorks task exits, it is the responsibility of the task to **fclose( )** its file pointers, except **stdin**, **stdout**, and **stderr**. If a task is to be terminated asynchronously, use **kill( )** and arrange for a signal handler to clean up.

The address of the **FILE** object used to control a stream may be significant; a copy of a **FILE** object may not necessarily serve in place of the original.

At program startup, three text streams are predefined and need not be opened explicitly: standard input (for reading conventional input), standard output (for writing conventional output), and standard error (for writing diagnostic output). When opened, the standard error stream is not fully buffered; the standard input and standard output streams are fully buffered if and only if the stream can be determined not to refer to an interactive device.

Functions that open additional (non-temporary) files require a file name, which is a string. VxWorks allows the same file to be open multiple times simultaneously. It is up to the user to maintain synchronization between different tasks accessing the same file.

FIOLIB

Several routines normally considered part of standard I/O -- **printf( )**, **sprintf( )**, **vprintf( )**, **vsprintf( )**, and **sscanf( )** -- are not implemented as part of the buffered standard I/O library; they are instead implemented in **fioLib**. They do not use the standard I/O buffering scheme. They are self-contained, formatted, but unbuffered I/O functions. This allows a limited amount of formatted I/O to be achieved without the overhead of the standard I/O library.

SEE ALSO

fioLib, *American National Standard for Information Systems - Programming Language - C, ANSI X3.159-1989: Input/Output (**stdio.h**)*

ansiStdlib

NAME

ansiStdlib – ANSI **stdlib** documentation

ROUTINES

abort( ) - cause abnormal program termination (ANSI)
abs( ) - compute the absolute value of an integer (ANSI)
atexit( ) - call a function at program termination (Unimplemented) (ANSI)
atof( ) - convert a string to a **double** (ANSI)
atoi( ) - convert a string to an **int** (ANSI)
atol( ) - convert a string to a **long** (ANSI)
bsearch( ) - perform a binary search (ANSI)
div( ) - compute a quotient and remainder (ANSI)
div_r( ) - compute a quotient and remainder (reentrant)
labs( ) - compute the absolute value of a **long** (ANSI)
ldiv( ) - compute the quotient and remainder of the division (ANSI)
ldiv_r( ) - compute a quotient and remainder (reentrant)
mblen( ) - calculate the length of a multibyte character (Unimplemented) (ANSI)
mbtowc( ) - convert a multibyte character to a wide character (Unimplemented) (ANSI)
wctomb( ) - convert a wide character to a multibyte character (Unimplemented) (ANSI)
mbstowcs( ) - convert a series of multibyte char's to wide char's (Unimplemented) (ANSI)
wcstombs( ) - convert a series of wide char's to multibyte char's (Unimplemented) (ANSI)
qsort( ) - sort an array of objects (ANSI)
rand( ) - generate a pseudo-random integer between 0 and **RAND_MAX** (ANSI)
srand( ) - reset the value of the seed used to generate random numbers (ANSI)
strtod( ) - convert the initial portion of a string to a double (ANSI)
strtol( ) - convert a string to a long integer (ANSI)

strtoul( ) - convert a string to an unsigned long integer (ANSI)
system( ) - pass a string to a command processor (Unimplemented) (ANSI)

DESCRIPTION

This library includes several standard ANSI routines. Note that where there is a pair of routines, such as **div( )** and **div_r( )**, only the routine **xxx_r( )** is reentrant. The **xxx( )** routine is not reentrant.

The header **stdlib.h** declares four types and several functions of general utility, and defines several macros.

Types

The types declared are **size_t**, **wchar_t**, and:

div_t
is the structure type of the value returned by the **div( )**.

ldiv_t
is the structure type of the value returned by the **ldiv_t( )**.

Macros

The macros defined are **NULL** and:

EXIT_FAILURE, **EXIT_SUCCESS**
expand to integral constant expressions that may be used as the argument to **exit( )** to return unsuccessful or successful termination status, respectively, to the host environment.

RAND_MAX
expands to a positive integer expression whose value is the maximum number of bytes on a multibyte character for the extended character set specified by the current locale, and whose value is never greater than **MB_LEN_MAX**.

INCLUDE FILES

stdlib.h

SEE ALSO

American National Standard X3.159-1989

ansiString

NAME

ansiString – ANSI **string** documentation

ROUTINES

memchr( ) - search a block of memory for a character (ANSI)
memcmp( ) - compare two blocks of memory (ANSI)
memcpy( ) - copy memory from one location to another (ANSI)
memmove( ) - copy memory from one location to another (ANSI)
memset( ) - set a block of memory (ANSI)
strcat( ) - concatenate one string to another (ANSI)
strchr( ) - find the first occurrence of a character in a string (ANSI)
strcmp( ) - compare two strings lexicographically (ANSI)
strcoll( ) - compare two strings as appropriate to **LC_COLLATE** (ANSI)
strcpy( ) - copy one string to another (ANSI)
strcspn( ) - return the string length up to the first character from a given set (ANSI)
strerror_r( ) - map an error number to an error string (POSIX)
strerror( ) - map an error number to an error string (ANSI)
strlen( ) - determine the length of a string (ANSI)
strncat( ) - concatenate characters from one string to another (ANSI)
strncmp( ) - compare the first *n* characters of two strings (ANSI)
strncpy( ) - copy characters from one string to another (ANSI)
strpbrk( ) - find the first occurrence in a string of a character from a given set (ANSI)
strrchr( ) - find the last occurrence of a character in a string (ANSI)
strspn( ) - return the string length up to the first character not in a given set (ANSI)
strstr( ) - find the first occurrence of a substring in a string (ANSI)
strtok( ) - break down a string into tokens (ANSI)
strtok_r( ) - break down a string into tokens (reentrant) (POSIX)
strxfrm( ) - transform up to *n* characters of *s2* into *s1* (ANSI)

DESCRIPTION

This library includes several standard ANSI routines. Note that where there is a pair of routines, such as **did( )** and **div_r( )**, only the routine **xxx_r( )** is reentrant. The **xxx( )** routine is not reentrant.

The header **string.h** declares one type and several functions, and defines one macro useful for manipulating arrays of character type and other objects treated as array of character type. The type is **size_t** and the macro **NULL**. Various methods are used for determining the lengths of the arrays, but in all cases a **char *** or **void *** argument points to the initial (lowest addressed) character of the array. If an array is accessed beyond the end of an object, the behavior is undefined.

SEE ALSO

American National Standard X3.159-1989

A

ansiTime

NAME

ansiTime – ANSI **time** documentation

ROUTINES

asctime( ) - convert broken-down time into a string (ANSI)
asctime_r( ) - convert broken-down time into a string (POSIX)
clock( ) - determine the processor time in use (ANSI)
ctime( ) - convert time in seconds into a string (ANSI)
ctime_r( ) - convert time in seconds into a string (POSIX)
difftime( ) - compute the difference between two calendar times (ANSI)
gmtime( ) - convert calendar time into UTC broken-down time (ANSI)
gmtime_r( ) - convert calendar time into broken-down time (POSIX)
localtime( ) - convert calendar time into broken-down time (ANSI)
localtime_r( ) - convert calendar time into broken-down time (POSIX)
mktime( ) - convert broken-down time into calendar time (ANSI)
strftime( ) - convert broken-down time into a formatted string (ANSI)
time( ) - determine the current calendar time (ANSI)

DESCRIPTION

The header **time.h** defines two macros and declares four types and several functions for manipulating time. Many functions deal with a **calendar time** that represents the current date (according to the Gregorian calendar) and time. Some functions deal with **local time**, which is the calendar time expressed for some specific time zone, and with Daylight Saving Time, which is a temporary change in the algorithm for determining local time. The local time zone and Daylight Saving Time are implementation-defined.

Macros

The macros defined are **NULL** and:

CLOCKS_PER_SEC
the number of ticks per second.

Types

The types declared are **size_t** and:

clock_t, **time_t**
arithmetic types capable of representing times.

struct tm
holds the components of a calendar time in what is known as "broken-down time." The structure contains at least the following members, in any order. The semantics of the members and their normal ranges are expressed in the comments.

int **tm_sec**;	seconds after the minute	- [0, 59]
int **tm_min**;	minutes after the hour	- [0, 59]
int **tm_hour**;	hours after midnight	- [0, 23]
int **tm_mday**;	day of the month	- [1, 31]
int **tm_mon**;	months since January	- [0, 11]
int **tm_year**;	years since 1900	
int **tm_wday**;	days since Sunday	- [0, 6]
int **tm_yday**;	days since January 1	- [0, 365]
int **tm_isdst**;	Daylight Saving Time flag	

The value of **tm_isdst** is positive if Daylight Saving Time is in effect, zero if Daylight Saving Time is not in effect, and negative if the information is not available.

If the environment variable TIMEZONE is set, the information is retrieved from this variable, otherwise from the locale information. TIMEZONE is of the form:

name_of_zone:<(unused)<:*time_in_minutes_from_UTC*:*daylight_start*:*daylight_end*

To calculate local time, the value of *time_in_minutes_from_UTC* is subtracted from UTC; *time_in_minutes_from_UTC* must be positive.

Daylight information is expressed as mmddhh (month-day-hour), for example:

```
UTC::0:040102:100102
```

REENTRANCY
Where there is a pair of routines, such as **div( )** and **div_r( )**, only the routine **xxx_r( )** is reentrant. The **xxx( )** routine is not reentrant.

INCLUDE FILES
time.h

SEE ALSO
ansiLocale, American National Standard X3.159-1989

arpLib

NAME
arpLib – Address Resolution Protocol (ARP) table manipulation library

ROUTINES
arpAdd( ) - create or modify an ARP table entry
arpDelete( ) - remove an ARP table entry
arpFlush( ) - flush all entries in the system ARP table
arpResolve( ) - resolve a hardware address for a specified Internet address

DESCRIPTION
This library provides direct access to the address translation table maintained by the Address Resolution Protocol (ARP). Each entry in the table maps an Internet Protocol (IP)

address to a physical hardware address. This library supports only those entries that translate between IP and Ethernet addresses. It is linked into the VxWorks image if **INCLUDE_ARP** is defined at the time the image is built. The underlying ARP protocol, which creates and maintains the table, is included automatically as part of the IP component.

RELATED INTERFACES

The **arpShow( )** routine (in the **netShow** library) displays the current contents of the ARP table.

A low -level interface to the ARP table is available with the socket-specific **SIOCSARP**, **SIOCDARP** and **SIOCGARP** ioctl functions.

INCLUDE FILES **arpLib.h**

SEE ALSO **inetLib**, **routeLib**, **netShow**

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 (because of alignment problems inherent in the MC68000), 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

string.h

SEE ALSO

ansiString

bootConfig

NAME

bootConfig – system configuration module for boot ROMs

ROUTINES

No Callable Routines

DESCRIPTION

This is the WRS-supplied configuration module for the VxWorks boot ROM. It is a stripped-down version of **usrConfig.c**, having no VxWorks shell or debugging facilities. Its primary function is to load an object module over the network with either RSH or FTP. Additionally, a simple set of single letter commands is provided for displaying and modifying memory contents. Use this module as a starting point for placing applications in ROM.

bootInit

NAME **bootInit** – ROM initialization module

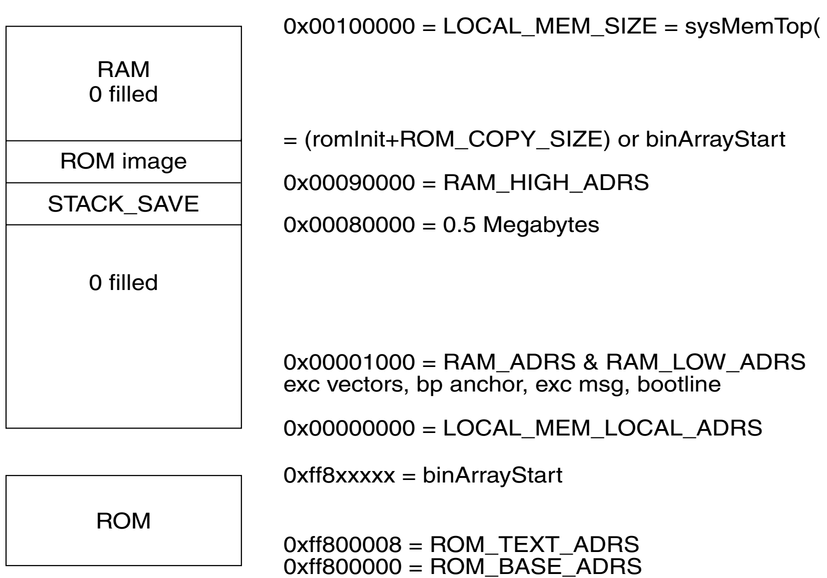
ROUTINES **romStart( )** - generic ROM initialization

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

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

ROM AND RAM MEMORY LAYOUT

Example memory layout for a 1-megabyte board:



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

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

B

bootLib

NAME

bootLib – boot ROM subroutine library

ROUTINES

bootStringToStruct( ) - interpret the boot parameters from the boot line
bootStructToString( ) - construct a boot line
bootParamsShow( ) - display boot line parameters
bootParamsPrompt( ) - prompt for boot line parameters
bootLeaseExtract( ) - extract the lease information from an Internet address
bootNetmaskExtract( ) - extract the net mask field from an Internet address
bootBpAnchorExtract( ) - extract a backplane address from a device field

DESCRIPTION

This library contains routines for manipulating a boot line. Routines are provided to interpret, construct, print, and prompt for a boot line.

When VxWorks is first booted, certain parameters can be specified, such as network addresses, boot device, host, and start-up file. This information is encoded into a single ASCII string known as the boot line. The boot line is placed at a known address (specified in **config.h**) by the boot ROMs so that the system being booted can discover the parameters that were used to boot the system. The boot line is the only means of communication from the boot ROMs to the booted system.

The boot line is of the form:

`bootdev(`*unitnum***`,`***procnum***`)`***hostname***`:`***filename* **`e=#`** **`b=#`** **`h=#`** **`g=#`** **`u=`***userid* **`pw=`***passwd* **`f=#`**
`tn=`*targetname* **`s=`***startupscript* **`o=`***other*

where:

bootdev
: the boot device (required); for example, “ex” for Excelan Ethernet, “bp” for backplane. For the backplane, this field can have an optional anchor address specification of the form “bp=*adrs*” (see **bootBpAnchorExtract( )**).

unitnum
: the unit number of the boot device (0..n).

procnum
: the processor number on the backplane, 0..n (required for VME boards).

hostname
: the name of the boot host (required).

filename
: the file to be booted (required).

e
: the Internet address of the Ethernet interface. This field can have an optional subnet mask of the form *inet_adrs*:*subnet_mask*. If DHCP is used to obtain the configuration

parameters, lease timing information may also be present. This information takes the form *lease_duration*:*lease_origin* and is appended to the end of the field. (see **bootNetmaskExtract( )** and **bootLeaseExtract( )**).

b
: the Internet address of the backplane interface. This field can have an optional subnet mask and/or lease timing information as "e".

h
: the Internet address of the boot host.

g
: the Internet address of the gateway to the boot host. Leave this parameter blank if the host is on same network.

u
: a valid user name on the boot host.

pw
: the password for the user on the host. This parameter is usually left blank. If specified, FTP is used for file transfers.

f
: the system-dependent configuration flags. This parameter contains an **or** of option bits defined in **sysLib.h**.

tn
: the name of the system being booted

s
: the name of a file to be executed as a start-up script.

o
: "other" string for use by the application.

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

EXAMPLE

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

INCLUDE FILES **bootLib.h**

SEE ALSO **bootConfig**

bootpLib

NAME **bootpLib** – Bootstrap Protocol (BOOTP) client library

ROUTINES
bootpLibInit( ) - BOOTP client library initialization
bootpParamsGet( ) - retrieve boot parameters using BOOTP
bootpMsgGet( ) - send a BOOTP request message and retrieve reply

DESCRIPTION This library implements the client side of the Bootstrap Protocol (BOOTP). This protocol allows a host to initialize automatically by obtaining its IP address, boot file name, and boot host's IP address over a network. The **bootpLibInit( )** routine links this library into the VxWorks image. This happens automatically if **INCLUDE_BOOTP** is defined at the time the image is built.

CONFIGURATION INTERFACE

When used during boot time, the BOOTP library attempts to retrieve the required configuration information from a BOOTP server using the interface described below. If it is successful, the remainder of the boot process continues as if the information were entered manually.

HIGH-LEVEL INTERFACE

The **bootpParamsGet( )** routine retrieves a set of configuration parameters according to the client-server interaction described in RFC 951 and clarified in RFC 1542. The parameter descriptor structure it accepts as an argument allows the retrieval of any combination of the options described in RFC 1533 (if supported by the BOOTP server and specified in the database). During the default system boot process, the routine obtains the boot file, the Internet address, and the host Internet address. It also obtains the subnet mask and the Internet address of an IP router, if available.

LOW-LEVEL INTERFACE

The **bootpMsgGet( )** routine transmits an arbitrary BOOTP request message and provides direct access to any reply. This interface provides a method for supporting alternate BOOTP implementations which may not fully comply with the recommended behavior in RFC 1542. For example, it allows transmission of BOOTP messages to an arbitrary UDP port and provides access to the vendor-specific field to handle custom formats which differ from the RFC 1533 implementation. The **bootpParamsGet( )** routine already extracts all options which that document defines.

EXAMPLE The following code fragment demonstrates use of the BOOTP library:

```
#include "bootpLib.h"
#define _MAX_BOOTP_RETRIES  1
struct bootpParams  bootParams;
struct in_addr      clntAddr;
```

```
struct in_addr      hostAddr;
char                bootFile [SIZE_FILE];
int                 subnetMask;
struct in_addr_list routerList;
struct in_addr      gateway;
struct ifnet *      pIf;
/* Retrieve the interface descriptor of the transmitting device. */
pIf = ifunit ("ln0");
if (pIf == NULL)
    {
    printf ("Device not found.\n");
    return (ERROR);
    }
/* Setup buffers for information from BOOTP server. */
bzero ( (char *)&clntAddr, sizeof (struct in_addr));
bzero ( (char *)&hostAddr, sizeof (struct in_addr));
bzero (bootFile, SIZE_FILE);
subnetMask  = 0;
bzero ( (char *)&gateway, sizeof (struct in_addr));
/* Set all pointers in parameter descriptor to NULL. */
bzero ((char *)&bootParams, sizeof (struct bootpParams));
/* Set pointers corresponding to desired options. */
bootParams.netmask = (struct in_addr *)&subnetMask;
routerlist.addr = &gateway;
routerlist.num = 1;
bootParams.routers = &routerlist;
/*
 @ Send request and wait for reply, retransmitting as necessary up to
 @ given limit. Copy supplied entries into buffers if reply received.
 */
result = bootpParamsGet (pIf, _MAX_BOOTP_RETRIES,
                      &clntAddr, &hostAddr, NULL, bootFile, &bootParams);
if (result != OK)
    return (ERROR);
```

INCLUDE FILES
bootpLib.h

SEE ALSO
RFC 951, RFC 1542, RFC 1533,

B

bpfDrv

NAME **bpfDrv** – Berkeley Packet Filter (BPF) I/O driver library

ROUTINES **bpfDrv( )** - initialize the BPF driver
bpfDevCreate( ) - create Berkeley Packet Filter device
bpfDevDelete( ) - destroy Berkeley Packet Filter device

DESCRIPTION This library provides a driver which supports the customized retrieval of incoming network data that meets the criteria imposed by a user-specified filter.

USER-CALLABLE ROUTINES

The **bpfDrv( )** routine initializes the driver and the **bpfDevCreate( )** routine creates a packet filter device. Each BPF device allows direct access to the incoming data from one or more network interfaces.

CREATING BPF DEVICES

In order to retrieve incoming network data, a BPF device must be created by calling the **bpfDevCreate( )** routine:

```
STATUS bpfDevCreate
    (
    char *  pDevName,       /* I/O system device name */
    int     numUnits,       /* number of device units */
    int     bufSize         /* block size for the BPF device */
    )
```

The *numUnits* parameter specifies the maximum number of BPF units for the device. Each unit is accessed through a separate file descriptor for use with a unique filter and/or a different network interface. For example, the following call creates the **/bpf0** and **/bpf1** units:

```
bpfDevCreate ("/bpf", 2, 4096);
```

CONFIGURING BPF DEVICES

After opening a device unit, the associated file descriptor must be bound to a specific network interface with the **BIOCSETIF** **ioctl( )** option. The **BIOCSETF** **ioctl( )** option adds any filter instructions. Each file descriptor receives a copy of any data which matches the filter. Different file descriptors may share the same interface. The underlying filters will receive an identical data stream.

IOCTL FUNCTIONS The BPF driver supports the following **ioctl( )** functions:

NOTE: When reading data from BPF units, the supplied buffer must be able to accept an entire block of data as defined by the *bufSize* parameter to the **bpfDevCreate( )** routine. That value is also available with the **BIOCGBLEN ioctl( )** option described above.

INCLUDE FILES **bpfDrv.h**

SEE ALSO **ioLib**

cache4kcLib

NAME **cache4kcLib – MIPS 4kc cache management library**

ROUTINES **cache4kcLibInit( )** - initialize the 4kc cache library

DESCRIPTION This library contains architecture-specific cache library functions for the MIPS 4kc architecture. The 4kc utilizes a variable-size instruction and data cache that operates in write-through mode. Cache line size also varies.

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

INCLUDE FILES **cacheLib.h**

SEE ALSO **cacheLib**

cacheArchLib

NAME **cacheArchLib** – architecture-specific cache management library

ROUTINES
cacheArchLibInit( ) - initialize the cache library
cacheArchClearEntry( ) - clear an entry from a cache (68K, x86)
cacheStoreBufEnable( ) - enable the store buffer (MC68060 only)
cacheStoreBufDisable( ) - disable the store buffer (MC68060 only)

DESCRIPTION This library contains architecture-specific cache library functions for the following processor cache families: Motorola 68K, Intel 960, Intel x86, PowerPC, ARM, and the Solaris and Windows simulators. Each routine description indicates which architecture families support it. Within families, different members support different cache mechanisms; thus, some operations cannot be performed by certain processors because they lack particular functionalities. In such cases, the routines in this library return **ERROR**. Processor-specific constraints are addressed in the manual entries for routines in this library. If the caches are unavailable or uncontrollable, the routines return **ERROR**. The exception to this rule is the 68020; although the 68020 has no cache, data cache operations return **OK**.

The MIPS architecture families have cache-related routines in individual BSP libraries. See the reference pages for the individual libraries and routines.

INCLUDE FILES **cacheLib.h**, **mmuLib.h** (ARM only)

SEE ALSO **cacheLib**, **vmLib**

cacheAuLib

NAME **cacheAuLib** – Alchemy Au cache management library

ROUTINES **cacheAuLibInit( )** - initialize the Au cache library

DESCRIPTION This library contains architecture-specific cache library functions for the Alchemy Au architecture. The Au utilizes a variable-size instruction and data cache that operates in write-through mode. Cache line size also varies.

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

INCLUDE FILES **cacheLib.h**

SEE ALSO **cacheLib**

cacheLib

NAME **cacheLib** – cache management library

ROUTINES
cacheLibInit( ) - initialize the cache library for a processor architecture
cacheEnable( ) - enable the specified cache
cacheDisable( ) - disable the specified cache
cacheLock( ) - lock all or part of a specified cache
cacheUnlock( ) - unlock all or part of a specified cache
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
cachePipeFlush( ) - flush processor write buffers to memory
cacheTextUpdate( ) - synchronize the instruction and data caches
cacheDmaMalloc( ) - allocate a cache-safe buffer for DMA devices and drivers
cacheDmaFree( ) - free the buffer acquired with **cacheDmaMalloc( )**
cacheDrvFlush( ) - flush the data cache for drivers
cacheDrvInvalidate( ) - invalidate data cache for drivers
cacheDrvVirtToPhys( ) - translate a virtual address for drivers
cacheDrvPhysToVirt( ) - translate a physical address for drivers

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

C

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

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

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

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

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

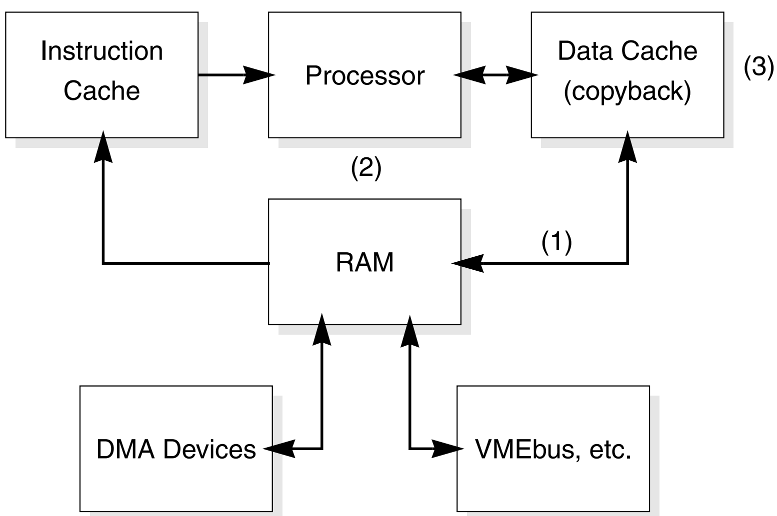
Several routines accept two additional arguments: an address and the number of bytes. Some cache operations can be applied to the entire cache (bytes = **ENTIRE_CACHE**) or to a portion of the cache. This range specification allows the cache to be selectively locked, unlocked, flushed, invalidated, and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

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

BACKGROUND

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

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



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

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

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

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

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

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

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

USING THE CACHE LIBRARY

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

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

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

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

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

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

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

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

Several flush and invalidate macros are defined in **cacheLib.h**. Since optimized code uses these macros, they provide a mechanism to avoid unnecessary cache calls and accomplish the necessary work (return **OK**). Needless work includes flushing a write-through cache, flushing or invalidating cache entries in a system with bus snooping, and flushing or invalidating cache entries in a system without caches. The macros are set to reflect the state of the cache system hardware and software. Example 1 The following example is of a simple driver that uses **cacheFlush( )** and **cacheInvalidate( )** from the cache library to maintain coherency and performance. There are two buffers (lines 3 and 4), one for input and one for output. The output buffer is obtained by the call to **memalign( )**, a special version of the well-known **malloc( )** routine (line 6). It returns a pointer that is rounded down and up to the alignment parameter's specification. Note that cache lines should not be shared, therefore **_CACHE_ALIGN_SIZE** is used to force alignment. If the memory allocator fails (line 8), the driver will typically return **ERROR** (line 9) and quit.

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

More driver code is executed (line 16), then the driver is ready to receive data that the device has placed in an input buffer in memory (line 18). Before the driver can work with the incoming data, it must invalidate the data cache entries (line 19) that correspond to the input buffer's data in order to eliminate stale entries. That done, it is safe for the driver to retrieve the input data from memory (line 21). Remember to free (line 23) the buffer acquired from the memory allocator. The driver will return **OK** (line 24) to distinguish a successful from an unsuccessful operation.

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

6:  pOutBuf = memalign (_CACHE_ALIGN_SIZE, BUF_SIZE);
8:  if (pOutBuf == NULL)
9:      return (ERROR);          /* memory allocator failed */
11: /* other driver initialization and buffer filling */
13: cacheFlush (DATA_CACHE, pOutBuf, BUF_SIZE);
14: drvWrite (pOutBuf);          /* output data to device */
16: /* more driver code */
18: cacheClear (DATA_CACHE, pInBuf, BUF_SIZE);
19: pInBuf = drvRead ();         /* wait for device data */
21: /* handle input data from device */
23: free (pOutBuf);              /* return buffer to memory pool */
24: return (OK);
    }
```

Extending this flush/invalidate concept further, individual buffers can be treated this way, not just the entire cache system. The idea is to avoid unnecessary flush and/or invalidate operations on a per-buffer basis by allocating cache-safe buffers. Calls to **cacheDmaMalloc( )** optimize the flush and invalidate function pointers to **NULL**, if possible, while maintaining data integrity. Example 2 The following example is of a high-performance driver that takes advantage of the cache library to maintain coherency. It uses **cacheDmaMalloc( )** and the macros **CACHE_DMA_FLUSH** and **CACHE_DMA_INVALIDATE**. A buffer pointer is passed as a parameter (line 2). If the pointer is not **NULL** (line 7), it is assumed that the buffer will not experience any cache coherency problems. If the driver was not provided with a cache-safe buffer, it will get one (line 11) from **cacheDmaMalloc( )**. A **CACHE_FUNCS** structure (see **cacheLib.h**) is used to create a buffer that will not suffer from cache coherency problems. If the memory allocator fails (line 13), the driver will typically return **ERROR** (line 14) and quit.

The driver fills the output buffer with initialization information, device commands, and data (line 17), and is prepared to pass the buffer to the device. Before doing so, the driver must flush the data cache (line 19) to ensure that the buffer is in memory, not hidden in the cache. The routine **drvWrite( )** lets the device know that the data is ready and where in memory it is located (line 20).

More driver code is executed (line 22), and the driver is then ready to receive data that the device has placed in the buffer in memory (line 24). Before the driver cache can work with the incoming data, it must invalidate the data cache entries (line 25) that correspond to the input buffer's data in order to eliminate stale entries. That done, it is safe for the driver to handle the input data (line 27), which the driver retrieves from memory. Remember to free the buffer (line 29) acquired from the memory allocator. The driver will return **OK** (line 30) to distinguish a successful from an unsuccessful operation.

```
STATUS drvExample2 (pBuf)       /* simple driver - great performance */
2:  void *      pBuf;           /* buffer pointer parameter */
    {
5:  if (pBuf != NULL)
        {
7:      /* no cache coherency problems with buffer passed to driver */
        }
    else
        {
11:     pBuf = cacheDmaMalloc (BUF_SIZE);
13:     if (pBuf == NULL)
14:         return (ERROR);     /* memory allocator failed */
        }
17: /* other driver initialization and buffer filling */
19: CACHE_DMA_FLUSH (pBuf, BUF_SIZE);
20: drvWrite (pBuf);            /* output data to device */
22: /* more driver code */
24: drvWait ();                 /* wait for device data */
25: CACHE_DMA_INVALIDATE (pBuf, BUF_SIZE);
27: /* handle input data from device */
29: cacheDmaFree (pBuf);        /* return buffer to memory pool */
30: return (OK);
    }
```

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

A call to **cacheLibInit( )** in write-through mode makes the flush function pointers **NULL**. Setting the caches in copyback mode (if supported) should set the pointer to and call an architecture-specific flush routine. The invalidate and flush macros may be NULLified if the hardware provides bus snooping and there are no cache coherency problems. Example 3 The next example shows a more complex driver that requires address translations to assist in the cache coherency scheme. The previous example had *a priori* knowledge of the system memory map and/or the device interaction with the memory system. This next driver demonstrates a case in which the virtual address returned by **cacheDmaMalloc( )** might differ from the physical address seen by the device. It uses the **CACHE_DMA_VIRT_TO_PHYS** and **CACHE_DMA_PHYS_TO_VIRT** macros in addition to the **CACHE_DMA_FLUSH** and **CACHE_DMA_INVALIDATE** macros.

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

More driver code is executed (line 13), and the driver is then ready to receive data that the device has placed in the buffer in memory (line 15). Note the use of the **CACHE_DMA_PHYS_TO_VIRT** macro on the buffer pointer received from the device. Before the driver cache can work with the incoming data, it must invalidate the data cache entries (line 16) that correspond to the input buffer's data in order to eliminate stale entries. That done, it is safe for the driver to handle the input data (line 17), which it retrieves from memory. Remember to free (line 19) the buffer acquired from the memory allocator. The driver will return **OK** (line 20) to distinguish a successful from an unsuccessful operation.

```
STATUS drvExample3 ()           /* complex driver - great performance */ {
3:  void * pBuf = cacheDmaMalloc (BUF_SIZE);
5:  if (pBuf == NULL)
6:      return (ERROR);         /* memory allocator failed */
8: /* other driver initialization and buffer filling */
10: CACHE_DMA_FLUSH (pBuf, BUF_SIZE);
11: drvWrite (CACHE_DMA_VIRT_TO_PHYS (pBuf));
13: /* more driver code */
15: pBuf = CACHE_DMA_PHYS_TO_VIRT (drvRead ());
16: CACHE_DMA_INVALIDATE (pBuf, BUF_SIZE);
17: /* handle input data from device */
19: cacheDmaFree (pBuf);        /* return buffer to memory pool */
20: return (OK);
    }
```

Driver Summary

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

When **cacheDmaMalloc( )** does something that makes the virtual address different from the physical address needed by the device, it provides the translation procedures. This is often the case when using translation lookaside buffers (TLB) or a segmented address space to inhibit caching (*e.g.*, by creating a different virtual address for the same physical space.) If the virtual address returned by **cacheDmaMalloc( )** is the same as the physical address, the function pointers are made **NULL** so that no calls are made when the macros are expanded. Board Support Packages Each board for an architecture with more than one cache implementation has the potential for a different cache system. Hence the BSP for selecting the appropriate cache library. The function pointer **sysCacheLibInit** is set to **cacheXxxLibInit( )** ("Xxx" refers to the chip-specific name of a library or function) so that the function pointers for that cache system will be initialized and the linker will pull in only the desired cache library. Below is an example of **cacheXxxLib** being linked in by **sysLib.c**. For systems without caches and for those architectures with only one cache design, there is no need for the **sysCacheLibInit** variable.

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

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

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

There may be some drivers that require numerous cache calls, so many that they interfere with the code clarity. Additional checking can be done at the initialization stage to determine if **cacheDmaMalloc( )** returned a buffer in non-cacheable space. Remember that it will return a cache-safe buffer by virtue of the function pointers. Ideally, these are **NULL**, since the MMU was used to mark the pages as non-cacheable. The macros **CACHE_Xxx_IS_WRITE_COHERENT** and **CACHE_Xxx_IS_READ_COHERENT** can be used to check the flush and invalidate function pointers, respectively.

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

INCLUDE FILES **cacheLib.h**

SEE ALSO Architecture-specific cache-management libraries (**cacheXxxLib**), **vmLib**, *VxWorks Programmer's Guide: I/O System*

cacheR3kLib

NAME **cacheR3kLib** – MIPS R3000 cache management library

ROUTINES **cacheR3kLibInit( )** - initialize the R3000 cache library

DESCRIPTION This library contains architecture-specific cache library functions for the MIPS R3000 architecture. The R3000 utilizes a variable-size instruction and data cache that operates in write-through mode. Cache line size also varies. Cache tags may be invalidated on a per-word basis by execution of a byte write to a specified word while the cache is isolated. See also the manual entry for **cacheR3kALib**.

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

INCLUDE FILES **cacheLib.h**

SEE ALSO **cacheR3kALib**, **cacheLib**, Gerry Kane: *MIPS R3000 RISC Architecture*

cacheR4kLib

NAME **cacheR4kLib** – MIPS R4000 cache management library

ROUTINES **cacheR4kLibInit( )** - initialize the R4000 cache library

DESCRIPTION This library contains architecture-specific cache library functions for the MIPS R4000 architecture. The R4000 utilizes a variable-size instruction and data cache that operates in write-back mode. Cache line size also varies.

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

INCLUDE FILES **cacheLib.h**

SEE ALSO **cacheLib**

cacheR5kLib

NAME **cacheR5kLib** – MIPS R5000 cache management library

ROUTINES **cacheR5kLibInit( )** - initialize the R5000 cache library

DESCRIPTION This library contains architecture-specific cache library functions for the MIPS R5000 architecture. The R5000 utilizes a variable-size instruction and data cache that operates in write-back mode. Cache line size also varies.

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

INCLUDE FILES **cacheLib.h**

SEE ALSO **cacheLib**

cacheR7kLib

NAME **cacheR7kLib** – MIPS R7000 cache management library

ROUTINES **cacheR7kLibInit( )** - initialize the R7000 cache library

DESCRIPTION This library contains architecture-specific cache library functions for the MIPS R7000 architecture. The R7000 utilizes a variable-size instruction and data cache that operates in write-back mode. Cache line size also varies.

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

INCLUDE FILES **cacheLib.h**

SEE ALSO **cacheLib**

C

cacheR10kLib

NAME **cacheR10kLib** – MIPS R10000 cache management library

ROUTINES **cacheR10kLibInit( )** - initialize the R10000 cache library

DESCRIPTION This library contains architecture-specific cache library functions for the MIPS R10000 architecture. The R10000 utilizes a variable-size instruction and data cache that operates in write-back mode. Cache line size also varies.

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

INCLUDE FILES **cacheLib.h**

SEE ALSO **cacheLib**

cacheR32kLib

NAME **cacheR32kLib** – MIPS RC32364 cache management library

ROUTINES **cacheR32kLibInit( )** - initialize the RC32364 cache library
cacheR32kMalloc( ) - allocate a cache-safe buffer, if possible

DESCRIPTION This library contains architecture-specific cache library functions for the MIPS IDT RC32364 architecture.

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

INCLUDE FILES **cacheLib.h**

SEE ALSO **cacheLib**

cacheR33kLib

NAME **cacheR33kLib** – MIPS R33000 cache management library

ROUTINES **cacheR33kLibInit( )** - initialize the R33000 cache library

DESCRIPTION This library contains architecture-specific cache library functions for the MIPS R33000 architecture. The R33000 utilizes a 8-Kbyte instruction cache and a 1-Kbyte data cache that operate in write-through mode. Cache line size is fixed at 16 bytes. Cache tags may be invalidated on a per-line basis by execution of a store to a specified line while the cache is in invalidate mode.

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

INCLUDE FILES **arch/mips/lr33000.h**, **cacheLib.h**

SEE ALSO **cacheLib**, *LSI Logic LR33000 MIPS Embedded Processor User's Manual*

cacheR333x0Lib

NAME **cacheR333x0Lib** – MIPS R333x0 cache management library

ROUTINES **cacheR333x0LibInit( )** - initialize the R333x0 cache library

DESCRIPTION This library contains architecture-specific cache library functions for the MIPS R333x0 architecture. The R33300 utilizes a 4-Kbyte instruction cache and a 2-Kbyte data cache that operate in write-through mode. The R33310 utilizes a 8-Kbyte instruction cache and a 4-Kbyte data cache that operate in write-through mode. Cache line size is fixed at 16 bytes. Cache tags may be invalidated on a per-line basis by execution of a store to a specified line while the cache is in invalidate mode.

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

INCLUDE FILES **arch/mips/lr33300.h**, **cacheLib.h**

SEE ALSO **cacheLib**, *LSI Logic LR33300 and LR33310 Self-Embedding Processors User's Manual*

C

cacheSh7040Lib

NAME

cacheSh7040Lib – Hitachi SH7040 cache management library

ROUTINES

cacheSh7040LibInit( ) - initialize the SH7040 cache library

DESCRIPTION

This library contains architecture-specific cache library functions for the Hitachi SH7040 architecture. This architecture has a 1-Kbyte instruction cache.

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

INCLUDE FILES

cacheLib.h

SEE ALSO

cacheLib

cacheSh7604Lib

NAME

cacheSh7604Lib – Hitachi SH7604/SH7615 cache management library

ROUTINES

cacheSh7604LibInit( ) - initialize the SH7604/SH7615 cache library

DESCRIPTION

This library contains architecture-specific cache library functions for the Hitachi SH7604/SH7615 instruction and data mixed cache.

INCLUDE FILES

cacheLib.h

SEE ALSO

cacheLib

cacheSh7622Lib

NAME **cacheSh7622Lib** – SH7622 cache management library

ROUTINES **cacheSh7622LibInit( )** - initialize the SH7622 cache library

DESCRIPTION This library contains architecture-specific cache library functions for the Hitachi SH7622 instruction and data caches.

INCLUDE FILES **cacheLib.h**

SEE ALSO **cacheLib**

cacheSh7700Lib

NAME **cacheSh7700Lib** – Hitachi SH7700 cache management library

ROUTINES **cacheSh7700LibInit( )** - initialize the SH7700 cache library

DESCRIPTION This library contains architecture-specific cache library functions for the Hitachi SH7700 architecture. There is a 8-Kbyte (2-Kbyte for SH7702) mixed instruction and data cache that operates in write-through or write-back (copyback) mode. The 8-Kbyte cache can be divided into 4-Kbyte cache and 4-Kbyte memory. Cache line size is fixed at 16 bytes, and the cache address array holds physical addresses as cache tags. Cache entries may be “flushed” by accesses to the address array in privileged mode. There is a write-back buffer which can hold one line of cache entry, and the completion of write-back cycle is assured by accessing to any cache through region.

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

INCLUDE FILES **cacheLib.h**

SEE ALSO **cacheLib**

C

cacheSh7729Lib

NAME **cacheSh7729Lib** – Hitachi SH7729 cache management library

ROUTINES **cacheSh7729LibInit( )** - initialize the SH7729 cache library

DESCRIPTION This library contains architecture-specific cache library functions for the Hitachi SH7729 architecture.

The cache is 16-Kbytes (16 bytes X 256 entries X 4 ways) mixed instruction and data cache that operates in write-through or write-back (copyback) mode. Cache line size is fixed at 16 bytes, and the cache address array holds physical addresses as cache tags. Cache entries may be "flushed" by accesses to the address array in privileged mode. There is a write-back buffer which can hold one line of cache entry, and the completion of write-back cycle is assured by accessing to any cache through region.

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

INCLUDE FILES **cacheLib.h**

SEE ALSO **cacheLib**

cacheSh7750Lib

NAME **cacheSh7750Lib** – Hitachi SH7750 cache management library

ROUTINES **cacheSh7750LibInit( )** - initialize the SH7750 cache library

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

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

INCLUDE FILES **cacheLib.h**

SEE ALSO **cacheLib**

cacheSun4Lib

NAME **cacheSun4Lib** – Sun-4 cache management library

ROUTINES
cacheSun4LibInit( ) - initialize the Sun-4 cache library
cacheSun4ClearLine( ) - clear a line from a Sun-4 cache
cacheSun4ClearPage( ) - clear a page from a Sun-4 cache
cacheSun4ClearSegment( ) - clear a segment from a Sun-4 cache
cacheSun4ClearContext( ) - clear a specific context from a Sun-4 cache

DESCRIPTION This library contains architecture-specific cache library functions for the Sun Microsystems Sun-4 architecture. There is a 64-Kbyte mixed instruction and data cache that operates in write-through mode. Each cache line contains 16 bytes. Cache tags may be "flushed" by accesses to alternate space in supervisor mode. Invalidate operations are performed in software by writing zero to the cache tags in an iterative manner. Tag operations are performed on "page," "segment," or "context" granularity.

MMU (Memory Management Unit) support is needed to mark pages cacheable or non-cacheable. For more information, see the manual entry for **vmLib**.

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

INCLUDE FILES **cacheLib.h**

SEE ALSO **cacheLib**, **vmLib**

cacheTx49Lib

NAME **cacheTx49Lib** – Toshiba Tx49 cache management library

ROUTINES **cacheTx49LibInit( )** - initialize the Tx49 cache library

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

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

INCLUDE FILES **cacheLib.h**

SEE ALSO **cacheLib**

cbioLib

NAME

cbioLib – cached block I/O library

ROUTINES

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

DESCRIPTION

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

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

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

CBIO MODULES AND DEVICES

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

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

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

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

TYPES OF CBIO DEVICES

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

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

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

The logical layers of a typical system using a CBIO RAM DISK are as pictured below:

```
               +--------------------+
               | Application module |
               +--------------------+ <-- read(), write(), ioctl()
                          |
               +--------------------+
               | VxWorks I/O System |
               +--------------------+ <-- IOS layer iosRead,Write,ioctl
                          |               (iosDrvInstall rtns from dosFsLib)
          +--------------- -----------+
          | File System (DOSFS/RAWFS) |
          +---------------------------+ <-- CBIO API (cbioBlkRW, cbioIoctl, etc.)
                          |
+-----------------------------------------------+
| CBIO API device driver module (ramDiskCbio.c)|
+-----------------------------------------------+
                          |
                    +----------+
                    | Hardware |
                    +----------+
```

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

```
                    +--------------------+
                    | Application module |
                    +--------------------+ <-- read(), write(), ioctl()
                               |
```

C

```
              +-------------------+
              | VxWorks IO System |
              +-------------------+ <-- IOS layer Read,Write, ioctl
                        |              (iosDrvInstall rtns from dosFsLib)
           +---------------------------+
           | File System (DOSFS/RAWFS) |
           +---------------------------+ <-- CBIO API RTNS (cbioLib.h)
                        |
        +---------------------------------+
        | CBIO to CBIO device (dpartCbio) |
        +---------------------------------+ <-- CBIO API RTNS
                        |
       +----------------------------------+
       | CBIO to CBIO device (dcacheCbio) |
       +----------------------------------+ <-- CBIO API RTNS
                        |
    +-------------------------------------------------+
    | basic CBIO to BLK_DEV wrapper device (cbioLib) |
    +-------------------------------------------------+ <-- BLK_DEV (blkIo.h)
                        |
  +--------------------------------------------------------+
  | BLK_DEV API device driver. scsiLib, ataDrv, fdDrv,etc |
  +--------------------------------------------------------+
                        |
            +-------------------------+
            | Storage Device Hardware |
            +-------------------------+
```

PUBLIC CBIO API

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

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

The following functions make up the public CBIO API:

cbioLibInit( ) - Library initialization routine

cbioBlkRW( ) - Transfer blocks (sectors) from/to a memory buffer

cbioBytesRW( ) - Transfer bytes from/to a memory buffer

cbioBlkCopy( ) - Copy directly from block to block (sector to sector)

cbioIoctl( ) - Perform I/O control operations on the CBIO device

cbioModeGet( ) - Get the CBIO device mode (**O_RDONLY**, **O_WRONLY**, or **O_RDWR**)

cbioModeSet( ) - Set the CBIO device mode (**O_RDONLY**, **O_WRONLY**, or **O_RDWR**)

cbioRdyChgdGet( ) - Determine the CBIO device ready status state

cbioRdyChgdSet( ) - Force a change in the CBIO device ready status state

cbioLock( ) - Obtain exclusive ownership of the CBIO device

cbioUnlock( ) - Release exclusive ownership of the CBIO device

cbioParamsGet( ) - Fill a **CBIO_PARAMS** structure with data from the CBIO device

cbioDevVerify( ) - Verify valid CBIO device

cbioWrapBlkDev( ) - Create CBIO wrapper atop a **BLK_DEV**

cbioShow( ) - Display information about a CBIO device

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

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

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

THE BASIC CBIO TO BLK_DEV WRAPPER MODULE

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

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

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

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

INCLUDE FILES

cbioLib.h, **cbioLibP.h**

SEE ALSO

VxWorks Programmers Guide: I/O System.

cdromFsLib

NAME

cdromFsLib – ISO 9660 CD-ROM read-only file system library

ROUTINES

cdromFsInit( ) - initialize **cdromFsLib**
cdromFsVolConfigShow( ) - show the volume configuration information
cdromFsDevCreate( ) - create a **cdromFsLib** device

DESCRIPTION

This library defines **cdromFsLib**, a utility that lets you use standard POSIX I/O calls to read data from a CD-ROM formatted according to the ISO 9660 standard file system.

It provides access to CD-ROM file systems using any standard **BLOCK_DEV** structure (that is, a disk-type driver).

The basic initialization sequence is similar to installing a DOS file system on a SCSI device.

1. Initialize the cdrom file system library (preferably in **sysScsiConfig( )** in **sysScsi.c**):

```
cdromFsInit ();
```

2. Locate and create a SCSI physical device:

```
pPhysDev=scsiPhysDevCreate(pSysScsiCtrl,0,0,0,NONE,1,0,0);
```

3. Create a SCSI block device on the physical device:

```
pBlkDev = (SCSI_BLK_DEV *) scsiBlkDevCreate (pPhysDev, 0, 0);
```

4. Create a CD-ROM file system on the block device:

```
cdVolDesc = cdromFsDevCreate ("cdrom:", (BLK_DEV *) pBlkDev);
```

Call **cdromFsDevCreate( )** once for each CD-ROM drive attached to your target. After the successful completion of **cdromFsDevCreate( )**, the CD-ROM file system will be available like any DOS file system, and you can access data on the named CD-ROM device using **open( )**, **close( )**, **read( )**, **ioctl( )**, **readdir( )**, and **stat( )**. A **write( )** always returns an error.

The **cdromFsLib** utility supports multiple drives, concurrent access from multiple tasks, and multiple open files.

FILE AND DIRECTORY NAMING

The strict ISO 9660 specification allows only uppercase file names consisting of 8 characters plus a 3 character suffix. To support multiple versions of the same file, the ISO 9660 specification also supports version numbers. When specifying a file name in an **open( )** call, you can select the file version by appending the file name with a semicolon (;) followed by a decimal number indicating the file version. If you omit the version number, **cdromFsLib** opens the latest version of the file.

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

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

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

IOCTL CODES SUPPORTED

FIOGETNAME
Returns the file name for a specific file descriptor.

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

FIOWHERE
Determines the current file position.

FIOSEEK
Changes the current file position.

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

FIOREADDIR
Reads the next directory entry.

FIODISKCHANGE
Announces that a disk has been replaced (in case the block driver is not able to provide this indication).

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

FIOFSTATGET
Gets the file status information (directory entry data).

MODIFYING A BSP TO USE CDROMFS

The following example describes mounting cdromFS on a SCSI device.

Edit your BSP's **config.h** to make the following changes:

1. Insert the following macro definition:

```
#define INCLUDE_CDROMFS
```

2. Change **FALSE** to **TRUE** in the section under the following comment:

```
/* change FALSE to TRUE for SCSI interface */
```

Make the following changes in **sysScsi.c** (or **sysLib.c** if your BSP has no **sysScsi.c**):

The main goal of the above code fragment is to call **cdromFsDevCreate( )**. As input, **cdromFsDevCreate( )** expects a pointer to a block device. In the example above, the **scsiPhysDevCreate( )** and **scsiBlkDevCreate( )** calls set up a block device interface for a SCSI CD-ROM device.

After the successful completion of **cdromFsDevCreate( )**, the device called "cdrom" is accessible using the standard **open( )**, **close( )**, **read( )**, **ioctl( )**, **readdir( )**, and **stat( )** calls.

INCLUDE FILES **cdromFsLib.h**

CAVEATS The **cdromFsLib** utility does not support CD sets containing multiple disks.

SEE ALSO **ioLib**, ISO 9660 Specification

---

clockLib

NAME **clockLib** – clock library (POSIX)

ROUTINES

clock_getres( ) - get the clock resolution (POSIX)
clock_setres( ) - set the clock resolution
clock_gettime( ) - get the current time of the clock (POSIX)
clock_settime( ) - set the clock to a specified time (POSIX)

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 Only one *clock_id* is supported, the required **CLOCK_REALTIME**. Conceivably, additional "virtual" clocks could be supported, or support for additional auxiliary clock hardware (if available) could be added.

INCLUDE FILES **timers.h**

SEE ALSO IEEE *VxWorks Programmer's Guide: Basic OS,* POSIX 1003.1b documentation

cplusLib

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

ROUTINES
cplusCallNewHandler( ) - call the allocation failure handler (C++)
cplusCtors( ) - call static constructors (C++)
cplusCtorsLink( ) - call all linked static constructors (C++)
cplusDemanglerSet( ) - change C++ demangling mode (C++)
cplusDemanglerStyleSet( ) - change C++ demangling style (C++)
cplusDtors( ) - call static destructors (C++)
cplusDtorsLink( ) - call all linked static destructors (C++)
cplusLibInit( ) - initialize the C++ library (C++)
cplusXtorSet( ) - change C++ static constructor calling strategy (C++)
operator delete( ) - default run-time support for memory deallocation (C++)
operator new( ) - default run-time support for operator new (C++)
operator new( ) - default run-time support for operator new (nothrow) (C++)
operator new( ) - run-time support for operator new with placement (C++)
set_new_handler( ) - set **new_handler** to user-defined function (C++)
set_terminate( ) - set terminate to user-defined function (C++)

DESCRIPTION This library provides run-time support and shell utilities that support the development of VxWorks applications in C++. The run-time support can be broken into three categories:

– Support for C++ new and delete operators.

– Support for initialization and cleanup of static objects.

Shell utilities are provided for:

– Resolving overloaded C++ function names.

– Hiding C++ name mangling, with support for terse or complete name demangling.

– Manual or automatic invocation of static constructors and destructors.

The usage of **cplusLib** is more fully described in the *VxWorks Programmer's Guide: C++ Development*.

SEE ALSO *VxWorks Programmer's Guide: C++ Development*

dbgArchLib

NAME

dbgArchLib – architecture-dependent debugger library

ROUTINES

a0( ) - return the contents of register **a0** (also **a1** - **a7**) (68K)
d0( ) - return the contents of register **d0** (also **d1** - **d7**) (68K)
sr( ) - return the contents of the status register (68K, SH)
dbgBpTypeBind( ) - bind a breakpoint handler to a breakpoint type (MIPS)
edi( ) - return the contents of register **edi** (also **esi** - **eax**) (x86)
eflags( ) - return the contents of the status register (x86)
r0( ) - return the contents of register **r0** (also **r1** - **r14**) (ARM)
cpsr( ) - return the contents of the current processor status register (ARM)
psrShow;1( ) - display the meaning of a specified PSR value, symbolically (ARM)
r0( ) - return the contents of general register **r0** (also **r1**-**r15**) (SH)
sr( ) - return the contents of control register **sr** (also **gbr**, **vbr**) (SH)
mach( ) - return the contents of system register **mach** (also **macl**, **pr**) (SH)
o0( ) - return the contents of register o0 (also o1-o7) (SimSolaris)
l0( ) - return the contents of register l0 (also l1-l7) (SimSolaris)
i0( ) - return the contents of register i0 (also i1-i7) (SimSolaris)
npc( ) - return the contents of the next program counter (SimSolaris)
psr( ) - return the contents of the processor status register (SimSolaris)
wim( ) - return the contents of the window invalid mask register (SimSolaris)
y( ) - return the contents of the y register (SimSolaris)
edi( ) - return the contents of register **edi** (also **esi** - **eax**) (x86/SimNT)
eflags( ) - return the contents of the status register (x86/SimNT)

DESCRIPTION

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

MC680x0:

> **a0( )** - **a7( )** - address registers (**a0** - **a7**)
>
> **d0( )** - **d7( )** - data registers (**d0** - **d7**)
>
> **sr( )** - status register (**sr**)

MIPS:

> **dbgBpTypeBind( )** - bind a breakpoint handler to a breakpoint type

x86/SimNT:

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

SH:

r0( ) - **r15( )** - general registers (**r0** - **r15**)

sr( ) - status register (**sr**)

gbr( ) - global base register (**gbr**)

vbr( ) - vector base register (**vbr**)

mach( ) - multiply and accumulate register high (**mach**)

macl( ) - multiply and accumulate register low (**macl**)

pr( ) - procedure register (**pr**)

ARM:

r0( ) - **r14( )** - general-purpose registers (**r0** - **r14**)

cpsr( ) - current processor status reg (**cpsr**)

psrShow( ) - **psr** value, symbolically

SimSolaris:

g0( ) - **g7( )** - global registers (**g0** - **g7**)

o0( ) - **o7( )** - out registers (**o0** - **o7**, note lower-case “o”)

l0( ) - **l7( )** - local registers (**l0** - **l7**, note lower-case “l”)

i0( ) - **i7( )** - in registers (**i0** - **i7**)

npc( ) - next program counter (**npc**)

psr( ) - processor status register (**psr**)

wim( ) - window invalid mask (**wim**)

y( ) - **y** register

NOTE: The routine **pc( )**, for accessing the program counter, is found in **usrLib**.

SEE ALSO **dbgLib**, *VxWorks Programmer’s Guide: Target Shell*

D

dbgLib

NAME

dbgLib – debugging facilities

ROUTINES

dbgHelp( ) - display debugging help menu
dbgInit( ) - initialize the local debugging package
b( ) - set or display breakpoints
e( ) - set or display eventpoints (WindView)
bh( ) - set a hardware breakpoint
bd( ) - delete a breakpoint
bdall( ) - delete all breakpoints
c( ) - continue from a breakpoint
cret( ) - continue until the current subroutine returns
s( ) - single-step a task
so( ) - single-step, but step over a subroutine
l( ) - disassemble and display a specified number of instructions
tt( ) - display a stack trace of a task

DESCRIPTION

This library contains VxWorks's primary interactive debugging routines, which provide the following facilities:

- task breakpoints
- task single-stepping
- symbolic disassembly
- symbolic task stack tracing

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

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

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

INITIALIZATION

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

BREAKPOINTS

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

When a task hits a breakpoint, the task is suspended and a message is displayed on the console. At this point, the task can be examined, traced, deleted, its variables changed, *etc*.

If you examine the task at this point (using the **i( )** routine), you will see that it is in a suspended state. The instruction at the breakpoint address has not yet been executed.

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

EVENTPOINTS (WINDVIEW)

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

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

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

UNBREAKABLE TASKS

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

DISASSEMBLER AND STACK TRACER

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

SHELL ABORT AND EXCEPTION HANDLING

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

THE DEFAULT TASK AND TASK REFERENCING

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

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

WARNING: When a task is continued, **c( )** and **s( )** routines do not yet distinguish between a suspended task or a task suspended by the debugger. Therefore, use of these routines should be restricted to only those tasks being debugged.

INCLUDE FILES **dbgLib.h**

SEE ALSO **excLib**, **tyLib**, **taskIdDefault( )**, **taskOptionsSet( )**, **tyAbortSet( )**, **tyAbortFuncSet( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

dcacheCbio

NAME **dcacheCbio** – disk cache driver

ROUTINES

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

DESCRIPTION

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

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

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

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

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

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

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

– Least Recently Used block re-use policy

– Read-ahead

– Write-behind with sorting and grouping

– Hidden writes

– Disk cache bypass for large requests

– Background disk updating (flushing changes to disk) with an adjustable update period (ioctl flushes occur without delay.)

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

DISK CACHE ALGORITHM

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

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

– Combining cache blocks with adjacent disk block numbers, in order to write them to disk in groups, and save on latency and overhead

– Reading ahead a group of blocks, and then converting them to normal cache blocks.

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

TUNABLE PARAMETERS

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

See **dcacheDevTune( )** for description of the tunable parameters. It is recommended to call **dcacheShow( )** after calling **dcacheTune( )** in order to verify that the parameters where set as requested, and to inspect the cache statistics which may change dramatically.

Note that the hit ratio is a principal indicator of cache efficiency, and should be inspected during such tuning.

BACKGROUND UPDATING

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

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

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

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

dcacheUpdTaskStack
: is used to set the update task stack size.

dcacheUpdTaskOptions
: controls the task options for the update task.

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

REMOVABLE DEVICES

For removable devices, disk cache provides these additional features:

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

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

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

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

CACHE IMPLEMENTATION

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

– Most Recently Used blocks are stored in RAM, which results in the most frequently accessed data being retrieved from memory rather than from disk.

– Reading data from disk is performed in large units, relying on the read-ahead feature, one of the disk cache£s tunable parameters.

Write operations are optimized because they occur to memory first. Then updating the disk happens in an orderly manner, by delayed write, another tunable parameter.

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

The technique of delaying writes to disk means that if the system is turned off unexpectedly, updates that have not yet been written to the disk are lost. To minimize the effect of a possible crash, the disk cache periodically updates the disk. Modified blocks of data are not kept in memory more then a specified period of time. By specifying a small update period, the possible worst-case loss of data from a crash is the sum of changes possible during that specified period. For example, it is assumed that an update period of 2 seconds is sufficiently large to effectively optimize disk writes, yet small enough to make the potential loss of data a reasonably minor concern. It is possible to set the update period to 0, in which case, all updates are flushed to disk immediately. This is essentially the equivalent of using the **DOS_OPT_AUTOSYNC** option in earlier **dosFsLib** implementations. The disk cache allows you to negotiate between disk performance and memory consumption: The more memory allocated to the disk cache, the higher the "hit ratio" observed, which means increasingly better performance of file system operations. Another tunable parameter is the bypass threshold, which defines how much data constitutes a request large enough to justify bypassing the disk cache. When significantly large read or write requests are made by the application, the disk cache is circumvented and there is a direct transfer of data between the disk controller and the user data buffer. The use of bypassing, in conjunction with support for contiguous file allocation and access (via the **FIOCONTIG ioctl( )** command and the **DOS_O_CONTIG open( )** flag), should provide performance equivalent to that offered by the raw file system (rawFs).

PARTITION INTERACTION

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

For example:

dosFsLib
dpart
dcache
blkIoDev

ENABLE/DISABLE THE DISK CACHE

The function **dcacheDevEnable( )** is used to enable the disk cache. The function **dcacheDevDisable( )** is used to disable the disk cache. When the disk cache is disabled, all I/O will bypass the cache layer.

SEE ALSO **dosFsLib**, **cbioLib**, **dpartCbio**

dhcpcBootLib

NAME **dhcpcBootLib** – DHCP boot-time client library

ROUTINES

dhcpcBootInit( ) - set up the DHCP client parameters and data structures
dhcpcBootBind( ) - initialize the network with DHCP at boot time
dhcpcBootInformGet( ) - obtain additional configuration parameters with DHCP

DESCRIPTION

This library contains the interface for the client side of the Dynamic Host Configuration Protocol (DHCP) used during system boot. DHCP is an extension of BOOTP, the bootstrap protocol. Like BOOTP, the protocol allows automatic system startup by providing an IP address, boot file name, and boot host's IP address over a network. Additionally, DHCP provides the complete set of configuration parameters defined in the Host Requirements RFCs and allows automatic reuse of network addresses by specifying a lease duration for a set of configuration parameters. This library is linked into the boot ROM image automatically if **INCLUDE_DHCPC** is defined at the time that image is constructed.

HIGH-LEVEL INTERFACE

The VxWorks boot program uses this library to obtain configuration parameters with DHCP according to the client-server interaction detailed in RFC 2131 using the boot device specified in the boot parameters. The DHCP client supports devices attached to the IP protocol with the MUX/END interface. It also supports BSD Ethernet devices attached to the IP protocol.

To use DHCP, first build a boot ROM image with **INCLUDE_DHCPC** defined and set the appropriate flag in the boot parameters before initiating booting with the "@" command. The DHCP client will attempt to retrieve entries for the boot file name, and host IP address, as well as a subnet mask and broadcast address for the boot device. If a target IP address is not available, the client will retrieve those parameters in the context of a lease. Otherwise, it will search for permanent assignments using a simpler message exchange. Any entries retrieved with either method will only be used if the corresponding fields in the boot parameters are blank.

NOTE: After DHCP retrieves the boot parameters, the specified boot file is loaded and the system restarts. As a result, the boot-time DHCP client cannot renew any lease which may be associated with the assigned IP address. To avoid potential IP address conflicts while loading the boot file, the **DHCPC_MIN_LEASE** value should be set to exceed the file transfer time. In addition, the boot file must also contain the DHCP client library so that the lease obtained before the restart can be renewed. Otherwise, the network initialization using the boot parameters will fail. These restrictions do not apply if the target IP address is entered manually since the boot parameters do not involve a lease in that case.

INCLUDE FILES

dhcpcBootLib.h

SEE ALSO

dhcpcLib, RFC 1541, RFC 1533

dhcpcCommonLib

NAME

dhcpcCommonLib – DHCP client interface shared code library

ROUTINES

dhcpcOptionSet( ) - add an option to the option request list
dhcpcOptionAdd( ) - add an option to the client messages

DESCRIPTION

This library contains the shared functions used by the both the run-time and boot-time portions of the DHCP client.

INCLUDE FILES

dhcpcLib.h

SEE ALSO

dhcpcLib

dhcpcLib

NAME

dhcpcLib – Dynamic Host Configuration Protocol (DHCP) run-time client API

ROUTINES

dhcpcLibInit( ) - DHCP client library initialization
dhcpcInit( ) - assign network interface and setup lease request
dhcpcEventHookAdd( ) - add a routine to handle configuration parameters
dhcpcEventHookDelete( ) - remove the configuration parameters handler
dhcpcCacheHookAdd( ) - add a routine to store and retrieve lease data
dhcpcCacheHookDelete( ) - delete a lease data storage routine
dhcpcBind( ) - obtain a set of network configuration parameters with DHCP
dhcpcVerify( ) - renew an established lease
dhcpcRelease( ) - relinquish specified lease
dhcpcInformGet( ) - obtain additional configuration parameters with DHCP
dhcpcShutdown( ) - disable DHCP client library
dhcpcOptionGet( ) - retrieve an option provided to a client and store in a buffer
dhcpcServerGet( ) - retrieve the current DHCP server
dhcpcTimerGet( ) - retrieve current lease timers
dhcpcParamsGet( ) - retrieve current configuration parameters

DESCRIPTION

This library implements the run-time access to the client side of the Dynamic Host Configuration Protocol (DHCP). DHCP is an extension of BOOTP. Like BOOTP, the protocol allows a host to initialize automatically by obtaining its IP address, boot file name, and boot host's IP address over a network. Additionally, DHCP provides a client with the complete set of parameters defined in the Host Requirements RFCs and allows automatic reuse of network addresses by specifying individual leases for each set of configuration parameters. The compatible message format allows DHCP participants to interact with BOOTP participants. The **dhcpcLibInit( )** routine links this library into the VxWorks image. This happens automatically if **INCLUDE_DHCPC** is defined at the time the image is built.

CONFIGURATION INTERFACE

When used during run time, the DHCP client library establishes and maintains one or more DHCP leases. Each lease provides access to a set of configuration parameters. If requested, the parameters retrieved will be used to reconfigure the associated network interface, but may also be handled separately through an event hook. The **dhcpcEventHookAdd( )** routine specifies a function which is invoked whenever the lease status changes. The **dhcpcEventHookDelete( )** routine will disable that notification. The automatic reconfiguration must be limited to one lease for a particular network interface. Otherwise, multiple leases would attempt to reconfigure the same device, with unpredictable results.

HIGH-LEVEL INTERFACE

To access the DHCP client during run time, an application must first call the **dhcpcInit( )** routine with a pointer to the network interface to be used for communication with a DHCP server. Each call to the initialization routine returns a unique identifier to be used in subsequent calls to the DHCP client routines. Next, the application must specify a client identifier for the lease using the **dhcpcOptionSet( )** call. Typically, the link-level hardware address is used for this purpose. Additional calls to the option set routine may be used to request specific DHCP options. After all calls to that routine are completed, a call to **dhcpcBind( )** will retrieve a set of configuration parameters according to the client-server interaction detailed in RFC 1541.

Each sequence of the three function calls described above, if successful, will retrieve a set of configuration parameters from a DHCP server. The **dhcpcServerGet( )** routine retrieves the address of the server that provided a particular lease. The **dhcpcTimerGet( )** routine will retrieve the current values for both lease timers.

Alternatively, the **dhcpcParamsGet( )** and **dhcpcOptionGet( )** routines will access any options provided by a DHCP server. In addition to the lease identifier obtained from the initialization routine, the **dhcpcParamsGet( )** routine accepts a parameter descriptor structure that selects any combination of the options described in RFC 1533 for retrieval. Similarly, the **dhcpcOptionGet( )** routine retrieves the values associated with a single option.

LOW-LEVEL INTERFACE

This library also contains several routines which explicitly generate DHCP messages. The **dhcpcVerify( )** routine causes the client to renew a particular lease, regardless of the time remaining. The **dhcpcRelease( )** routine relinquishes the specified lease. The associated parameters are no longer valid. If those parameters were used by the underlying network device, the routine also shuts off all network processing for that interface. Finally, the **dhcpcShutdown( )** routine will release all active leases and disable all the DHCP client library routines.

OPTIONAL INTERFACE

The **dhcpcCacheHookAdd( )** routine registers a function that the client will use to store and retrieve lease data. The client can then re-use this information if it is rebooted. The **dhcpcCacheHookDelete( )** routine prevents the re-use of lease data. Initially, a function to access permanent storage is not provided.

INCLUDE FILES

dhcpcLib.h

SEE ALSO

RFC 1541, RFC 1533

dhcpcShow

NAME **dhcpcShow** – DHCP run-time client information display routines

ROUTINES **dhcpcShowInit( )** - initialize the DHCP show facility
dhcpcServerShow( ) - display current DHCP server
dhcpcTimersShow( ) - display current lease timers
dhcpcParamsShow( ) - display current lease parameters

DESCRIPTION This library provides routines that display various data related to the DHCP run-time client library such as the lease timers and responding server. The **dhcpcShowInit( )** routine links the show facility into the VxWorks image. This happens automatically if **INCLUDE_NET_SHOW** and **INCLUDE_DHCPC** are defined at the time the image is built.

INCLUDE FILES **dhcpcLib.h**

SEE ALSO **dhcpcLib**

dhcprLib

NAME **dhcprLib** – DHCP relay agent library

ROUTINES No Callable Routines

DESCRIPTION This library implements a relay agent for the Dynamic Host Configuration Protocol (DHCP). DHCP is an extension of BOOTP. Like BOOTP, it allows a target to configure itself dynamically by using the network to get its IP address, a boot file name, and the DHCP server's address. The relay agent forwards DHCP messages between clients and servers resident on different subnets. The standard DHCP server, if present on a subnet, can also forward messages across subnet boundaries. The relay agent is needed only if there is no DHCP server running on the subnet. The **dhcprLibInit( )** routine links this library into the VxWorks system. This happens automatically if **INCLUDE_DHCPR** is defined at the time the system is built, as long as **INCLUDE_DHCPS** is *not* also defined.

HIGH-LEVEL INTERFACE

The **dhcprInit( )** routine initializes the relay agent automatically. The relay agent forwards incoming DHCP messages to the IP addresses specified at build time in **dhcpTargetTbl[]**.

INCLUDE FILES **dhcprLib.h**

SEE ALSO RFC 1541, RFC 1533

dhcpsLib

NAME

dhcpsLib – Dynamic Host Configuration Protocol (DHCP) server library

ROUTINES

dhcpsInit( ) - set up the DHCP server parameters and data structures
dhcpsLeaseEntryAdd( ) - add another entry to the address pool
dhcpsLeaseHookAdd( ) - assign a permanent lease storage hook for the server
dhcpsAddressHookAdd( ) - assign a permanent address storage hook for the server

DESCRIPTION

This library implements the server side of the Dynamic Host Configuration Protocol (DHCP). DHCP is an extension of BOOTP. Like BOOTP, it allows a target to configure itself dynamically by using the network to get its IP address, a boot file name, and the DHCP server's address. Additionally, DHCP provides for automatic reuse of network addresses by specifying individual leases as well as many additional options. The compatible message format allows DHCP participants to inter-operate with BOOTP participants. The **dhcpsInit( )** routine links this library into the VxWorks image. This happens automatically if **INCLUDE_DHCPS** is defined when the image is built.

PRIMARY INTERFACE

The **dhcpsInit( )** routine initializes the server. It reads the hard-coded server configuration data that is stored in three separate tables. The first table contains entries as follows:

```
DHCPS_LEASE_DESC dhcpsLeaseTbl [] =
    {
    {"sample1", "90.11.42.24", "90.11.42.24", "clid=\"1:0x08003D21FE90\""},
    {"sample2", "90.11.42.25", "90.11.42.28", "maxl=90:dfll=60"},
    {"sample3", ″90.11.42.29","90.11.42.34","maxl=0xffffffff:file=/vxWorks"},
    {"sample4", "90.11.42.24", "90.11.42.24", "albp=true:file=/vxWorks"}
    };
```

Each entry contains a name of up to eight characters, the starting and ending IP addresses of a range, and the parameters associated with the lease. The four samples shown demonstrate the four types of leases.

Manual leases contain a specific client ID, and are issued only to that client, with an infinite duration. The example shown specifies a MAC address, which is the identifier type used by the VxWorks DHCP client.

Dynamic leases specify a finite maximum length, and can be issued to any requesting client. These leases allow later re-use of the assigned IP address. If not explicitly specified in the parameters field, these leases use the values of **DHCPS_MAX_LEASE** and **DHCPS_DFLT_LEASE** to determine the lease length.

Automatic leases are implied by the infinite maximum length. Their IP addresses are assigned permanently to any requesting client.

The last sample demonstrates a lease that is also available to BOOTP clients. The infinite maximum length is implied, and any timing-related parameters are ignored.

The DHCP server supplies leases to DHCP clients according to the lease type in the order shown above. Manual leases have the highest priority and leases available to BOOTP clients the lowest.

Entries in the parameters field may be one of these types:

bool
: Takes values of "true" or "false", for example, ipfd=true. Unrecognized values default to false.

str
: Takes a character string as a value, for example, hstn="clapton". If the string includes a delimiter character, such as a colon, it should be enclosed in quotation marks.

octet
: Takes an 8-bit integer in decimal, octal, or hexadecimal, for example, 8, 070, 0xff.

short
: Takes a 16-bit integer.

long
: Takes a 32-bit integer.

ip
: Takes a string that is interpreted as a 32-bit IP address. One of the following formats is expected: a.b.c.d, **a.b.c** or a.b. In the second format, c is interpreted as a 16-bit value. In the third format, b is interpreted as a 24-bit value, for example siad=90.11.42.1.

iplist
: Takes a list of IP addresses, separated by white space, for example, rout=133.4.31.1 133.4.31.2 133.4.31.3.

ippairs
: Takes a list of IP address pairs. Each IP address is separated by white space and grouped in pairs, for example, strt=133.4.27.0 133.4.31.1 133.4.36.0 133.4.31.1.

mtpt
: Takes a list of 16 bit integers, separated by white space, for example, mtpt=1 2 3 4 6 8.

clid
: Takes a client identifier as a value. Client identifiers are represented by the quoted string "*type*:*data*", where *type* is an integer from 0 to 255, as defined by the IANA, and *data* is a sequence of 8-bit values in hexadecimal. The client ID is usually a MAC address, for example, clid="1:0x08004600e5d5".

The following table lists the option specifiers and descriptions for every possible entry in the parameter list. When available, the option code from RFC 2132 is included.

Name	Code	Type	Description
snam	-	str	Optional server name.
file	-	str	Name of file containing the boot image.
siad	-	ip	Address of server that offers the boot image.
albp	-	bool	If true, this entry is also available to BOOTP clients. For entries using static allocation, this value becomes true by default and *maxl* becomes infinity.
maxl	-	long	Maximum lease duration in seconds.
dfll	-	long	Default lease duration in seconds. If a client does not request a specific lease duration, the server uses this value.
clid	-	clid	This specifies a client identifier for manual leases. The VxWorks client uses a MAC address as the client identifier.
pmid	-	clid	This specifies a client identifier for client-specific parameters to be included in a lease. It should be present in separate entries without IP addresses.
clas	-	str	This specifies a class identifier for class-specific parameters to be included in a lease. It should be present in separate entries without IP addresses.
snmk	1	ip	Subnet mask of the IP address to be allocated. The default is a natural mask corresponding to the IP address. The server will not issue IP addresses to clients on different subnets.
tmof	2	long	Time offset from UTC in seconds.
rout	3	iplist	A list of routers on the same subnet as the client.
tmsv	4	iplist	A list of time servers (RFC 868).
nmsv	5	iplist	A list of name servers (IEN 116).
dnsv	6	iplist	A list of DNS servers (RFC 1035).
lgsv	7	iplist	A list of MIT-LCS UDP log servers.
cksv	8	iplist	A list of Cookie servers (RFC 865).
lpsv	9	iplist	A list of LPR servers (RFC 1179).
imsv	10	iplist	A list of Imagen Impress servers.
rlsv	11	iplist	A list of Resource Location servers (RFC 887).
hstn	12	str	Hostname of the client.
btsz	13	short	Size of boot image.
mdmp	14	str	Path name to which client dumps core.
dnsd	15	str	Domain name for DNS.
swsv	16	ip	IP address of swap server.
rpth	17	str	Path name of root disk of the client.
epth	18	str	Extensions Path (See RFC 1533).
ipfd	19	bool	If true, the client performs IP forwarding.
nlsr	20	bool	If true, the client can perform non-local source routing.
plcy	21	ippairs	Policy filter for non-local source routing. A list of pairs of (Destination IP, Subnet mask).

Name	Code	Type	Description
mdgs	22	short	Maximum size of IP datagram that the client should be able to reassemble.
ditl	23	octet	Default IP TTL.
mtat	24	long	Aging timeout (in seconds) to be used with Path MTU discovery (RFC 1191).
mtpt	25	mtpt	A table of MTU sizes to be used with Path MTU Discovery.
ifmt	26	short	MTU to be used on an interface.
asnl	27	bool	If true, the client assumes that all subnets to which the client is connected use the same MTU.
brda	28	ip	Broadcast address in use on the client's subnet. The default is calculated from the subnet mask and the IP address.
mskd	29	bool	If true, the client should perform subnet mask discovery using ICMP.
msks	30	bool	If true, the client should respond to subnet mask requests using ICMP.
rtrd	31	bool	If true, the client should solicit routers using Router Discovery defined in RFC 1256.
rtsl	32	ip	Destination IP address to which the client sends router solicitation requests.
strt	33	ippairs	A table of static routes for the client, which are pairs of (Destination, Router). It is illegal to specify default route as a destination.
trlr	34	bool	If true, the client should negotiate the use of trailers with ARP (RFC 893).
arpt	35	long	Timeout in seconds for ARP cache.
encp	36	bool	If false, the client uses RFC 894 encapsulation. If true, it uses RFC 1042 (IEEE 802.3) encapsulation.
dttl	37	octet	Default TTL of TCP.
kain	38	long	Interval of the client's TCP keepalive in seconds.
kagb	39	bool	If true, the client should send TCP keepalive messages with a octet of garbage for compatibility.
nisd	40	str	Domain name for NIS.
nisv	41	iplist	A list of NIS servers.
ntsv	42	iplist	A list of NTP servers.
nnsv	44	iplist	A list of NetBIOS name server. (RFC 1001, 1002)
ndsv	45	iplist	A list of NetBIOS datagram distribution servers (RFC 1001, 1002).
nbnt	46	octet	NetBIOS node type (RFC 1001, 1002).
nbsc	47	str	NetBIOS scope (RFC 1001, 1002).
xfsv	48	iplist	A list of font servers of X Window system.
xdmn	49	iplist	A list of display managers of X Window system.

Name	Code	Type	Description
dht1	58	short	This value specifies when the client should start RENEWING. The default of 500 means the client starts RENEWING after 50% of the lease duration passes.
dht2	59	short	This value specifies when the client should start REBINDING. The default of 875 means the client starts REBINDING after 87.5% of the lease duration passes.

Finally, to function correctly, the DHCP server requires access to some form of permanent storage. The **DHCPS_LEASE_HOOK** constant specifies the name of a storage routine with the following interface:

```
STATUS dhcpsStorageHook (int op, char *buffer, int datalen);
```

The storage routine is installed by a call to the **dhcpsLeaseHookAdd( )** routine The manual pages for **dhcpsLeaseHookAdd( )** describe the parameters and required operation of the storage routine.

SECONDARY INTERFACE

In addition to the hard-coded entries, address entries may be added after the server has started by calling the following routine:

```
STATUS dhcpsLeaseEntryAdd (char *name, char *start, char *end, char *config);
```

The parameters specify an entry name, starting and ending values for a block of IP addresses, and additional configuration information in the same format as shown above for the hard-coded entries. Each parameter must be formatted as a **NULL**-terminated string.

The **DHCPS_ADDRESS_HOOK** constant specifies the name of a storage routine, used to preserve address entries added after startup, which has the following prototype:

```
STATUS dhcpsAddressStorageHook (int op,
                                char *name, char *start, char *end,
                                char *params);
```

The storage routine is installed with the **dhcpsAddressHookAdd( )** routine, and is fully described in the manual pages for that function.

OPTIONAL INTERFACE

The DHCP server can also receive messages forwarded from different subnets by a relay agent. To provide addresses to clients on different subnets, the appropriate relay agents must be listed in the provided table in **usrNetwork.c**. A sample configuration is:

```
DHCPS_RELAY_DESC dhcpsRelayTbl [] =
    {
    {"90.11.46.75", "90.11.46.0"}
    };
```

Each entry in the table specifies the address of a relay agent that will transmit the request and the corresponding subnet number. To issue leases successfully, the address pool must also contain IP addresses for the monitored subnets.

The following table allows a DHCP server to act as a relay agent in addition to its default function of processing messages. It consists of a list of IP addresses.

```
DHCP_TARGET_DESC dhcpTargetTbl [] =
    {
    {"90.11.43.2"},
    {"90.11.44.1"}
    };
```

Each IP address in this list receives a copy of any client messages generated on the subnets monitored by the server.

INCLUDE FILES **dhcpsLib.h**

SEE ALSO RFC 1541, RFC 1533

---

dirLib

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

ROUTINES

opendir( ) - open a directory for searching (POSIX)
readdir( ) - 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.

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**

distIfShow

NAME **distIfShow** – distributed objects interface adapter show routines (VxFusion Opt.)

ROUTINES **distIfShow( )** - display information about the installed interface adapter (VxFusion Opt.)

DESCRIPTION This library provides a show routine for displaying information about the installed interface adapter.

AVAILABILITY This module is distributed as a component of the unbundled distributed message queues option, VxFusion.

INCLUDE FILES **distIfLib.h**

SEE ALSO **distStatLib**

distLib

NAME **distLib** – distributed objects initialization and control library (VxFusion Opt.)

ROUTINES **distInit( )** - initialize and bootstrap the current node (VxFusion Opt.)
distCtl( ) - perform a distributed objects control function (VxFusion Opt.)

DESCRIPTION This library provides an initialization and control interface for VxFusion.

Use **distInit( )** to initialize VxFusion on the current node. In addition to performing local initialization, **distInit( )** attempts to locate remote VxFusion nodes on the network and download copies of the databases from one of the remote nodes.

Call **distCtl( )** to set VxFusion run-time parameters using an **ioctl( )**-like syntax.

NOTE: In this release, the **distInit( )** routine is called automatically with default parameters when a target boots using a VxWorks image with VxFusion installed.

AVAILABILITY This module is distributed as a component of the unbundled distributed message queues option, VxFusion.

INCLUDE FILES **distLib.h**

distNameLib

NAME **distNameLib** – distributed name database library (VxFusion Opt.)

ROUTINES
distNameAdd( ) - add an entry to the distributed name database (VxFusion Opt.)
distNameFind( ) - find an object by name in the local database (VxFusion Opt.)
distNameFindByValueAndType( ) - look up the name of an object by value and type (VxFusion Opt.)
distNameRemove( ) - remove an entry from the distributed name database (VxFusion Opt.)

DESCRIPTION This library contains the distributed objects distributed name database and routines for manipulating it. Symbolic names are bound to values, such as message queue identifiers or simple integers. Entries can be found by name or by value and type. The distributed name database is replicated throughout the system, with a copy sitting on each node.

The distributed name database library is initialized by calling **distInit( )** in **distLib**.

AVAILABILITY This module is distributed as a component of the unbundled distributed message queues option, VxFusion.

INCLUDE FILES **distNameLib.h**

SEE ALSO **distLib**, **distNameShow**

D

distNameShow

NAME **distNameShow** – distributed name database show routines (VxFusion Opt.)

ROUTINES **distNameShow( )** - display the entire distributed name database (VxFusion Opt.)
distNameFilterShow( ) - display the distributed name database filtered by type (VxFusion Opt.)

DESCRIPTION This library provides routines for displaying the contents of the distributed name database.

AVAILABILITY This module is distributed as a component of the unbundled distributed message queues option, VxFusion.

INCLUDE FILES **distNameLib.h**

SEE ALSO **distNameLib**

distTBufLib

NAME **distTBufLib** – distributed objects telegram buffer library (VxFusion Opt.)

ROUTINES **distTBufAlloc( )** - allocate a telegram buffer from the pool of buffers (VxFusion Opt.)
distTBufFree( ) - return a telegram buffer to the pool of buffers (VxFusion Opt.)

DESCRIPTION This library provides routines for allocating and freeing telegram buffers. Telegrams are the largest packets that can be sent between nodes by the distributed objects product; their size is limited by the MTU size of the underlying communications. If a distributed objects message exceeds the space allocated in a telegram for message data, that message is divided into multiple telegrams that are sent out in sequence.

AVAILABILITY This module is distributed as a component of the unbundled distributed message queues option, VxFusion.

INCLUDE FILES **distTBufLib.h**

dosFsFmtLib

NAME

dosFsFmtLib – MS-DOS media-compatible file system formatting library

ROUTINES

dosFsVolFormat( ) - format an MS-DOS compatible volume

DESCRIPTION

This module is a scalable companion module for **dosFsLib**, and is intended to facilitate high level formatting of disk volumes.

There are two ways to high level format a volume:

(1) Directly calling **dosFsVolFormat( )** routine allows to have complete control over the format used, parameters and allows to supply a hook routine which for instance could interactively prompt the user to modify disk parameters.

(2) Calling ioctl command **FIODISKINIT** will invoke the formatting routine via **dosFsLib**. This uses the default volume format and parameters.

AVAILABILITY

This routine is an optional part of the MS-DOS file system, and may be included in a target system if it is required to be able to format new volumes.

In order to include this option, the following function needs to be invoked during system initialization:

```
void dosFsFmtLibInit( void );
```

See reference page **dosFsVolFormat( )** for complete description of supported formats, options and arguments.

SEE ALSO

dosFsLib

dosFsLib

NAME

dosFsLib – MS-DOS media-compatible file system library

ROUTINES

dosSetVolCaseSens( ) - set case sensitivity of volume
dosFsVolDescGet( ) - convert a device name into a DOS volume descriptor pointer.
dosFsChkDsk( ) - make volume integrity checking.
dosFsLastAccessDateEnable( ) - enable last access date updating for this volume
dosFsLibInit( ) - prepare to use the dosFs library
dosFsDevCreate( ) - create file system device.
dosFsShow( ) - display dosFs volume configuration data.

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

USING THIS LIBRARY

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

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

Another dosFs routine is used for device initialization. For each dosFs device, **dosFsDevCreate( )** must be called to install the device in VxWorks device list. In the case where partitioned disks are used, **dosFsDevCreate( )** must be called for each partition that is anticipated, thereby it is associated with a logical device name, so it can be later accessed via the I/O system.

In case of a removable disk, **dosFsDevCreate( )** must be called during system initialization time, even if a cartridge or diskette may be absent from the drive at boot time. **dosFsDevCreate( )** will only associate the device with a logical device name. Device access will be done only when the logical device is first accessed by the application.

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

INITIALIZING DOSFSLIB

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

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

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

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

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

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

dosFsFmtLibInit( )
(optional) install the volume formatting module.

dosChkLibInit( )
(optional) install the file system consistency checking module.

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

Also, at least one *CBIO* module must be initialized on a per-device basis prior to calling **dosFsDevCreate( )**. See the related documentation for more details and examples.

DEFINING A DOSFS DEVICE

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

(1) A pointer to a name string, to be used to identify the device - logical device name. This will be part of the pathname for I/O operations which operate on the device. This name will appear in the I/O system device table, which may be displayed using the **iosDevShow( )** routine.

(2) **CBIO_DEV_ID** - a pointer to the **CBIO_DEV** structure which provides interface to particular disk, via a disk cache, or a partition manager or a combination of a number of *CBIO* modules which are stacked on top of each other to form one of many configurations possible.

(3) A maximum number of files can be simultaneously opened on a particular device.

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

For example:

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

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

DEVICE AND PATH NAMES

On true MS-DOS machines, disk device names are typically of the form "A:", that is, a single letter designator followed by a colon. Such names may be used with the VxWorks dosFs file system. However, it is possible (and desirable) to use longer, more mnemonic device names, such as **DOS1:**, or **/floppy0**. The name is specified during the **dosFsDevCreate( )** call.

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

For the sake of consistency however use of forward slashes ("/") is recommended at all times.

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

USING EXTENDED DIRECTORY STRUCTURE

This library supports DOS4.0 standard file names which fit the restrictions of eight upper-case characters optionally followed by a three-character extension, as well as Windows style VFAT standard long file names that are stored mixed cased on disk, but are case insensitive when searched and matched (*e.g.*, during **open( )** call). The VFAT long file name is stored in a variable number of consecutive directory entries. Both standards restrict file size to 4 GB (32 bit value).

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

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

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

READING DIRECTORY ENTRIES

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

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

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

FILE DATE AND TIME

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

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

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

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

FILE ATTRIBUTES

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

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

DOS_ATTR_HIDDEN
this attribute is not used by VxWorks.

DOS_ATTR_SYSTEM
this attribute is not used by VxWorks.

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

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

DOS_ATTR_ARCHIVE
this attribute is not used by VxWorks.

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

```
struct stat fileStat;
fd = open ("file", O_RDONLY, 0);     /* open file          */
fstat (fd, &fileStat);               /* get file status    */
ioctl (fd, FIOATTRIBSET, (fileStat.st_attrib | DOS_ATTR_RDONLY));
                                     /* set read-only flag */
close (fd);                          /* close file         */
```

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

CONTIGOUS FILE SUPPORT

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

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

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

When contiguous space is allocated to a file, the file remains empty, while the newly allocated space has not been initialized. The data should be then written to the file, and eventually, when all data has been written, the file is closed. When file is closed, its space is truncated to reflect the amount of data actually written to the file. This file may then be again opened and used for further I/O operations **read( )** or **write( )**, but it can not be guaranteed that appended data will be contiguous to the initially written data segment.

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

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

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

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

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

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

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

CHANGING, UNMOUNTING, AND SYNCHRONIZING DISKS

Buffering of disk data in RAM, synchronization of these buffers with the disk and detection of removable disk replacement are all handled by the disk cache. See reference manual on **dcacheCbio** for more details.

If a disk is physically removed, the disk cache will cause **dosFsLib** to *unmount* the volume, which will mark all currently open file descriptors as obsolete.

If a new disk is inserted, it will be automatically *mounted* on the next call to **open( )** or **creat( )**.

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

```
long long *arg ;
```

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

FIODISKINIT
Re-initializes a DOS file system on the disk volume. This function calls **dosFsVolFormat( )** to format the volume, so **dosFsFmtLib** must be installed for this to work. Third argument of **ioctl( )** is passed as argument *opt* to **dosFsVolFormat( )** routine. This routine does not perform a low level format, the physical media is expected to be already formatted. If DOS file system device has not been created yet for a particular device, only direct call to **dosFsVolFormat( )** can be used.

```
fd = open ("DEV1:", O_WRONLY);
status = ioctl (fd, FIODISKINIT, DOS_OPT_BLANK);
```

FIODISKCHANGE
Announces a media change. No buffers flushing is performed. This function may be called from interrupt level:

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

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

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

FIOGETNAME
Gets the file name of the file descriptor and copies it to the buffer *nameBuf*. Note that *nameBuf* must be large enough to contain the largest possible path name, which requires at least 256 bytes.

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

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

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

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

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

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

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

FIOSEEK64
Sets the current byte offset in the file to the position specified by *newOffset*. This function supports offsets in 64-bit value range:

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

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

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

FIOWHERE64
Returns the current byte position in the file. This is the byte offset of the next byte to be read or written. This function returns a 64-bit value in *position*:

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

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

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

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

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

FIOTRUNC

Truncates the specified file's length to *newLength* bytes. Any disk clusters which had been allocated to the file but are now unused are deallocated, and the directory entry for the file is updated to reflect the new length. Only regular files may be truncated; attempts to use **FIOTRUNC** on directories will return an error. **FIOTRUNC** may only be used to make files shorter; attempting to specify a *newLength* larger than the current size of the file produces an error (setting **errno** to **S_dosFsLib_INVALID_NUMBER_OF_BYTES**).

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

FIOTRUNC64

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

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

FIONREAD

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

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

FIONREAD64

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

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

FIONFREE

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

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

FIONFREE64

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

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

FIOMKDIR

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

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

FIORMDIR

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

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

FIOLABELGET

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

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

FIOLABELSET

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

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

FIOATTRIBSET

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

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

FIOCONTIG

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

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

FIOCONTIG64

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

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

FIONCONTIG

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

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

FIONCONTIG64

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

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

FIOREADDIR

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

D

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

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

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

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

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

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

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

DOS_CHK_REPAIR (2)
Repair any errors found.

Possible verbosity levels are:

DOS_CHK_VERB_SILENT (0xff00)
Do not emit any messages, except errors encountered.

DOS_CHK_VERB_1 (0x0100)
Display some volume statistics when done testing, as well

DOS_CHK_VERB_2 (0x0200)
In addition to the above option, display path of every file, while it is being checked. This option may significantly slow down the test process.

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

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

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

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

INCLUDE FILES
dosFsLib.h

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

dpartCbio

NAME
dpartCbio – generic disk partition manager

ROUTINES
dpartDevCreate( ) - Initialize a partitioned disk
dpartPartGet( ) - retrieve handle for a partition

DESCRIPTION
This module implements a generic partition manager using the CBIO API (see **cbioLib**) It supports creating a separate file system device for each of its partitions.

This partition manager depends upon an external library to decode a particular disk partition table format, and report the resulting partition layout information back to this module. This module is responsible for maintaining the partition logic during operation.

When using this module with the **dcacheCbio** module, it is recommended this module be the master CBIO device. This module should be above the cache CBIO module layer. This is because the cache layer is optimized to function efficiently atop a single physical disk drive. One should call **dcacheDevCreate( )** before **dpartDevCreate( )**.

An implementation of the de-facto standard partition table format which is created by the MSDOS FDISK program is provided with the **usrFdiskPartLib** module, which should be used to handle PC-style partitioned hard or removable drives.

EXAMPLE

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

```
usrPartDiskFsInit( BLK_DEV * blkDevId )
  {
  const char * devNames[] = { "/sd0a", "/sd0b", "/sd0c", "/sd0d" };
  cbioCache;
  CBIO_DEV_ID cbioParts;
  /* create a disk cache atop the entire BLK_DEV */
  cbioCache = dcacheDevCreate ( blkDevId, NULL, 0, "/sd0" );
  if (NULL == cbioCache)
      {
      return (ERROR);
      }
  /* create a partition manager with a FDISK style decoder */
  cbioParts = dpartDevCreate( cbioCache, 4, usrFdiskPartRead );
  if (NULL == cbioParts)
      {
      return (ERROR);
      }
  /* create file systems atop each partition */
  dosFsDevCreate( devNames[0], dpartPartGet(cbioParts,0), 0x10, NONE);
  dosFsDevCreate( devNames[1], dpartPartGet(cbioParts,1), 0x10, NONE);
  dosFsDevCreate( devNames[2], dpartPartGet(cbioParts,2), 0x10, NONE);
  dosFsDevCreate( devNames[3], dpartPartGet(cbioParts,3), 0x10, NONE);
  }
```

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

SEE ALSO

dcacheLib, **dosFsLib**, **usrFdiskPartLib**

dspLib

NAME **dspLib** – dsp support library

ROUTINES **dspInit( )** - initialize dsp support

DESCRIPTION This library provides a general interface to the dsp. To activate dsp support, **dspInit( )** must be called before any tasks using the dsp are spawned. This is done automatically by the root task, **usrRoot( )**, in **usrConfig.c** when **INCLUDE_DSP** is defined in **configAll.h**. For information about architecture-dependent dsp routines, see the entry for **dspArchLib**.

VX_DSP_TASK OPTION

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

NOTE: If a task does any dsp operations, it must be spawned with **VX_DSP_TASK**.

INTERRUPT LEVEL DSP registers are not saved and restored for interrupt service routines connected with **intConnect( )**. However, if necessary, an interrupt service routine can save and restore dsp registers by calling routines in **dspArchLib**.

INCLUDE FILES **dspLib.h**

SEE ALSO **dspArchLib**, **dspShow**, **intConnect( )**, *VxWorks Programmer's Guide: Basic OS*

dspShow

NAME **dspShow** – dsp show routines

ROUTINES **dspShowInit( )** - initialize the dsp show facility
dspTaskRegsShow( ) - print the contents of a task's dsp registers

DESCRIPTION This library provides routines necessary to show a task's optional dsp context. This facility must first be installed using **dspShowInit( )**. It is included automatically when **INCLUDE_SHOW_ROUTINES** and **INCLUDE_DSP** are defined in **configAll.h**.

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

INCLUDE FILES **dspLib.h**

SEE ALSO **dspLib**

envLib

NAME

envLib – environment variable library

ROUTINES

envLibInit( ) - initialize environment variable facility
envPrivateCreate( ) - create a private environment
envPrivateDestroy( ) - destroy a private environment
putenv( ) - set an environment variable
getenv( ) - get an environment variable (ANSI)
envShow( ) - display the environment for a task

DESCRIPTION

This library provides a UNIX-compatible environment variable facility. Environment variables are created or modified with a call to **putenv( )**:

```
putenv ("variableName=value");
```

The value of a variable may be retrieved with a call to **getenv( )**, which returns a pointer to the value string.

Tasks may share a common set of environment variables, or they may optionally create their own private environments, either automatically when the task create hook is installed, or by an explicit call to **envPrivateCreate( )**. The task must be spawned with the **VX_PRIVATE_ENV** option set to receive a private set of environment variables. Private environments created by the task creation hook inherit the values of the environment of the task that called **taskSpawn( )** (since task create hooks run in the context of the calling task).

INCLUDE FILES

envLib.h

SEE ALSO

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

errnoLib

NAME

errnoLib – error status library

ROUTINES

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

DESCRIPTION

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

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

(1) For tasks, VxWorks maintains the **errno** value for each context separately, and saves and restores the value of **errno** with every context switch. The value of **errno** for a non-executing task is stored in the task's TCB. Thus, regardless of task context, code can always reference or modify **errno** directly.

(2) For interrupt service routines, VxWorks saves and restores **errno** on the interrupt stack as part of the interrupt enter and exit code provided automatically with the **intConnect( )** facility. Thus, interrupt service routines can also reference or modify **errno** directly.

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.

ERROR STATUS VALUES

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.

PRINTING ERROR STATUS VALUES

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

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

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

```
#define S_myFile_ERROR_TOO_MANY_COOKS       0x230003
```

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

On Unix:

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

On Windows:

```
makeStatTbl target/h
```

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

If the user now types from the VxWorks shell:

```
-> printErrno 0x230003
```

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

```
S_myFile_ERROR_TOO_MANY_COOKS
```

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

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

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

```
#define S_thisFile_MEANINGFUL_ERROR_MESSAGE   0xnnnn
```

where *thisFile* is the name of the module.

This facility is available to the user by adding header files with status lines of the appropriate forms and remaking VxWorks.

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( )**, **makeStatTbl**, *VxWorks Programmer's Guide: Basic OS*

etherMultiLib

NAME

etherMultiLib – a library to handle Ethernet multicast addresses

ROUTINES

etherMultiAdd( ) - add multicast address to a multicast address list
etherMultiDel( ) - delete an Ethernet multicast address record
etherMultiGet( ) - retrieve a table of multicast addresses from a driver

DESCRIPTION

This library manages a list of multicast addresses for network drivers. This abstracts the management of these drivers into a device-independent library.

To use this feature, include the following component:
INCLUDE_NETWRS_ETHERMULTILIB

INCLUDE FILES

string.h, **errno.h**, **netinet/in.h**, **net/if.h**, **lstLib.h**, **etherMultiLib.h**

eventLib

NAME

eventLib – VxWorks events library

ROUTINES

eventReceive( ) - wait for event(s)
eventSend( ) - send event(s)
eventClear( ) - clear all events for current task

DESCRIPTION

Events are a means of communication between tasks and interrupt routines, based on a synchronous model. Only tasks can receive events, and both tasks and ISRs can send them.

Events are similar to signals in that they are directed at one task but differ in the fact that they are synchronous in nature. Thus, the receiving task must pend when waiting for events to occur. Also, unlike signals, a handler is not needed since, when wanted events are received, the pending task continues its execution (like after a call to **msgQReceive( )** or **semTake( )**).

Each task has its own events field that can be filled by having tasks (even itself) and/or ISRs sending events to the task. 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.

INCLUDE FILES **eventLib.h**

SEE ALSO **taskLib**, **semLib**, **semBLib**, **semCLib**, **semMLib**, **msgQLib**, *VxWorks Programmer's Guide: Basic OS*

excArchLib

NAME **excArchLib** – architecture-specific exception-handling facilities

ROUTINES

excVecInit( ) - initialize the exception/interrupt vectors
excConnect( ) - connect a C routine to an exception vector (PowerPC)
excIntConnect( ) - connect a C routine to an asynchronous exception vector (PowerPC, ARM)
excCrtConnect( ) - connect a C routine to a critical exception vector (PowerPC 403)
excIntCrtConnect( ) - connect a C routine to a critical interrupt vector (PowerPC 403)
excVecSet( ) - set a CPU exception vector (PowerPC, ARM)
excVecGet( ) - get a CPU exception vector (PowerPC, ARM)

DESCRIPTION This library contains exception-handling facilities that are architecture dependent. For information about generic (architecture-independent) exception-handling, see the manual entry for **excLib**.

INCLUDE FILES **excLib.h**

SEE ALSO **excLib**, **dbgLib**, **sigLib**, **intLib**

excLib

NAME

excLib – generic exception handling facilities

ROUTINES

excInit( ) - initialize the exception handling package
excHookAdd( ) - specify a routine to be called with exceptions
excTask( ) - handle task-level exceptions

DESCRIPTION

This library provides generic initialization facilities for handling exceptions. It safely traps and reports exceptions caused by program errors in VxWorks tasks, and it reports occurrences of interrupts that are explicitly connected to other handlers. For information about architecture-dependent exception handling facilities, see the manual entry for **excArchLib**.

INITIALIZATION

Initialization of **excLib** facilities occurs in two steps. First, the routine **excVecInit( )** is called to set all vectors to the default handlers for an architecture provided by the corresponding architecture exception handling library. Since this does not involve VxWorks' kernel facilities, it is usually done early in the system start-up routine **usrInit( )** in the library **usrConfig.c** with interrupts disabled.

The rest of this package is initialized by calling **excInit( )**, which spawns the exception support task, **excTask( )**, and creates the message queues used to communicate with it.

Exceptions or uninitialized interrupts that occur after the vectors have been initialized by **excVecInit( )**, but before **excInit( )** is called, cause a trap to the ROM monitor.

NORMAL EXCEPTION HANDLING

When a program error generates an exception (such as divide by zero, or a bus or address error), the task that was executing when the error occurred is suspended, and a description of the exception is displayed on standard output. The VxWorks kernel and other system tasks continue uninterrupted. The suspended task can be examined with the usual VxWorks routines, including **ti( )** for task information and **tt( )** for a stack trace. It may be possible to fix the task and resume execution with **tr( )**. However, tasks aborted in this way are often unsalvageable and can be deleted with **td( )**.

When an interrupt that is not connected to a handler occurs, the default handler provided by the architecture-specific module displays a description of the interrupt on standard output.

ADDITIONAL EXCEPTION HANDLING ROUTINE

The **excHookAdd( )** routine adds a routine that will be called when a hardware exception occurs. This routine is called at the end of normal exception handling.

TASK-LEVEL SUPPORT

The **excInit( )** routine spawns **excTask( )**, which performs special exception handling functions that need to be done at task level. Do not suspend, delete, or change the priority of this task.

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

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

INCLUDE FILES **excLib.h**

SEE ALSO **dbgLib**, **sigLib**, **intLib**

fioLib

NAME

fioLib – formatted I/O library

ROUTINES

fioLibInit( ) - initialize the formatted I/O support library
printf( ) - write a formatted string to the standard output stream (ANSI)
printErr( ) - write a formatted string to the standard error stream
fdprintf( ) - write a formatted string to a file descriptor
sprintf( ) - write a formatted string to a buffer (ANSI)
vprintf( ) - write a string formatted with a variable argument list to standard output (ANSI)
vfdprintf( ) - write a string formatted with a variable argument list to a file descriptor
vsprintf( ) - write a string formatted with a variable argument list to a buffer (ANSI)
fioFormatV( ) - convert a format string
fioRead( ) - read a buffer
fioRdString( ) - read a string from a file
sscanf( ) - read and convert characters from an ASCII string (ANSI)

DESCRIPTION

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

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

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

INCLUDE FILES

fioLib.h, **stdio.h**

SEE ALSO

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

floatLib

NAME **floatLib** – floating-point formatting and scanning library

ROUTINES **floatInit( )** - initialize floating-point I/O support

DESCRIPTION This library provides the floating-point I/O formatting and scanning support routines.

The floating-point formatting and scanning support routines are not directly callable; they are connected to call-outs in the **printf( )**/**scanf( )** family of functions in **fioLib**. This is done dynamically by the routine **floatInit( )**, which is called by the root task, **usrRoot( )**, in **usrConfig.c** when the configuration macro **INCLUDE_FLOATING_POINT** is defined. If this option is omitted (*i.e.*, **floatInit( )** is not called), floating-point format specifications in **printf( )** and **sscanf( )** are not supported.

INCLUDE FILES **math.h**

SEE ALSO **fioLib**

fppArchLib

NAME **fppArchLib** – architecture-dependent floating-point coprocessor support

ROUTINES
fppSave( ) - save the floating-point coprocessor context
fppRestore( ) - restore the floating-point coprocessor context
fppProbe( ) - probe for the presence of a floating-point coprocessor
fppTaskRegsGet( ) - get the floating-point registers from a task TCB
fppTaskRegsSet( ) - set the floating-point registers of a task

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

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

INITIALIZATION

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

x86 ARCHITECTURE

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

x86 INITIALIZATION

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

x86 VX_FP_TASK OPTION

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

NOTE: If a task does any floating-point operations, MMX operations, and streaming SIMD operation, it must be spawned with **VX_FP_TASK**. It is deadly to execute any floating-point operations in a task spawned without **VX_FP_TASK** option, and very difficult to find. To detect that illegal/unintentional/accidental floating-point operations, a new API and mechanism is added. The mechanism is to enable or disable the FPU by toggling the TS flag in the CR0 in the new task switch hook routine - **fppArchSwitchHook( )** - respecting the **VX_FP_TASK** option. If **VX_FP_TASK** option is not set in the switching-in task, the FPU is disabled. Thus the device-not-available exception will be raised if that task does any floating-point operations. This mechanism is disabled in the default. To enable, call the enabler - **fppArchSwitchHookEnable( )** - with a parameter **TRUE**(1). A parameter **FALSE**(0) disables the mechanism.

x86 MIXING MMX AND FPU INSTRUCTIONS

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

– Keep the code in separate modules, procedures, or routines.

– Do not rely on register contents across transitions between FPU and MMX code modules.

– When transitioning between MMX code and FPU code, save the MMX register state (if it will be needed in the future) and execute an EMMS instruction to empty the MMX state.

– When transitioning between FPU and MMX code, save the FPU state, if it will be needed in the future.

x86 MIXING SSE/SSE2 AND FPU/MMX INSTRUCTIONS

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

– Those SSE and SSE2 instruction that operate only on the XMM registers (such as the packed and scalar floating-point instructions and the 128-bit SIMD integer instructions) can be executed in the same instruction stream with 64-bit SIMD integer or FPU instructions without any restrictions. For example, an application can perform the majority of its floating-point computations in the XMM registers, using the packed and scalar floating-point instructions, and at the same time use the FPU to perform trigonometric and other transcendental computations. Likewise, an application can perform packed 64-bit and 128-bit SIMD integer operations can be executed together without restrictions.

– Those SSE and SSE2 instructions that operate on MMX registers (such as the CVTPS2PI, CVTTPS2PI, CVTPI2PS, CVTPD2PI, CVTTPD2PI, CVTPI2PD, MOVDQ2Q, MOVQ2DQ, PADDQ, and PSUBQ instructions) can also be executed in the same instruction stream as 64-bit SIMD integer or FPU instructions, however, here they subject to the restrictions on the simultaneous use of MMX and FPU instructions, which mentioned in the previous paragraph.

x86 INTERRUPT LEVEL

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

x86 EXCEPTIONS

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

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

ARM ARCHITECTURE

This architecture does not currently support floating-point coprocessors.

INCLUDE FILES **fppLib.h**

SEE ALSO **fppLib**, **intConnect( )**, *Motorola MC68881/882 Floating-Point Coprocessor User's Manual*, *Intel 80960SA/SB Reference Manual*, *Intel 80960KB Programmer's Reference Manual*, *Intel 387 DX User's Manual*, *Intel Architecture Software Developer's Manual*, *Hitachi SH7750 Hardware Manual*, Gerry Kane and Joe Heinrich: *MIPS RISC Architecture Manual*

fppLib

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

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

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

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

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

VX_FP_TASK OPTION

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

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

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

INCLUDE FILES **fppLib.h**

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

fppShow

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

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

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

If you use the configuration header files, define **INCLUDE_SHOW_ROUTINES** in **config.h**.

If you use the Tornado project facility, select **INCLUDE_HW_FP_SHOW**.

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

INCLUDE FILES **fppLib.h**

SEE ALSO **fppLib**

ftpdLib

NAME **ftpdLib** – File Transfer Protocol (FTP) server

ROUTINES
ftpdInit( ) - initialize the FTP server task
ftpdDelete( ) - terminate the FTP server task

DESCRIPTION This library implements the server side of the File Transfer Protocol (FTP), which provides remote access to the file systems available on a target. The protocol is defined in RFC 959.

This implementation supports all commands required by that specification, as well as several additional commands.

USER INTERFACE

During system startup, the **ftpdInit( )** routine creates a control connection at the predefined FTP server port which is monitored by the primary FTP task. Each FTP session established is handled by a secondary server task created as necessary. The server accepts the following commands:

HELP	- List supported commands.
USER	- Verify user name.
PASS	- Verify password for the user.
QUIT	- Quit the session.
LIST	- List out contents of a directory.
NLST	- List directory contents using a concise format.
RETR	- Retrieve a file.
STOR	- Store a file.
CWD	- Change working directory.
TYPE	- Change the data representation type.
PORT	- Change the port number.
PWD	- Get the name of current working directory.
STRU	- Change file structure settings.
MODE	- Change file transfer mode.
ALLO	- Reserve sufficient storage.
ACCT	- Identify the user's account.
PASV	- Make the server listen on a port for data connection.
NOOP	- Do nothing.
DELE	- Delete a file

The **ftpdDelete( )** routine will disable the FTP server until restarted. It reclaims all system resources used by the server tasks and cleanly terminates all active sessions.

To use this feature, include the following component: **INCLUDE_FTP_SERVER**

INCLUDE FILES

ftpdLib.h

SEE ALSO

ftpLib, **netDrv**, *RFC-959 File Transfer Protocol*

F

ftpLib

NAME **ftpLib** – File Transfer Protocol (FTP) library

ROUTINES
ftpCommand( ) - send an FTP command and get the reply
ftpCommandEnhanced( ) - send an FTP command and get the complete RFC reply code
ftpXfer( ) - initiate a transfer via FTP
ftpReplyGet( ) - get an FTP command reply
ftpReplyGetEnhanced( ) - get an FTP command reply
ftpHookup( ) - get a control connection to the FTP server on a specified host
ftpLogin( ) - log in to a remote FTP server
ftpDataConnInitPassiveMode( ) - initialize an FTP data connection using PASV mode
ftpDataConnInit( ) - initialize an FTP data connection using PORT mode
ftpDataConnGet( ) - get a completed FTP data connection
ftpLs( ) - list directory contents via FTP
ftpLibDebugOptionSet( ) - set the debug level of the ftp library routines
ftpTransientConfigSet( ) - set parameters for host **FTP_TRANSIENT** responses
ftpTransientConfigGet( ) - get parameters for host **FTP_TRANSIENT** responses
ftpTransientFatalInstall( ) - set applette to stop FTP transient host responses

DESCRIPTION This library provides facilities for transferring files to and from a host via File Transfer Protocol (FTP). This library implements only the “client” side of the FTP facilities.

FTP IN VXWORKS For most purposes, you should access the services of **ftpLib** by means of **netDrv**, a VxWorks I/O driver that supports transparent access to remote files by means of standard I/O system calls. Before attempting to access **ftpLib** services directly, you should check whether **netDrv** already provides the same access for less trouble.

HIGH-LEVEL INTERFACE

The routines **ftpXfer( )** and **ftpReplyGet( )** provide the highest level of direct interface to FTP. The routine **ftpXfer( )** connects to a specified remote FTP server, logs in under a specified user name, and initiates a specified data transfer command. The routine **ftpReplyGet( )** receives control reply messages sent by the remote FTP server in response to the commands sent.

LOW-LEVEL INTERFACE

The routines **ftpHookup( )**, **ftpLogin( )**, **ftpDataConnInit( )**, **ftpDataConnGet( )**, **ftpCommand( )**, **ftpCommandEnhanced( )** provide the primitives necessary to create and use control and data connections to remote FTP servers. The following example shows how to use these low-level routines. It implements roughly the same function as **ftpXfer( )**.

```
char *host, *user, *passwd, *acct, *dirname, *filename;
int ctrlSock = ERROR; /* This is the control socket file descriptor */
```

```
int dataSock = ERROR; /* This is the data path socket file descriptor */
if (((ctrlSock = ftpHookup (host)) == ERROR) ||
    (ftpLogin (ctrlSock, user, passwd, acct) == ERROR) ||
    (ftpCommand (ctrlSock, "TYPE I", 0, 0, 0, 0, 0, 0) != FTP_COMPLETE) ||
    (ftpCommand (ctrlSock, "CWD %s", dirname, 0,0,0,0,0) != FTP_COMPLETE) ||
    ((dataSock = ftpDataConnInit (ctrlSock)) == ERROR) ||
    (ftpCommand (ctrlSock, "RETR %s", filename, 0,0,0,0,0) != FTP_PRELIM) ||
    ((dataSock = ftpDataConnGet (dataSock)) == ERROR))
    {
    /* an error occurred; close any open sockets and return */
    if (ctrlSock != ERROR)
        close (ctrlSock);
    if (dataSock != ERROR)
        close (dataSock);
    return (ERROR);
    }
```

For even lower-level access, please note that the sockets provided by **ftpHookup( )** and **ftpDataConnInit( )** are standard TCP/IP sockets. Developers may implement **read( )**, **write( )** and **select( )** calls using these sockets for maximum flexibility.

To use this feature, include the following component: **INCLUDE_FTP**

TUNING FOR MULTIPLE FILE ACCESS

Please note that accessing multiple files simultaneously may require increasing the memory available to the network stack. You can examine memory requirements by using **netStackSysPoolShow( )** and **netStackDataPoolShow( )** before opening and after closing files.

You may need to modify the following macro definitions according to your specific memory requirements:

NUM_64
NUM_128
NUM_256
NUM_512
NUM_1024
NUM_2048
NUM_SYS_64
NUM_SYS_128
NUM_SYS_256
NUM_SYS_512
NUM_SYS_1024
NUM_SYS_2048

Please also note that each concurrent file access requires three file descriptors (File, Control and Socket). The following macro definition may need modification per your application: **NUM_FILES**

Developers are encouraged to enable the error reporting facility during debugging using the function **ftpLibDebugOptionsSet( )**. The output is displayed via the logging facility.

INCLUDE FILES **ftpLib.h**

SEE ALSO **netDrv**, **logLib**

ftruncate

NAME **ftruncate** – POSIX file truncation

ROUTINES **ftruncate( )** - truncate a file (POSIX)

hostLib

NAME **hostLib** – host table subroutine library

ROUTINES

hostTblInit( ) - initialize the network host table
hostAdd( ) - add a host to the host table
hostDelete( ) - delete a host from the host table
hostGetByName( ) - look up a host in the host table by its name
hostGetByAddr( ) - look up a host in the host table by its Internet address
sethostname( ) - set the symbolic name of this machine
gethostname( ) - get the symbolic name of this machine

DESCRIPTION

This library provides routines to store and access the network host database. The host table contains information regarding the known hosts on the local network. The host table (displayed with **hostShow( )**) contains the Internet address, the official host name, and aliases.

By convention, network addresses are specified in dotted (".") decimal notation. The library **inetLib** contains Internet address manipulation routines. Host names and aliases may contain any printable character.

Before any of the routines in this module can be used, the library must be initialized by **hostTblInit( )**. This is done automatically if **INCLUDE_HOST_TBL** is defined.

INCLUDE FILES **hostLib.h**

SEE ALSO **inetLib**

icmpShow

NAME **icmpShow** – ICMP Information display routines

ROUTINES
icmpShowInit( ) - initialize ICMP show routines
icmpstatShow( ) - display statistics for ICMP

DESCRIPTION This library provides routines to show ICMP related statistics.

Interpreting these statistics requires detailed knowledge of Internet network protocols. Information on these protocols can be found in the following books:

TCP/IP Illustrated Volume II, The Implementation, by Richard Stevens

The Design and Implementation of the 4.4 BSD UNIX Operating System, by Leffler, McKusick, Karels and Quarterman

The **icmpShowInit( )** routine links the ICMP show facility into the VxWorks system. This is performed automatically if **INCLUDE_NET_SHOW** is defined.

SEE ALSO **netLib**, **netShow**

ifIndexLib

NAME **ifIndexLib** – interface index library

ROUTINES
ifIndexLibInit( ) - initializes library variables
ifIndexLibShutdown( ) - frees library variables
ifIndexAlloc( ) - return a unique interface index
ifIndexTest( ) - returns true if an index has been allocated.

ifLib

NAME

ifLib – network interface library

ROUTINES

ifUnnumberedSet( ) - configure an interface to be unnumbered
ifAddrAdd( ) - add an interface address for a network interface
ifAddrSet( ) - set an interface address for a network interface
ifAddrDelete( ) - delete an interface address for a network interface
ifAddrGet( ) - get the Internet address of a network interface
ifBroadcastSet( ) - set the broadcast address for a network interface
ifBroadcastGet( ) - get the broadcast address for a network interface
ifDstAddrSet( ) - define an address for the other end of a point-to-point link
ifDstAddrGet( ) - get the Internet address of a point-to-point peer
ifMaskSet( ) - define a subnet for a network interface
ifMaskGet( ) - get the subnet mask for a network interface
ifFlagChange( ) - change the network interface flags
ifFlagSet( ) - specify the flags for a network interface
ifFlagGet( ) - get the network interface flags
ifMetricSet( ) - specify a network interface hop count
ifMetricGet( ) - get the metric for a network interface
ifRouteDelete( ) - delete routes associated with a network interface
ifAllRoutesDelete( ) - delete all routes associated with a network interface
ifunit( ) - map an interface name to an interface structure pointer
ifNameToIfIndex( ) - returns the interface index given the interface name
ifIndexToIfName( ) - returns the interface name given the interface index

DESCRIPTION

This library contains routines to configure the network interface parameters. Generally, each routine corresponds to one of the functions of the UNIX command **ifconfig**.

To use this feature, include the following component: **INCLUDE_NETWRS_IFLIB**

INCLUDE FILES

ifLib.h

SEE ALSO

hostLib

igmpShow

NAME **igmpShow** – IGMP information display routines

ROUTINES **igmpShowInit( )** - initialize IGMP show routines
igmpstatShow( ) - display statistics for IGMP

DESCRIPTION This library provides routines to show IGMP related statistics.

Interpreting these statistics requires detailed knowledge of Internet network protocols. Information on these protocols can be found in the following books:

TCP/IP Illustrated Volume II, The Implementation, by Richard Stevens

The Design and Implementation of the 4.4 BSD UNIX Operating System, by Leffler, McKusick, Karels and Quarterman

The **igmpShowInit( )** routine links the IGMP show facility into the VxWorks system. This is performed automatically if **INCLUDE_NET_SHOW** and **INCLUDE_IGMP** are defined.

SEE ALSO **netLib**, **netShow**

inetLib

NAME **inetLib** – internet address manipulation routines

ROUTINES **inet_addr( )** - convert a dot notation Internet address to a long integer
inet_lnaof( ) - get the local address (host number) from the Internet address
inet_makeaddr_b( ) - form an Internet address from network and host numbers
inet_makeaddr( ) - form an Internet address from network and host numbers
inet_netof( ) - return the network number from an Internet address
inet_netof_string( ) - extract the network address in dot notation
inet_network( ) - convert an Internet network number from string to address
inet_ntoa_b( ) - convert an network address to dot notation, store it in a buffer
inet_ntoa( ) - convert a network address to dotted decimal notation
inet_aton( ) - convert a network address from dot notation, store in a structure

DESCRIPTION This library provides routines for manipulating Internet addresses, including the UNIX BSD 4.3 **inet_** routines. It includes routines for converting between character addresses in Internet standard dotted decimal notation and integer addresses, routines for extracting the network and host portions out of an Internet address, and routines for constructing Internet addresses given the network and host address parts.

All Internet addresses are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine format integer values.

INTERNET ADDRESSES

Internet addresses are typically specified in dotted decimal notation or as a 4-byte number. Values specified using the dotted decimal notation take one of the following forms:

```
a.b.c.d
a.b.c
a.b
a
```

If four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address. Note that when an Internet address is viewed as a 32-bit integer quantity on any MC68000 family machine, the bytes referred to above appear as "a.b.c.d" and are ordered from left to right.

If a three-part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right-most two bytes of the network address. This makes the three-part address format convenient for specifying Class B network addresses as "128.net.host".

If a two-part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right-most three bytes of the network address. This makes the two-part address format convenient for specifying Class A network addresses as "net.host".

If only one part is given, the value is stored directly in the network address without any byte rearrangement.

Although dotted decimal notation is the default, it is possible to use the dot notation with hexadecimal or octal numbers. The base is indicated using the same prefixes as are used in C. That is, a leading 0x or 0X indicates a hexadecimal number. A leading 0 indicates an octal number. If there is no prefix, the number is interpreted as decimal.

To use this feature, include the following component: **INCLUDE_NETWRS_INETLIB**

INCLUDE FILES **inetLib.h**, **inet.h**

SEE ALSO UNIX BSD 4.3 manual entry for inet(3N)

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** (compressed data is generated by **deflate( )** and **binToAsm**). The boot ROM startup code (in **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/**.

intArchLib

NAME **intArchLib** – architecture-dependent interrupt library

ROUTINES

intLevelSet( ) - set the interrupt level (68K, x86, ARM, SimSolaris, SimNT and SH)
intLock( ) - lock out interrupts
intUnlock( ) - cancel interrupt locks
intEnable( ) - enable corresponding interrupt bits (MIPS, PowerPC, ARM)
intDisable( ) - disable corresponding interrupt bits (MIPS, PowerPC, ARM)
intCRGet( ) - read the contents of the cause register (MIPS)
intCRSet( ) - write the contents of the cause register (MIPS)
intSRGet( ) - read the contents of the status register (MIPS)
intSRSet( ) - update the contents of the status register (MIPS)
intConnect( ) - connect a C routine to a hardware interrupt
intHandlerCreate( ) - construct interrupt handler for C routine (68K, x86, MIPS, SimSolaris)
intLockLevelSet( ) - set current interrupt lock-out level (68K, x86, ARM, SH, SimSolaris, SimNT)
intLockLevelGet( ) - get current interrupt lock-out level (68K, x86, ARM, SH, SimSolaris, SimNT)
intVecBaseSet( ) - set vector (trap) base address (68K, x86, MIPS, ARM, SimSolaris, SimNT)
intVecBaseGet( ) - get vector (trap) base address (68K, x86, MIPS, ARM, SimSolaris, SimNT)
intVecSet( ) - set a CPU vector (trap) (68K, x86, MIPS, SH, SimSolaris, SimNT)

intVecGet( ) - get an interrupt vector (68K, x86, MIPS, SH, SimSolaris, SimNT)
intVecTableWriteProtect( ) - write-protect exception vector table (68K, x86, ARM, SimSolaris, SimNT)
intUninitVecSet( ) - set the uninitialized vector handler (ARM)
intHandlerCreateI86( ) - construct an interrupt handler for a C routine (x86)
intVecSet2( ) - set a CPU vector, gate type(int/trap), and selector (x86)
intVecGet2( ) - get a CPU vector, gate type(int/trap), and gate selector (x86)
intStackEnable( ) - enable or disable the interrupt stack usage (x86)

DESCRIPTION

This library provides architecture-dependent routines to manipulate and connect to hardware interrupts. Any C language routine can be connected to any interrupt by calling **intConnect( )**. Vectors can be accessed directly by **intVecSet( )** and **intVecGet( )**. The vector (trap) base register (if present) can be accessed by the routines **intVecBaseSet( )** and **intVecBaseGet( )**.

Tasks can lock and unlock interrupts by calling **intLock( )** and **intUnlock( )**. The lock-out level can be set and reported by **intLockLevelSet( )** and **intLockLevelGet( )** (68K, x86, ARM and SH only). The routine **intLevelSet( )** changes the current interrupt level of the processor (68K, ARM, SimSolaris, and SH).

WARNING: Do not call VxWorks system routines with interrupts locked. Violating this rule may re-enable interrupts unpredictably.

INTERRUPT VECTORS AND NUMBERS

Most of the routines in this library take an interrupt vector as a parameter, which is generally the byte offset into the vector table. Macros are provided to convert between interrupt vectors and interrupt numbers:

IVEC_TO_INUM (intVector)
converts a vector to a number.

INUM_TO_IVEC (intNumber)
converts a number to a vector.

TRAPNUM_TO_IVEC (trapNumber)
converts a trap number to a vector.

EXAMPLE

To switch between one of several routines for a particular interrupt, the following code fragment is one alternative:

```
vector  = INUM_TO_IVEC(some_int_vec_num);
oldfunc = intVecGet (vector);
newfunc = intHandlerCreate (routine, parameter);
intVecSet (vector, newfunc);
...
intVecSet (vector, oldfunc);    /* use original routine */
...
intVecSet (vector, newfunc);    /* reconnect new routine */
```

INCLUDE FILES **iv.h**, **intLib.h**

SEE ALSO **intLib**

intLib

NAME **intLib** – architecture-independent interrupt subroutine library

ROUTINES
intContext( ) - determine if the current state is in interrupt or task context
intCount( ) - get the current interrupt nesting depth

DESCRIPTION This library provides generic routines for interrupts. Any C language routine can be connected to any interrupt (trap) by calling **intConnect( )**, which resides in **intArchLib**. The **intCount( )** and **intContext( )** routines are used to determine whether the CPU is running in an interrupt context or in a normal task context. For information about architecture-dependent interrupt handling, see the manual entry for **intArchLib**.

INCLUDE FILES **intLib.h**

SEE ALSO **intArchLib**, *VxWorks Programmer's Guide: Basic OS*

ioLib

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

ROUTINES
creat( ) - create a file
open( ) - open a file
unlink( ) - delete a file (POSIX)
remove( ) - remove a file (ANSI)
close( ) - close a file
rename( ) - change the name of a file
read( ) - read bytes from a file or device
write( ) - write bytes to a file
ioctl( ) - perform an I/O control function
lseek( ) - set a file read/write pointer
ioDefPathSet( ) - set the current default path
ioDefPathGet( ) - get the current default path
chdir( ) - set the current default path
getcwd( ) - get the current default path (POSIX)

getwd( ) - get the current default path
ioGlobalStdSet( ) - set the file descriptor for global standard input/output/error
ioGlobalStdGet( ) - get the file descriptor for global standard input/output/error
ioTaskStdSet( ) - set the file descriptor for task standard input/output/error
ioTaskStdGet( ) - get the file descriptor for task standard input/output/error
isatty( ) - return whether the underlying driver is a *tty* device

DESCRIPTION

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

Interfaces to the seven basic driver-provided functions: **creat( )**, **remove( )**, **open( )**, **close( )**, **read( )**, **write( )**, and **ioctl( )**.

Interfaces to several file system functions, including **rename( )** and **lseek( )**.

Routines to set and get the current working directory.

Routines to assign task and global standard file descriptors.

FILE DESCRIPTORS

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

Three file descriptors are reserved and have special meanings:

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

VxWorks allows two levels of redirection. First, there is a global assignment of the three standard file descriptors. By default, new tasks use this global assignment. The global assignment of the three standard file descriptors is controlled by the routines **ioGlobalStdSet( )** and **ioGlobalStdGet( )**.

Second, individual tasks may override the global assignment of these file descriptors with their own assignments that apply only to that task. The assignment of task-specific standard file descriptors is controlled by the routines **ioTaskStdSet( )** and **ioTaskStdGet( )**.

INCLUDE FILES

ioLib.h

SEE ALSO

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

iosLib

NAME **iosLib** – I/O system library

ROUTINES
iosInit( ) - initialize the I/O system
iosDrvInstall( ) - install an I/O driver
iosDrvRemove( ) - remove an I/O driver
iosDevAdd( ) - add a device to the I/O system
iosDevDelete( ) - delete a device from the I/O system
iosDevFind( ) - find an I/O device in the device list
iosFdValue( ) - validate an open file descriptor and return the driver-specific value

DESCRIPTION This library is the driver-level interface to the I/O system. Its primary purpose is to route user I/O requests to the proper drivers, using the proper parameters. To do this, **iosLib** keeps tables describing the available drivers (*e.g.*, names, open files).

The I/O system should be initialized by calling **iosInit( )**, before calling any other routines in **iosLib**. Each driver then installs itself by calling **iosDrvInstall( )**. The devices serviced by each driver are added to the I/O system with **iosDevAdd( )**.

The I/O system is described more fully in the *I/O System* chapter of the *Programmer's Guide*.

INCLUDE FILES **iosLib.h**

SEE ALSO **intLib**, **ioLib**, *VxWorks Programmer's Guide: I/O System*

iosShow

NAME **iosShow** – I/O system show routines

ROUTINES
iosShowInit( ) - initialize the I/O system show facility
iosDrvShow( ) - display a list of system drivers
iosDevShow( ) - display the list of devices in the system
iosFdShow( ) - display a list of file descriptor names in the system

DESCRIPTION This library contains I/O system information display routines.

The routine **iosShowInit( )** links the I/O system information show facility into the VxWorks system. It is called automatically when **INCLUDE_SHOW_ROUTINES** is defined in **configAll.h**.

SEE ALSO **intLib**, **ioLib**, *VxWorks Programmer's Guide: I/O System*, **windsh**, *Tornado User's Guide: Shell*

ipFilterLib

NAME **ipFilterLib** – IP filter hooks library

ROUTINES
ipFilterLibInit( ) - initialize IP filter facility
ipFilterHookAdd( ) - add a routine to receive all internet protocol packets
ipFilterHookDelete( ) - delete a IP filter hook routine

DESCRIPTION This library provides utilities that give direct access to IP packets. You can examine or process incoming raw IP packets using the hooks you installed with **ipFilterHookAdd( )**. Using a filter hook, you can build IP traffic monitoring and testing tools.

However, you should not use an IP filter hook as a standard means to provide network access to an application. The filter hook lets you see, process, and even consume packets before their intended recipients have seen the packets. For most network applications, this is too much responsibility. Thus, most network applications should access the network access through the higher-level socket interface provided by **sockLib**.

The **ipFilterLibInit( )** routine links the IP filtering facility into the VxWorks system. This is performed automatically if **INCLUDE_IP_FILTER** is defined.

VXWORKS AE PROTECTION DOMAINS
Under VxWorks AE, you can call **ipFilterHookAdd( )** from within the kernel protection domain only, and the function referenced in the *ipFilterHook* parameter must reside in the kernel protection domain. This restriction does not apply to non-AE versions of VxWorks.

ipProto

NAME **ipProto** – an interface between the BSD IP protocol and the MUX

ROUTINES
ipAttach( ) - a generic attach routine for the TCP/IP network stack
ipDetach( ) - a generic detach routine for the TCP/IP network stack

DESCRIPTION This library provides an interface between the Berkeley protocol stack and the MUX interface for both NPT and END devices. The **ipAttach( )** routine binds the IP protocol to a specific device. It is called automatically during network initialization if **INCLUDE_END** is defined. The **ipDetach( )** routine removes an existing binding to an END device.

NOTE: The library can only transmit data to link-level destination addresses less than or equal to 64 bytes in length.

INCLUDE FILES **end.h**, **muxLib.h**, **etherMultiLib.h**, **sys/ioctl.h**

kernelLib

NAME

kernelLib – VxWorks kernel library

ROUTINES

kernelInit( ) - initialize the kernel
kernelVersion( ) - return the kernel revision string
kernelTimeSlice( ) - enable round-robin selection

DESCRIPTION

The VxWorks kernel provides tasking control services to an application. The libraries **kernelLib**, **taskLib**, **semLib**, **tickLib**, and **wdLib** comprise the kernel functionality. This library is the interface to the VxWorks kernel initialization, revision information, and scheduling control.

KERNEL INITIALIZATION

The kernel must be initialized before any other kernel operation is performed. Normally kernel initialization is taken care of by the system configuration code in **usrInit( )** in **usrConfig.c**.

Kernel initialization consists of the following:

(1) Defining the starting address and size of the system memory partition. The **malloc( )** routine uses this partition to satisfy memory allocation requests of other facilities in VxWorks.

(2) Allocating the specified memory size for an interrupt stack. Interrupt service routines will use this stack unless the underlying architecture does not support a separate interrupt stack, in which case the service routine will use the stack of the interrupted task.

(3) Specifying the interrupt lock-out level. VxWorks will not exceed the specified level during any operation. The lock-out level is normally defined to mask the highest priority possible. However, in situations where extremely low interrupt latency is required, the lock-out level may be set to ensure timely response to the interrupt in question. Interrupt service routines handling interrupts of priority greater than the interrupt lock-out level may not call any VxWorks routine.

Once the kernel initialization is complete, a root task is spawned with the specified entry point and stack size. The root entry point is normally **usrRoot( )** of the **usrConfig.c** module. The remaining VxWorks initialization takes place in **usrRoot( )**.

ROUND-ROBIN SCHEDULING

Round-robin scheduling allows the processor to be shared fairly by all tasks of the same priority. Without round-robin scheduling, when multiple tasks of equal priority must share the processor, a single non-blocking task can usurp the processor until preempted by a task of higher priority, thus never giving the other equal-priority tasks a chance to run.

Round-robin scheduling is disabled by default. It can be enabled or disabled with the routine **kernelTimeSlice( )**, which takes a parameter for the "time slice" (or interval) that each task will be allowed to run before relinquishing the processor to another equal-priority task. If the parameter is zero, round-robin scheduling is turned off. If round-robin scheduling is enabled and preemption is enabled for the executing task, the system tick handler will increment the task's time-slice count. When the specified time-slice interval is completed, the system tick handler clears the counter and the task is placed at the tail of the list of tasks at its priority. New tasks joining a given priority group are placed at the tail of the group with a run-time counter initialized to zero.

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

If a task blocks or is preempted by a higher priority task during its interval, it's time-slice count is saved and then restored when the task is eligible for execution. In the case of preemption, the task will resume execution once the higher priority task completes, assuming no other task of a higher priority is ready to run. For the case when the task blocks, it is placed at the tail of the list of tasks at its priority. If preemption is disabled during round-robin scheduling, the time-slice count of the executing task is not incremented.

Time-slice counts are accrued against the task that is executing when a system tick occurs regardless of whether the task has executed for the entire tick interval. Due to preemption by higher priority tasks or ISRs stealing CPU time from the task, scenarios exist where a task can execute for less or more total CPU time than it's allotted time slice.

INCLUDE FILES **kernelLib.h**

SEE ALSO **taskLib**, **intLib**, *VxWorks Programmer's Guide: Basic OS*

ledLib

NAME **ledLib** – line-editing library

ROUTINES
ledOpen( ) - create a new line-editor ID
ledClose( ) - discard the line-editor ID
ledRead( ) - read a line with line-editing
ledControl( ) - change the line-editor ID parameters

DESCRIPTION

This library provides a line-editing layer on top of a *tty* device. The shell uses this interface for its history-editing features.

The shell history mechanism is similar to the UNIX Korn shell history facility, with a built-in line-editor similar to UNIX **vi** that allows previously typed commands to be edited. The command **h( )** displays the 20 most recent commands typed into the shell; old commands fall off the top as new ones are entered.

To edit a command, type **ESC** to enter edit mode, and use the commands listed below. The **ESC** key switches the shell to edit mode. The RETURN key always gives the line to the shell from either editing or input mode.

The following list is a summary of the commands available in edit mode.

Movement and search commands:

*n*G	- Go to command number *n*.
/*s*	- Search for string *s* backward in history.
?*s*	- Search for string *s* forward in history.
n	- Repeat last search.
N	- Repeat last search in opposite direction.
*n*k	- Get *n*th previous shell command in history.
n-	- Same as "k".
*n*j	- Get *n*th next shell command in history.
n+	- Same as "j".
*n*h	- Move left *n* characters.
CTRL-H	- Same as "h".
*n*l	- Move right *n* characters.
f1SPACEfP	- Same as "l".
*n*w	- Move *n* words forward.
*n*W	- Move *n* blank-separated words forward.
*n*e	- Move to end of the *n*th next word.
*n*E	- Move to end of the *n*th next blank-separated word.
*n*b	- Move back *n* words.

*n*B	- Move back *n* blank-separated words.
f*c*	- Find character *c*, searching forward.
F*c*	- Find character *c*, searching backward.
^	- Move cursor to first non-blank character in line.
$	- Go to end of line.
0	- Go to beginning of line.

Insert commands (input is expected until an **ESC** is typed):

a	- Append.
A	- Append at end of line.
c f1SPACEfP	- Change character.
cl	- Change character.
cw	- Change word.
cc	- Change entire line.
c$	- Change everything from cursor to end of line.
C	- Same as “c$”.
S	- Same as “cc”.
i	- Insert.
I	- Insert at beginning of line.
R	- Type over characters.

Editing commands:

*n*r*c*	- Replace the following *n* characters with *c*.
*n*x	- Delete *n* characters starting at cursor.
*n*X	- Delete *n* characters to the left of the cursor.
d f1SPACEfP	- Delete character.
dl	- Delete character.
dw	- Delete word.
dd	- Delete entire line.
d$	- Delete everything from cursor to end of line.
D	- Same as “d$”.
p	- Put last deletion after the cursor.
P	- Put last deletion before the cursor.
u	- Undo last command.
~	- Toggle case, lower to upper or vice versa.

Special commands:

CTRL-U	- Delete line and leave edit mode.
CTRL-L	- Redraw line.
CTRL-D	- Complete symbol name.
f1RETURNfP	- Give line to shell and leave edit mode.

The default value for *n* is 1.

DEFICIENCIES

Since the shell toggles between raw mode and line mode, type-ahead can be lost. The **ESC**, redraw, and non-printable characters are built-in. The **EOF**, backspace, and line-delete are not imported well from **tyLib**. Instead, **tyLib** should supply and/or support these characters via **ioctl( )**.

Some commands do not take counts as users might expect. For example, "*n*i" will not insert whatever was entered *n* times.

INCLUDE FILES

ledLib.h

SEE ALSO

VxWorks Programmer's Guide: Shell

L

loadLib

NAME

loadLib – object module loader

ROUTINES

loadModule( ) - load an object module into memory
loadModuleAt( ) - load an object module into memory

DESCRIPTION

This library provides a generic object module loading facility. Any supported format files may be loaded into memory, relocated properly, their external references resolved, and their external definitions added to the system symbol table for use by other modules and from the shell. Modules may be loaded from any I/O stream which allows repositioning of the pointer. This includes **netDrv**, NFS, or local file devices. It does not include sockets.

EXAMPLE

```
fdX = open ("/devX/objFile", O_RDONLY);
loadModule (fdX, LOAD_ALL_SYMBOLS);
close (fdX);
```

This code fragment would load the object file "objFile" located on device **/devX/** into memory which would be allocated from the system memory pool. All external and static definitions from the file would be added to the system symbol table.

This could also have been accomplished from the shell, by typing:

```
-> ld (1) </devX/objFile
```

INCLUDE FILE **loadLib.h**

SEE ALSO **usrLib**, **symLib**, **memLib**, *VxWorks Programmer's Guide: Basic OS*

loginLib

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

ROUTINES

loginInit( ) - initialize the login table
loginUserAdd( ) - add a user to the login table
loginUserDelete( ) - delete a user entry from the login table
loginUserVerify( ) - verify a user name and password in the login table
loginUserShow( ) - display the user login table
loginPrompt( ) - display a login prompt and validate a user entry
loginStringSet( ) - change the login string
loginEncryptInstall( ) - install an encryption routine
loginDefaultEncrypt( ) - default password encryption routine

DESCRIPTION

This library provides a login/password facility for network access to the VxWorks shell. When installed, it requires a user name and password match to gain access to the VxWorks shell from **rlogin** or **telnet**. Therefore VxWorks can be used in secure environments where access must be restricted.

Routines are provided to prompt for the user name and password, and verify the response by looking up the name/password pair in a login user table. This table contains a list of user names and encrypted passwords that will be allowed to log in to the VxWorks shell remotely. Routines are provided to add, delete, and access the login user table. The list of user names can be displayed with **loginUserShow( )**.

INSTALLATION

The login security feature is initialized by the root task, **usrRoot( )**, in **usrConfig.c**, if the configuration macro **INCLUDE_SECURITY** is defined. Defining this macro also adds a single default user to the login table. The default user and password are defined as **LOGIN_USER_NAME** and **LOGIN_PASSWORD**. These can be set to any desired name and password. More users can be added by making additional calls to **loginUserAdd( )**. If **INCLUDE_SECURITY** is not defined, access to VxWorks will not be restricted and secure.

The name/password pairs are added to the table by calling **loginUserAdd( )**, which takes the name and an encrypted password as arguments. The VxWorks host tool **vxencrypt** is used to generate the encrypted form of a password. For example, to add a user name of "fred" and password of "flintstone", first run **vxencrypt** on the host to find the encryption of "flintstone" as follows:

```
% vxencrypt
please enter password: flintstone
encrypted password is ScebRezb9c
```

Then invoke the routine **loginUserAdd( )** in VxWorks:

```
    loginUserAdd ("fred", "ScebRezb9c");
```

This can be done from the shell, a start-up script, or application code.

LOGGING IN

When the login security facility is installed, every attempt to **rlogin** or **telnet** to the VxWorks shell will first prompt for a user name and password.

```
% rlogin target
VxWorks login: fred
Password: flintstone
->
```

The delay in prompting between unsuccessful logins is increased linearly with the number of attempts, in order to slow down password-guessing programs.

ENCRYPTION ALGORITHM

This library provides a simple default encryption routine, **loginDefaultEncrypt( )**. This algorithm requires that passwords be at least 8 characters and no more than 40 characters.

The routine **loginEncryptInstall( )** allows a user-specified encryption function to be used instead of the default.

INCLUDE FILES

loginLib.h

SEE ALSO

shellLib, **vxencrypt**, *VxWorks Programmer's Guide: Shell*

logLib

NAME **logLib** – message logging library

ROUTINES **logInit( )** - initialize message logging library
logMsg( ) - log a formatted error message
logFdSet( ) - set the primary logging file descriptor
logFdAdd( ) - add a logging file descriptor
logFdDelete( ) - delete a logging file descriptor
logTask( ) - message-logging support task

DESCRIPTION This library handles message logging. It is usually used to display error messages on the system console, but such messages can also be sent to a disk file or printer.

The routines **logMsg( )** and **logTask( )** are the basic components of the logging system. The **logMsg( )** routine has the same calling sequence as **printf( )**, but instead of formatting and outputting the message directly, it sends the format string and arguments to a message queue. The task **logTask( )** waits for messages on this message queue. It formats each message according to the format string and arguments in the message, prepends the ID of the sender, and writes it on one or more file descriptors that have been specified as logging output streams (by **logInit( )** or subsequently set by **logFdSet( )** or **logFdAdd( )**).

USE IN INTERRUPT SERVICE ROUTINES

Because **logMsg( )** does not directly cause output to I/O devices, but instead simply writes to a message queue, it can be called from an ISR as well as from tasks. Normal I/O, such as **printf( )** output to a serial port, cannot be done from an ISR.

DEFERRED LOGGING

Print formatting is performed within the context of **logTask( )**, rather than the context of the task calling **logMsg( )**. Since formatting can require considerable stack space, this can reduce stack sizes for tasks that only need to do I/O for error output.

However, this also means that the arguments to **logMsg( )** are not interpreted at the time of the call to **logMsg( )**, but rather are interpreted at some later time by **logTask( )**. This means that the arguments to **logMsg( )** should not be pointers to volatile entities. For example, pointers to dynamic or changing strings and buffers should not be passed as arguments to be formatted. Thus the following would not give the desired results:

```
doLog (which)
    {
    char string [100];
    strcpy (string, which ? "hello" : "goodbye");
    ...
    logMsg (string);
    }
```

By the time **logTask( )** formats the message, the stack frame of the caller may no longer exist and the pointer *string* may no longer be valid. On the other hand, the following is correct since the string pointer passed to the **logTask( )** always points to a static string:

```
doLog (which)
    {
    char *string;
    string = which ? "hello" : "goodbye";
    ...
    logMsg (string);
    }
```

INITIALIZATION To initialize the message logging facilities, the routine **logInit( )** must be called before calling any other routine in this module. This is done by the root task, **usrRoot( )**, in **usrConfig.c**.

INCLUDE FILES **logLib.h**

SEE ALSO **msgQLib**, *VxWorks Programmer's Guide: I/O System*

lstLib

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

ROUTINES

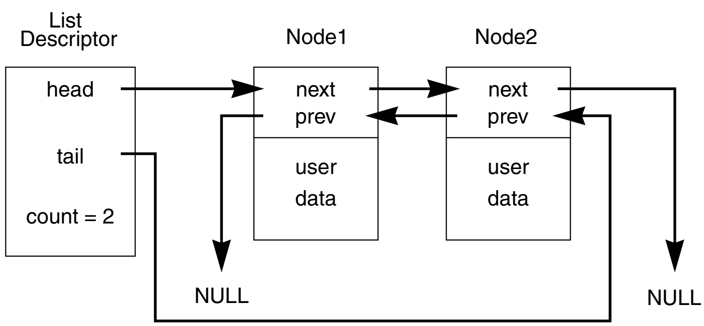
lstLibInit( ) - initializes **lstLib** module
lstInit( ) - initialize a list descriptor
lstAdd( ) - add a node to the end of a list
lstConcat( ) - concatenate two lists
lstCount( ) - report the number of nodes in a list
lstDelete( ) - delete a specified node from a list
lstExtract( ) - extract a sublist 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

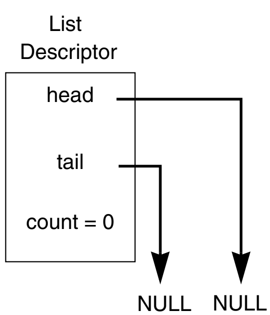
This subroutine library supports the creation and maintenance of a doubly linked list. The user supplies a list descriptor (type LIST) that will contain pointers to the first and last nodes in the list, and a count of the number of nodes in the list. The nodes in the list can be any user-defined structure, 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



EMPTY LIST



INCLUDE FILES

lstLib.h

m2IcmpLib

NAME **m2IcmpLib** – MIB-II ICMP-group API for SNMP Agents

ROUTINES **m2IcmpInit( )** - initialize MIB-II ICMP-group access
m2IcmpGroupInfoGet( ) - get the MIB-II ICMP-group global variables
m2IcmpDelete( ) - delete all resources used to access the ICMP group

DESCRIPTION This library provides MIB-II services for the ICMP group. It provides routines to initialize the group, and to access the group scalar variables. For a broader description of MIB-II services, see the manual entry for **m2Lib**.

To use this feature, include the following component: **INCLUDE_MIB2_ICMP**

USING THIS LIBRARY

This library can be initialized and deleted by calling the routines **m2IcmpInit( )** and **m2IcmpDelete( )** respectively, if only the ICMP group's services are needed. If full MIB-II support is used, this group and all other groups can be initialized and deleted by calling **m2Init( )** and **m2Delete( )**.

The group scalar variables are accessed by calling **m2IcmpGroupInfoGet( )** as follows:

```
M2_ICMP   icmpVars;
if (m2IcmpGroupInfoGet (&icmpVars) == OK)
    /* values in icmpVars are valid */
```

INCLUDE FILES **m2Lib.h**

SEE ALSO **m2Lib**, **m2IfLib**, **m2IpLib**, **m2TcpLib**, **m2SysLib**

m2IfLib

NAME **m2IfLib** – MIB-II interface-group API for SNMP agents

ROUTINES **m2IfAlloc( )** - allocate the structure for the interface table
m2IfFree( ) - free an interface data structure
m2IfGenericPacketCount( ) - increment the interface packet counters
m2If8023PacketCount( ) - increment the packet counters for an 802.3 device
m2IfCounterUpdate( ) - increment interface counters
m2IfVariableUpdate( ) - update the contents of an interface non-counter object
m2IfPktCountRtnInstall( ) - install an interface packet counter routine
m2IfCtrUpdateRtnInstall( ) - install an interface counter update routine

m2IfVarUpdateRtnInstall( ) - install an interface variable update routine
m2IfInit( ) - initialize MIB-II interface-group routines
m2IfTableUpdate( ) - insert or remove an entry in the **ifTable**
rcvEtherAddrGet( ) - populate the *rcvAddr* fields for the **ifRcvAddressTable**
rcvEtherAddrAdd( ) - add a physical address into the linked list
m2IfTblEntryGet( ) - get a MIB-II interface-group table entry
m2IfDefaultValsGet( ) - get the default values for the counters
m2IfCommonValsGet( ) - get the common values
m2IfTblEntrySet( ) - set the state of a MIB-II interface entry to UP or DOWN
m2IfGroupInfoGet( ) - get the MIB-II interface-group scalar variables
m2IfStackTblUpdate( ) - update the relationship between the sub-layers
stackEntryIsTop( ) - test if an **ifStackTable** interface has no layers above
stackEntryIsBottom( ) - test if an interface has no layers beneath it
m2IfStackEntryGet( ) - get a MIB-II interface-group table entry
m2IfStackEntrySet( ) - modify the status of a relationship
m2IfRcvAddrEntryGet( ) - get the *rcvAddress* table entries for a given address
m2IfRcvAddrEntrySet( ) - modify the entries of the **rcvAddressTable**
m2IfDelete( ) - delete all resources used to access the interface group
nextIndex( ) - the comparison routine for the AVL tree

DESCRIPTION

This library provides MIB-II services for the interface group. It provides routines to initialize the group, access the group scalar variables, read the table interfaces and change the state of the interfaces. For a broader description of MIB-II services, see the manual entry for **m2Lib**.

To use this feature, include the following component: **INCLUDE_MIB2_IF**

USING THIS LIBRARY

This library can be initialized and deleted by calling **m2IfInit( )** and **m2IfDelete( )** respectively, if only the interface group's services are needed. If full MIB-II support is used, this group and all other groups can be initialized and deleted by calling **m2Init( )** and **m2Delete( )**.

The interface group supports the Simple Network Management Protocol (SNMP) concept of traps, as specified by RFC 1215. The traps supported by this group are "link up" and "link down." This library enables an application to register a hook routine and an argument. This hook routine can be called by the library when a "link up" or "link down" condition is detected. The hook routine must have the following prototype:

```
void TrapGenerator (int trapType,  /* M2_LINK_DOWN_TRAP or M2_LINK_UP_TRAP */
                    int interfaceIndex,
                    void * myPrivateArg);
```

The trap routine and argument can be specified at initialization time as input parameters to the routine **m2IfInit( )** or to the routine **m2Init( )**.

The interface-group global variables can be accessed as follows:

```
M2_INTERFACE   ifVars;
if (m2IfGroupInfoGet (&ifVars) == OK)
    /* values in ifVars are valid */
```

An interface table entry can be retrieved as follows:

```
M2_INTERFACETBL  interfaceEntry;
/* Specify zero as the index to get the first entry in the table */
interfaceEntry.ifIndex = 2;     /* Get interface with index 2 */
if (m2IfTblEntryGet (M2_EXACT_VALUE, &interfaceEntry) == OK)
    /* values in interfaceEntry are valid */
```

An interface entry operational state can be changed as follows:

```
M2_INTERFACETBL ifEntryToSet;
ifEntryToSet.ifIndex = 2; /* Select interface with index 2     */
                          /* MIB-II value to set the interface */
                          /* to the down state.                */
ifEntryToSet.ifAdminStatus = M2_ifAdminStatus_down;
if (m2IfTblEntrySet (&ifEntryToSet) == OK)
    /* Interface is now in the down state */
```

INCLUDE FILES **m2Lib.h**

SEE ALSO **m2Lib**, **m2SysLib**, **m2IpLib**, **m2IcmpLib**, **m2UdpLib**, **m2TcpLib**

M

m2Igmp

NAME **m2Igmp** – helper file for igmp Mib

ROUTINES No Callable Routines.

DESCRIPTION This library provides an interface between the Berkeley multicast code and the IGMP Mib code

INCLUDE FILES **m2Lib.h**

m2IpLib

NAME

m2IpLib – MIB-II IP-group API for SNMP agents

ROUTINES

m2IpInit( ) - initialize MIB-II IP-group access
m2IpGroupInfoGet( ) - get the MIB-II IP-group scalar variables
m2IpGroupInfoSet( ) - set MIB-II IP-group variables to new values
m2IpAddrTblEntryGet( ) - get an IP MIB-II address entry
m2IpAtransTblEntryGet( ) - get a MIB-II ARP table entry
m2IpAtransTblEntrySet( ) - add, modify, or delete a MIB-II ARP entry
m2IpRouteTblEntryGet( ) - get a MIB-2 routing table entry
m2IpRouteTblEntrySet( ) - set a MIB-II routing table entry
m2IpDelete( ) - delete all resources used to access the IP group

DESCRIPTION

This library provides MIB-II services for the IP group. It provides routines to initialize the group, access the group scalar variables, read the table IP address, route and ARP table. The route and ARP table can also be modified. For a broader description of MIB-II services, see the manual entry for **m2Lib**.

To use this feature, include the following component: **INCLUDE_MIB2_IP**

USING THIS LIBRARY

To use this library, the MIB-II interface group must also be initialized; see the manual entry for **m2IfLib**. This library (**m2IpLib**) can be initialized and deleted by calling **m2IpInit( )** and **m2IpDelete( )** respectively, if only the IP group's services are needed. If full MIB-II support is used, this group and all other groups can be initialized and deleted by calling **m2Init( )** and **m2Delete( )**.

The following example demonstrates how to access and change IP scalar variables:

```
M2_IP   ipVars;
int     varToSet;
if (m2IpGroupInfoGet (&ipVars) == OK)
    /* values in ipVars are valid */
/* if IP is forwarding packets (MIB-II value is 1) turn it off */
if (ipVars.ipForwarding == M2_ipForwarding_forwarding)
    {
                                /* Not forwarding (MIB-II value is 2) */
    ipVars.ipForwarding = M2_ipForwarding_not_forwarding;
    varToSet |= M2_IPFORWARDING;
    }
/* change the IP default time to live parameter */
ipVars.ipDefaultTTL = 55;
if (m2IpGroupInfoSet (varToSet, &ipVars) == OK)
    /* values in ipVars are valid */
```

The IP address table is a read-only table. Entries to this table can be retrieved as follows:

```
M2_IPADDRTBL ipAddrEntry;
/* Specify the index as zero to get the first entry in the table */
ipAddrEntry.ipAdEntAddr = 0;    /* Local IP address in host byte order */
/* get the first entry in the table */
if ((m2IpAddrTblEntryGet (M2_NEXT_VALUE, &ipAddrEntry) == OK)
    /* values in ipAddrEntry in the first entry are valid  */
/* Process first entry in the table */
/*
 * For the next call, increment the index returned in the previous call.
 * The increment is to the next possible lexicographic entry; for
 * example, if the returned index was 147.11.46.8 the index passed in the
 * next invocation should be 147.11.46.9.  If an entry in the table
 * matches the specified index, then that entry is returned.
 * Otherwise the closest entry following it, in lexicographic order,
 * is returned.
 */
/* get the second entry in the table */
if ((m2IpAddrTblEntryGet (M2_NEXT_VALUE, &ipAddrEntryEntry) == OK)
    /* values in ipAddrEntry in the second entry are valid  */
```

The IP Address Translation Table (ARP table) includes the functionality of the AT group plus additional functionality. The AT group is supported through this MIB-II table. Entries in this table can be added and deleted. An entry is deleted (with a set operation) by setting the **ipNetToMediaType** field to the MIB-II "invalid" value (2). The following example shows how to delete an entry:

```
M2_IPATRANSTBL        atEntry;
    /* Specify the index for the connection to be deleted in the table */
    atEntry.ipNetToMediaIfIndex     = 1       /* interface index */
    /* destination IP address in host byte order */
    atEntry.ipNetToMediaNetAddress  = 0x930b2e08;
                                            /* mark entry as invalid */
    atEntry.ipNetToMediaType        = M2_ipNetToMediaType_invalid;
    /* set the entry in the table */
    if ((m2IpAtransTblEntrySet (&atEntry) == OK)
        /* Entry deleted successfully */
```

The IP route table allows for entries to be read, deleted, and modified. This example demonstrates how an existing route is deleted:

```
M2_IPROUTETBL        routeEntry;
/* Specify the index for the connection to be deleted in the table */
/* destination IP address in host byte order */
routeEntry.ipRouteDest       = 0x930b2e08;
                                            /* mark entry as invalid */
routeEntry.ipRouteType       = M2_ipRouteType_invalid;
```

```
    /* set the entry in the table */
    if ((m2IpRouteTblEntrySet (M2_IP_ROUTE_TYPE, &routeEntry) == OK)
        /* Entry deleted successfully */
```

INCLUDE FILES **m2Lib.h**

SEE ALSO **m2Lib**, **m2SysLib**, **m2IfLib**, **m2IcmpLib**, **m2UdpLib**, **m2TcpLib**

m2Lib

NAME **m2Lib** – MIB-II API library for SNMP agents

ROUTINES
m2Init( ) - initialize the SNMP MIB-2 library
m2Delete( ) - delete all the MIB-II library groups

DESCRIPTION This library provides Management Information Base (MIB-II, defined in RFC 1213) services for applications wishing to have access to MIB parameters.

To use this feature, include the following component: **INCLUDE_MIB2_ALL**

There are no specific provisions for MIB-I: all services are provided at the MIB-II level. Applications that use this library for MIB-I must hide the MIB-II extensions from higher level protocols. The library accesses all the MIB-II parameters, and presents them to the application in data structures based on the MIB-II specifications.

The routines provided by the VxWorks MIB-II library are separated into groups that follow the MIB-II definition. Each supported group has its own interface library:

m2SysLib
systems group

m2IfLib
interface group

m2IpLib
IP group (includes AT)

m2IcmpLib
ICMP group

m2TcpLib
TCP group

m2UdpLib
UDP group

MIB-II retains the AT group for backward compatibility, but includes its functionality in the IP group. The EGP and SNMP groups are not supported by this interface. The variables in each group have been subdivided into two types: table entries and scalar variables. Each type has a pair of routines that get and set the variables.

USING THIS LIBRARY

There are four types of operations on each group:

- initializing the group
- getting variables and table entries
- setting variables and table entries
- deleting the group

Only the groups that are to be used need be initialized. There is one exception: to use the IP group, the interface group must also be initialized. Applications that require MIB-II support from all groups can initialize all groups at once by calling the **m2Init( )**. All MIB-II group services can be disabled by calling **m2Delete( )**. Applications that need access only to a particular set of groups need only call the initialization routines of the desired groups.

To read the scalar variables for each group, call one of the following routines:

m2SysGroupInfoGet( )
m2IfGroupInfoGet( )
m2IpGroupInfoGet( )
m2IcmpGroupInfoGet( )
m2TcpGroupInfoGet( )
m2UdpGroupInfoGet( )

The input parameter to the routine is always a pointer to a structure specific to the associated group. The scalar group structures follow the naming convention "M2_*groupname*". The get routines fill in the input structure with the values of all the group variables.

The scalar variables can also be set to a user supplied value. Not all groups permit setting variables, as specified by the MIB-II definition. The following group routines allow setting variables:

m2SysGroupInfoSet( )
m2IpGroupInfoSet( )

The input parameters to the variable-set routines are a bit field that specifies which variables to set, and a group structure. The structure is the same structure type used in the get operation. Applications need set only the structure fields corresponding to the bits that are set in the bit field.

The MIB-II table routines read one entry at a time. Each MIB-II group that has tables has a get routine for each table. The following table-get routines are available:

m2IfTblEntryGet( )
m2IpAddrTblEntryGet( )

m2IpAtransTblEntryGet( )
m2IpRouteTblEntryGet( )
m2TcpConnEntryGet( )
m2UdpTblEntryGet( )

The input parameters are a pointer to a table entry structure, and a flag value specifying one of two types of table search. Each table entry is a structure, where the struct type name follows this naming convention: "M2_*GroupnameTablename*TBL". The MIB-II RFC specifies an index that identifies a table entry. Each get request must specify an index value. To retrieve the first entry in a table, set all the index fields of the table-entry structure to zero, and use the search parameter **M2_NEXT_VALUE**. To retrieve subsequent entries, pass the index returned from the previous invocation, incremented to the next possible lexicographical entry. The search field can only be set to the constants **M2_NEXT_VALUE** or **M2_EXACT_VALUE**:

M2_NEXT_VALUE
retrieves a table entry that is either identical to the index value specified as input, or is the closest entry following that value, in lexicographic order.

M2_EXACT_VALUE
retrieves a table entry that exactly matches the index specified in the input structure.

Some MIB-II table entries can be added, modified and deleted. Routines to manipulate such entries are described in the manual pages for individual groups.

All the IP network addresses that are exchanged with the MIB-II library must be in host-byte order; use **ntohl( )** to convert addresses before calling these library routines.

The following example shows how to initialize the MIB-II library for all groups.

```
extern FUNCPTR myTrapGenerator;
extern void *  myTrapGeneratorArg;
M2_OBJECTID mySysObjectId = { 8, {1,3,6,1,4,1,731,1} };
if (m2Init ("VxWorks 5.1.1 MIB-II library (sysDescr)",
            "support@wrs.com (sysContact)",
            "500 Wind River Way Alameda, California 94501 (sysLocation)",
             &mySysObjectId,
             myTrapGenerator,
             myTrapGeneratorArg,
             0) == OK)
    /* MIB-II groups initialized successfully */
```

INCLUDE FILES **m2Lib.h**

SEE ALSO **m2IfLib**, **m2IpLib**, **m2IcmpLib**, **m2UdpLib**, **m2TcpLib**, **m2SysLib**

m2RipLib

NAME

m2RipLib – VxWorks interface routines to RIP for SNMP Agent

ROUTINES

m2RipInit( ) - initialize the RIP MIB support
m2RipDelete( ) - delete the RIP MIB support
m2RipGlobalCountersGet( ) - get MIB-II RIP-group global counters
m2RipIfStatEntryGet( ) - get MIB-II RIP-group interface entry
m2RipIfConfEntryGet( ) - get MIB-II RIP-group interface entry
m2RipIfConfEntrySet( ) - set MIB-II RIP-group interface entry

DESCRIPTION

This library provides routines to initialize the group, access the group global variables, read the table of network interfaces that RIP knows about, and change the state of such an interface. For a broader description of MIB-II services, see the manual entry for **m2Lib**.

USING THIS LIBRARY

This library can be initialized and deleted by calling **m2RipInit( )** and **m2RipDelete( )** respectively, if only the RIP group's services are needed. If full MIB-II support is used, this group and all other groups can be initialized and deleted by calling **m2Init( )** and **m2Delete( )**.

The group global variables are accessed by calling **m2RipGlobalCountersGet( )** as follows:

```
M2_RIP2_GLOBAL_GROUP   ripGlobal;
if (m2RipGlobalCountersGet (&ripGlobal) == OK)
    /* values in ripGlobal are valid */
```

To retrieve the RIP group statistics for a particular interface you call the **m2RipIfStatEntryGet( )** routine a pointer to an **M2_RIP2_IFSTAT_ENTRY** structure that contains the address of the interface you are searching for. For example:

```
M2_RIP2_IFSTAT_ENTRY ripIfStat;

    ripIfStat.rip2IfStatAddress = inet_addr("90.0.0.3");
    if (m2RipIfStatEntryGet(M2_EXACT_VALUE, &ripIfStat) == OK)
    /* values in ripIfState are valid */
```

To retrieve the configuration statistics for a particular interface the **m2RipIfConfEntryGet( )** routine must be called with an IP address encoded in an **M2_RIP2_IFSTAT_ENTRY** structure which is passed as the second argument. For example:

```
M2_RIP2_IFCONF_ENTRY ripIfConf;

    ripIfConf.rip2IfConfAddress = inet_addr("90.0.0.3");
    if (m2RipIfConfEntryGet(M2_EXACT_VALUE, &ripIfConf) == OK)
    /* values in ripIfConf are valid */
```

To set the values of for an interface the **m2RipIfConfEntrySet( )** routine must be called with an IP address in dot notation encoded into an **M2_RIP2_IFSTAT_ENTRY** structure, which is passed as the second argument. For example:

```
M2_RIP2_IFCONF_ENTRY ripIfConf;
    ripIfConf.rip2IfConfAddress = inet_addr("90.0.0.3");
    /* Set the authorization type. */
    ripIfConf.rip2IfConfAuthType = M2_rip2IfConfAuthType_simplePassword;
    bzero(ripIfConf.rip2IfConfAuthKey, 16);
    bcopy("Simple Password ", ripIfConf.rip2IfConfAuthKey, 16);
    /* We only accept version 1 packets. */
    ripIfConf.rip2IfConfSend = M2_rip2IfConfSend_ripVersion1;
    /* We only send version 1 packets. */
    ripIfConf.rip2IfConfReceive = M2_rip2IfConfReceive_rip1;
    /* Default routes have a metric of 2 */
    ripIfConf.rip2IfConfDefaultMetric = 2;
    /* If the interface is invalid it is turned off, we make it valid. */
    ripIfConf.rip2IfConfStatus = M2_rip2IfConfStatus_valid;

    if (m2RipIfConfEntrySet(varsToSet, &ripIfConf) == OK)
    /* Call succeded. */
```

INCLUDE FILES **rip/m2RipLib.h**, **rip/defs.h**

SEE ALSO **ripLib**

m2SysLib

NAME **m2SysLib** – MIB-II system-group API for SNMP agents

ROUTINES

m2SysInit( ) - initialize MIB-II system-group routines
m2SysGroupInfoGet( ) - get system-group MIB-II variables
m2SysGroupInfoSet( ) - set system-group MIB-II variables to new values
m2SysDelete( ) - delete resources used to access the MIB-II system group

DESCRIPTION

This library provides MIB-II services for the system group. It provides routines to initialize the group and to access the group scalar variables. For a broader description of MIB-II services, see the manual entry for **m2Lib**.

To use this feature, include the following component: **INCLUDE_MIB2_SYSTEM**

USING THIS LIBRARY

This library can be initialized and deleted by calling **m2SysInit( )** and **m2SysDelete( )**

respectively, if only the system group's services are needed. If full MIB-II support is used, this group and all other groups can be initialized and deleted by calling **m2Init( )** and **m2Delete( )**.

The system group provides the option to set the system variables at the time **m2Sysinit( )** is called. The MIB-II variables **sysDescr** and **sysobjectId** are read-only, and can be set only by the system-group initialization routine. The variables **sysContact**, **sysName** and **sysLocation** can be set through **m2SysGroupInfoSet( )** at any time.

The following is an example of system group initialization:

```
M2_OBJECTID mySysObjectId = { 8, {1,3,6,1,4,1,731,1} };
if (m2SysInit ("VxWorks MIB-II library ",
               "support@wrs.com",
               "1010 Atlantic Avenue Alameda, California 94501",
               &mySysObjectId) == OK)
    /* System group initialized successfully */
```

The system group variables can be accessed as follows:

```
M2_SYSTEM   sysVars;
if (m2SysGroupInfoGet (&sysVars) == OK)
    /* values in sysVars are valid */
```

The system group variables can be set as follows:

```
M2_SYSTEM    sysVars;
unsigned int varToSet;      /* bit field of variables to set */
/* Set the new system Name */
strcpy (m2SysVars.sysName, "New System Name");
varToSet |= M2SYSNAME;
/* Set the new contact name */
strcpy (m2SysVars.sysContact, "New Contact");
varToSet |= M2SYSCONTACT;
if (m2SysGroupInfoGet (varToSet, &sysVars) == OK)
    /* values in sysVars set */
```

INCLUDE FILES **m2Lib.h**

SEE ALSO **m2Lib**, **m2IfLib**, **m2IpLib**, **m2IcmpLib**, **m2UdpLib**, **m2TcpLib**

M

m2TcpLib

NAME

m2TcpLib – MIB-II TCP-group API for SNMP agents

ROUTINES

m2TcpInit( ) - initialize MIB-II TCP-group access
m2TcpGroupInfoGet( ) - get MIB-II TCP-group scalar variables
m2TcpConnEntryGet( ) - get a MIB-II TCP connection table entry
m2TcpConnEntrySet( ) - set a TCP connection to the closed state
m2TcpDelete( ) - delete all resources used to access the TCP group

DESCRIPTION

This library provides MIB-II services for the TCP group. It provides routines to initialize the group, access the group global variables, read the table of TCP connections, and change the state of a TCP connection. For a broader description of MIB-II services, see the manual entry for **m2Lib**.

To use this feature, include the following component: **INCLUDE_MIB2_TCP**

USING THIS LIBRARY

This library can be initialized and deleted by calling **m2TcpInit( )** and **m2TcpDelete( )** respectively, if only the TCP group's services are needed. If full MIB-II support is used, this group and all other groups can be initialized and deleted by calling **m2Init( )** and **m2Delete( )**.

The group global variables are accessed by calling **m2TcpGroupInfoGet( )** as follows:

```
M2_TCP   tcpVars;
if (m2TcpGroupInfoGet (&tcpVars) == OK)
    /* values in tcpVars are valid */
```

The TCP table of connections can be accessed in lexicographical order. The first entry in the table can be accessed by setting the table index to zero. Every other entry thereafter can be accessed by passing to **m2TcpConnTblEntryGet( )** the index retrieved in the previous invocation incremented to the next lexicographical value by giving **M2_NEXT_VALUE** as the search parameter. For example:

```
M2_TCPCONNTBL  tcpEntry;
    /* Specify a zero index to get the first entry in the table */
    tcpEntry.tcpConnLocalAddress = 0; /* Lcl IP address in host byte order */
    tcpEntry.tcpConnLocalPort    = 0; /* Local TCP port                    */
    tcpEntry.tcpConnRemAddress   = 0; /* remote IP address                 */
    tcpEntry.tcpConnRemPort    = 0; /* remote TCP port in host byte order  */
    /* get the first entry in the table */
    if ((m2TcpConnTblEntryGet (M2_NEXT_VALUE, &tcpEntry) == OK)
        /* values in tcpEntry in the first entry are valid  */
    /* process first entry in the table */
    /*
```

```
 * For the next call, increment the index returned in the previous call.
 * The increment is to the next possible lexicographic entry; for
 * example, if the returned index was 147.11.46.8.2000.147.11.46.158.1000
 * the index passed in the next invocation should be
 * 147.11.46.8.2000.147.11.46.158.1001.  If an entry in the table
 * matches the specified index, then that entry is returned.
 * Otherwise the closest entry following it, in lexicographic order,
 * is returned.
 */
/* get the second entry in the table */
if ((m2TcpConnTblEntryGet (M2_NEXT_VALUE, &tcpEntry) == OK)
    /* values in tcpEntry in the second entry are valid  */
```

The TCP table of connections allows only for a connection to be deleted as specified in the MIB-II. For example:

```
M2_TCPCONNTBL  tcpEntry;
/* Fill in the index for the connection to be deleted in the table */
/* Local IP address in host byte order, and local port number */
tcpEntry.tcpConnLocalAddress = 0x930b2e08;
tcpEntry.tcpConnLocalPort    = 3000;
/* Remote IP address in host byte order, and remote port number */
tcpEntry.tcpConnRemAddress   = 0x930b2e9e;
tcpEntry.tcpConnRemPort      = 3000;
tcpEntry.tcpConnState        = 12;  /* MIB-II state value for delete */
/* set the entry in the table */
if ((m2TcpConnTblEntrySet (&tcpEntry) == OK)
    /* tcpEntry deleted successfuly */
```

INCLUDE FILES **m2Lib.h**

SEE ALSO **m2Lib**, **m2IfLib**, **m2IpLib**, **m2IcmpLib**, **m2UdpLib**, **m2SysLib**

M

m2UdpLib

NAME

m2UdpLib – MIB-II UDP-group API for SNMP agents

ROUTINES

m2UdpInit( ) - initialize MIB-II UDP-group access
m2UdpGroupInfoGet( ) - get MIB-II UDP-group scalar variables
m2UdpTblEntryGet( ) - get a UDP MIB-II entry from the UDP list of listeners
m2UdpDelete( ) - delete all resources used to access the UDP group

DESCRIPTION

This library provides MIB-II services for the UDP group. It provides routines to initialize the group, access the group scalar variables, and read the table of UDP listeners. For a broader description of MIB-II services, see the manual entry for **m2Lib**.

To use this feature, include the following component: **INCLUDE_MIB2_UDP**

USING THIS LIBRARY

This library can be initialized and deleted by calling **m2UdpInit( )** and **m2UdpDelete( )** respectively, if only the UDP group's services are needed. If full MIB-II support is used, this group and all other groups can be initialized and deleted by calling **m2Init( )** and **m2Delete( )**.

The group scalar variables are accessed by calling **m2UdpGroupInfoGet( )** as follows:

```
M2_UDP   udpVars;
if (m2UdpGroupInfoGet (&udpVars) == OK)
    /* values in udpVars are valid */
```

The UDP table of listeners can be accessed in lexicographical order. The first entry in the table can be accessed by setting the table index to zero in a call to **m2UdpTblEntryGet( )**. Every other entry thereafter can be accessed by incrementing the index returned from the previous invocation to the next possible lexicographical index, and repeatedly calling **m2UdpTblEntryGet( )** with the **M2_NEXT_VALUE** constant as the search parameter. For example:

```
M2_UDPTBL  udpEntry;
    /* Specify zero index to get the first entry in the table */
    udpEntry.udpLocalAddress = 0;  /* local IP Address in host byte order  */
    udpEntry.udpLocalPort    = 0;  /* local port Number                    */
    /* get the first entry in the table */
    if ((m2UdpTblEntryGet (M2_NEXT_VALUE, &udpEntry) == OK)
        /* values in udpEntry in the first entry are valid  */
    /* process first entry in the table */
    /*
     * For the next call, increment the index returned in the previous call.
     * The increment is to the next possible lexicographic entry; for
     * example, if the returned index was 0.0.0.0.3000 the index passed in
```

```
 * the next invocation should be 0.0.0.0.3001.  If an entry in the table
 * matches the specified index, then that entry is returned.
 * Otherwise the closest entry following it, in lexicographic order,
 * is returned.
 */
/* get the second entry in the table */
if ((m2UdpTblEntryGet (M2_NEXT_VALUE, &udpEntry) == OK)
    /* values in udpEntry in the second entry are valid  */
```

INCLUDE FILES **m2Lib.h**

SEE ALSO **m2Lib**, **m2IfLib**, **m2IpLib**, **m2IcmpLib**, **m2TcpLib**, **m2SysLib**

mathALib

NAME **mathALib** – C interface library to high-level math functions

ROUTINES

acos( ) - compute an arc cosine (ANSI)
asin( ) - compute an arc sine (ANSI)
atan( ) - compute an arc tangent (ANSI)
atan2( ) - compute the arc tangent of y/x (ANSI)
cbrt( ) - compute a cube root
ceil( ) - compute the smallest integer greater than or equal to a specified value (ANSI)
cos( ) - compute a cosine (ANSI)
cosh( ) - compute a hyperbolic cosine (ANSI)
exp( ) - compute an exponential value (ANSI)
fabs( ) - compute an absolute value (ANSI)
floor( ) - compute the largest integer less than or equal to a specified value (ANSI)
fmod( ) - compute the remainder of x/y (ANSI)
infinity( ) - return a very large double
irint( ) - convert a double-precision value to an integer
iround( ) - round a number to the nearest integer
log( ) - compute a natural logarithm (ANSI)
log10( ) - compute a base-10 logarithm (ANSI)
log2( ) - compute a base-2 logarithm
pow( ) - compute the value of a number raised to a specified power (ANSI)
round( ) - round a number to the nearest integer
sin( ) - compute a sine (ANSI)
sincos( ) - compute both a sine and cosine
sinh( ) - compute a hyperbolic sine (ANSI)
sqrt( ) - compute a non-negative square root (ANSI)
tan( ) - compute a tangent (ANSI)

tanh( ) - compute a hyperbolic tangent (ANSI)
trunc( ) - truncate to integer
acosf( ) - compute an arc cosine (ANSI)
asinf( ) - compute an arc sine (ANSI)
atanf( ) - compute an arc tangent (ANSI)
atan2f( ) - compute the arc tangent of y/x (ANSI)
cbrtf( ) - compute a cube root
ceilf( ) - compute the smallest integer greater than or equal to a specified value (ANSI)
cosf( ) - compute a cosine (ANSI)
coshf( ) - compute a hyperbolic cosine (ANSI)
expf( ) - compute an exponential value (ANSI)
fabsf( ) - compute an absolute value (ANSI)
floorf( ) - compute the largest integer less than or equal to a specified value (ANSI)
fmodf( ) - compute the remainder of x/y (ANSI)
infinityf( ) - return a very large float
irintf( ) - convert a single-precision value to an integer
iroundf( ) - round a number to the nearest integer
logf( ) - compute a natural logarithm (ANSI)
log10f( ) - compute a base-10 logarithm (ANSI)
log2f( ) - compute a base-2 logarithm
powf( ) - compute the value of a number raised to a specified power (ANSI)
roundf( ) - round a number to the nearest integer
sinf( ) - compute a sine (ANSI)
sincosf( ) - compute both a sine and cosine
sinhf( ) - compute a hyperbolic sine (ANSI)
sqrtf( ) - compute a non-negative square root (ANSI)
tanf( ) - compute a tangent (ANSI)
tanhf( ) - compute a hyperbolic tangent (ANSI)
truncf( ) - truncate to integer

DESCRIPTION

This library provides a C interface to high-level floating-point math functions, which can use either a hardware floating-point unit or a software floating-point emulation library. The appropriate routine is called based on whether **mathHardInit( )** or **mathSoftInit( )** or both have been called to initialize the interface.

All angle-related parameters are expressed in radians. All functions in this library with names corresponding to ANSI C specifications are ANSI compatible.

WARNING: Not all functions in this library are available on all architectures. For information on available math functions, consult the VxWorks architecture supplement for your processor.

INCLUDE FILES

math.h

SEE ALSO

ansiMath, **fppLib**, **floatLib**, **mathHardLib**, **mathSoftLib**, the various *Architecture Supplements*, Kernighan & Ritchie: *The C Programming Language*, *2nd Edition*

mathHardLib

NAME
mathHardLib – hardware floating-point math library

ROUTINES
mathHardInit( ) - initialize hardware floating-point math support

DESCRIPTION
This library provides support routines for using hardware floating-point units with high-level math functions. The high-level functions include trigonometric operations, exponents, and so forth.

The routines in this library are used automatically for high-level math functions only if **mathHardInit( )** has been called previously.

WARNING: Not all architectures support hardware floating-point. See the architecture-specific appendices of the *VxWorks Programmer's Guide.*

INCLUDE FILES
math.h

SEE ALSO
mathSoftLib, **mathALib**, *VxWorks Programmer's Guide* architecture-specific appendices

mathSoftLib

NAME
mathSoftLib – high-level floating-point emulation library

ROUTINES
mathSoftInit( ) - initialize software floating-point math support

DESCRIPTION
This library provides software emulation of various high-level floating-point operations. This emulation is generally for use in systems that lack a floating-point coprocessor.

WARNING: Software floating point is not supported for all architectures. See the architecture-specific appendices of the *VxWorks Programmer's Guide.*

INCLUDE FILES
math.h

SEE ALSO
mathHardLib, **mathALib**, *VxWorks Programmer's Guide* architecture-specific appendices

memDrv

NAME

memDrv – pseudo-memory device driver

ROUTINES

memDrv( ) - install a memory driver
memDevCreate( ) - create a memory device
memDevCreateDir( ) - create a memory device for multiple files
memDevDelete( ) - delete a memory device

DESCRIPTION

This driver allows the I/O system to access memory directly as a pseudo-I/O device. Memory location and size are specified when the device is created. This feature is useful when data must be preserved between boots of VxWorks or when sharing data between CPUs.

Additionally, it can be used to build some files into a VxWorks binary image (having first converted them to data arrays in C source files, using a utility such as **memdrvbuild**), and then mount them in the file system; this is a simple way of delivering some non-changing files with VxWorks. For example, a system with an integrated web server may use this technique to build some HTML and associated content files into VxWorks.

memDrv can be used to simply provide a high-level method of reading and writing bytes in absolute memory locations through I/O calls. It can also be used to implement a simple, essentially read-only file system (existing files can be rewritten within their existing sizes); directory searches and a limited set of IOCTL calls (including **stat( )**) are supported.

USER-CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. Four routines, however, can be called directly: **memDrv( )** to initialize the driver, **memDevCreate( )** and **memDevCreateDir( )** to create devices, and **memDevDelete( )** to delete devices.

Before using the driver, it must be initialized by calling **memDrv( )**. This routine should be called only once, before any reads, writes, or **memDevCreate( )** calls. It may be called from **usrRoot( )** in **usrConfig.c** or at some later point.

IOCTL FUNCTIONS

The dosFs file system supports the following **ioctl( )** functions. The functions listed are defined in the header **ioLib.h**. Unless stated otherwise, the file descriptor used for these functions may be any file descriptor which is opened to a file or directory on the volume or to the volume itself.

FIOGETFL

Copies to *flags* the open mode flags of the file (**O_RDONLY**, **O_WRONLY**, **O_RDWR**):

```
int flags;
status = ioctl (fd, FIOGETFL, &flags);
```

FIOSEEK

Sets the current byte offset in the file to the position specified by *newOffset*:

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

The **FIOSEEK** offset is always relative to the beginning of the file. The offset, if any, given at open time by using pseudo-file name is overridden.

FIOWHERE

Returns the current byte position in the file. This is the byte offset of the next byte to be read or written. It takes no additional argument:

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

FIONREAD

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

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

FIOREADDIR

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

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

FIOFSTATGET

Gets file status information (directory entry data). The argument *statStruct* is a pointer to a stat structure that is filled with data describing the specified file. File inode numbers, user and group IDs, and times are not supported (returned as 0).

Normally, the **stat( )** or **fstat( )** routine is used to obtain file information, rather than using the **FIOFSTATGET** function directly. See **dirLib**.

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

Any other **ioctl( )** function codes will return error status.

SEE ALSO *VxWorks Programmer's Guide: I/O System*

memLib

NAME

memLib – full-featured memory partition manager

ROUTINES

memPartOptionsSet( ) - set the debug options for a memory partition
memalign( ) - allocate aligned memory
valloc( ) - allocate memory on a page boundary
memPartRealloc( ) - reallocate a block of memory in a specified partition
memPartFindMax( ) - find the size of the largest available free block
memOptionsSet( ) - set the debug options for the system memory partition
calloc( ) - allocate space for an array (ANSI)
realloc( ) - reallocate a block of memory (ANSI)
cfree( ) - free a block of memory
memFindMax( ) - find the largest free block in the system memory partition

DESCRIPTION

This library provides full-featured facilities for managing the allocation of blocks of memory from ranges of memory called memory partitions. The library is an extension of **memPartLib** and provides enhanced memory management features, including error handling, aligned allocation, and ANSI allocation routines. For more information about the core memory partition management facility, see the manual entry for **memPartLib**.

The system memory partition is created when the kernel is initialized by **kernelInit( )**, which is called by the root task, **usrRoot( )**, in **usrConfig.c**. The ID of the system memory partition is stored in the global variable **memSysPartId**; its declaration is included in **memLib.h**.

The **memalign( )** routine is provided for allocating memory aligned to a specified boundary.

This library includes three ANSI-compatible routines: **calloc( )** allocates a block of memory for an array; **realloc( )** changes the size of a specified block of memory; and **cfree( )** returns to the free memory pool a block of memory that was previously allocated with **calloc( )**.

ERROR OPTIONS

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. In both cases, the error status is returned. There are four error-handling options that can be individually selected:

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).

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).

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 either 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_BLOCK_CHECK
MEM_BLOCK_ERROR_LOG_FLAG
MEM_BLOCK_ERROR_SUSPEND_FLAG

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

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

INCLUDE FILES **memLib.h**

SEE ALSO **memPartLib**, **smMemLib**

memPartLib

NAME

memPartLib – core memory partition manager

ROUTINES

memPartCreate( ) - create a memory partition
memPartAddToPool( ) - add memory to a memory partition
memPartAlignedAlloc( ) - allocate aligned memory from a partition
memPartAlloc( ) - allocate a block of memory from a partition
memPartFree( ) - free a block of memory in a partition
memAddToPool( ) - add memory to the system memory partition
malloc( ) - allocate a block of memory from the system memory partition (ANSI)
free( ) - free a block of memory (ANSI)

DESCRIPTION

This library provides core facilities for managing the allocation of blocks of memory from ranges of memory called memory partitions. The library was designed to provide a compact implementation; full-featured functionality is available with **memLib**, which provides enhanced memory management features built as an extension of **memPartLib**. (For more information about enhanced memory partition management options, see the manual entry for **memLib**.) This library consists of two sets of routines. The first set, **memPart...( )**, comprises a general facility for the creation and management of memory partitions, and for the allocation and deallocation of blocks from those partitions. The second set provides a traditional ANSI-compatible **malloc( )**/**free( )** interface to the system memory partition.

The system memory partition is created when the kernel is initialized by **kernelInit( )**, which is called by the root task, **usrRoot( )**, in **usrConfig.c**. The ID of the system memory partition is stored in the global variable **memSysPartId**; its declaration is included in **memLib.h**.

The allocation of memory, using **malloc( )** in the typical case and **memPartAlloc( )** for a specific memory partition, is done with a first-fit algorithm. Adjacent blocks of memory are coalesced when they are freed with **memPartFree( )** and **free( )**. There is also a routine provided for allocating memory aligned to a specified boundary from a specific memory partition, **memPartAlignedAlloc( )**.

CAVEATS

Architectures have various alignment constraints. To provide optimal performance, **malloc( )** returns a pointer to a buffer having the appropriate alignment for the architecture in use. The portion of the allocated buffer reserved for system bookkeeping, known as the overhead, may vary depending on the architecture.

Architecture	Boundary	Overhead
ARM	4	8
COLDFIRE	4	8
I86	4	8

Architecture	Boundary	Overhead
M68K	4	8
MCORE	8	8
MIPS	16	16
PPC *	8/16	8/16
SH	4	8
SimNT	8	8
SimSolaris	8	8

* On PowerPC, the boundary and 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 and overhead are 8 bytes.

INCLUDE FILES **memLib.h**, **stdlib.h**

SEE ALSO **memLib**, **smMemLib**

M

memShow

NAME **memShow** – memory show routines

ROUTINES
memShowInit( ) - initialize the memory partition show facility
memShow( ) - show system memory partition blocks and statistics
memPartShow( ) - show partition blocks and statistics
memPartInfoGet( ) - get partition information

DESCRIPTION This library contains memory partition information display routines. To use this facility, it must first be installed using **memShowInit( )**, which is called automatically when the memory partition show facility is configured into VxWorks using either of the following methods:

If you use the configuration header files, define **INCLUDE_SHOW_ROUTINES** in **config.h**.

If you use the Tornado project facility, select **INCLUDE_MEM_SHOW**.

SEE ALSO **memLib**, **memPartLib**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

mmanPxLib

NAME
mmanPxLib – memory management library (POSIX)

ROUTINES
mlockall( ) - lock all pages used by a process into memory (POSIX)
munlockall( ) - unlock all pages used by a process (POSIX)
mlock( ) - lock specified pages into memory (POSIX)
munlock( ) - unlock specified pages (POSIX)

DESCRIPTION
This library contains POSIX interfaces designed to lock and unlock memory pages, *i.e.*, to control whether those pages may be swapped to secondary storage. Since VxWorks does not use swapping (all pages are always kept in memory), these routines have no real effect and simply return 0 (**OK**).

INCLUDE FILES
sys/mman.h

SEE ALSO
POSIX 1003.1b document

mmuMapLib

NAME
mmuMapLib – MMU mapping library for ARM Ltd. processors

ROUTINES
mmuVirtToPhys( ) - translate a virtual address to a physical address (ARM)
mmuPhysToVirt( ) - translate a physical address to a virtual address (ARM)

DESCRIPTION
This library provides additional MMU support routines. These are present in a separate module from **mmuLib.c**, so that these routines can be used without including all the code in that object module.

mmuPro32Lib

NAME **mmuPro32Lib** – MMU library for PentiumPro/2/3/4 32 bit mode

ROUTINES **mmuPro32LibInit( )** - initialize module

DESCRIPTION **mmuPro32Lib.c** provides the architecture dependent routines that directly control the memory management unit. It provides 10 routines that are called by the higher level architecture independent routines in **vmLib.c**:

mmuPro32LibInit( ) - initialize module
mmuTransTblCreate( ) - create a new translation table
mmuTransTblDelete( ) - delete a translation table.
mmuPro32Enable( ) - turn MMU on or off
mmuStateSet( ) - set state of virtual memory page
mmuStateGet( ) - get state of virtual memory page
mmuPageMap( ) - map physical memory page to virtual memory page
mmuGlobalPageMap( ) - map physical memory page to global virtual memory page
mmuTranslate( ) - translate a virtual address to a physical address
mmuCurrentSet( ) - change active translation table

Applications using the MMU will never call these routines directly; the visible interface is supported in **vmLib.c**.

mmuLib supports the creation and maintenance of multiple translation tables, one of which is the active translation table when the MMU is enabled. Note that VxWorks does not include a translation table as part of the task context; individual tasks do not reside in private virtual memory. However, we include the facilities to create multiple translation tables so that the user may create "private" virtual memory contexts and switch them in an application specific manner. New translation tables are created with a call to **mmuTransTblCreate( )**, and installed as the active translation table with **mmuCurrentSet( )**. Translation tables are modified and potentially augmented with calls to **mmuPageMap( )** and **mmuStateSet( )**. The state of portions of the translation table can be read with calls to **mmuStateGet( )** and **mmuTranslate( )**.

The traditional VxWorks architecture and design philosophy requires that all objects and operating systems resources be visible and accessible to all agents (tasks, isrs, watchdog timers, *etc*.) in the system. This has traditionally been insured by the fact that all objects and data structures reside in physical memory; thus, a data structure created by one agent may be accessed by any other agent using the same pointer (object identifiers in VxWorks are often pointers to data structures.) This creates a potential problem if you have multiple virtual memory contexts. For example, if a semaphore is created in one virtual memory context, you must guarantee that that semaphore will be visible in all virtual memory contexts if the semaphore is to be accessed at interrupt level, when a virtual memory context other than the one in which it was created may be active. Another example is that

M

code loaded using the incremental loader from the shell must be accessible in all virtual memory contexts, since code is shared by all agents in the system.

This problem is resolved by maintaining a global "transparent" mapping of virtual to physical memory for all the contiguous segments of physical memory (on board memory, i/o space, sections of vme space, *etc*.) that is shared by all translation tables; all available physical memory appears at the same address in virtual memory in all virtual memory contexts. This technique provides an environment that allows resources that rely on a globally accessible physical address to run without modification in a system with multiple virtual memory contexts.

An additional requirement is that modifications made to the state of global virtual memory in one translation table appear in all translation tables. For example, memory containing the text segment is made read only (to avoid accidental corruption) by setting the appropriate writable bits in the translation table entries corresponding to the virtual memory containing the text segment. This state information must be shared by all virtual memory contexts, so that no matter what translation table is active, the text segment is protected from corruption. The mechanism that implements this feature is architecture dependent, but usually entails building a section of a translation table that corresponds to the global memory, that is shared by all other translation tables. Thus, when changes to the state of the global memory are made in one translation table, the changes are reflected in all other translation tables.

mmuLib provides a separate call for constructing global virtual memory - **mmuGlobalPageMap( )** - which creates translation table entries that are shared by all translation tables. Initialization code in **usrConfig** makes calls to **vmGlobalMap( )** (which in turn calls **mmuGlobalPageMap( )**) to set up global transparent virtual memory for all available physical memory. All calls made to **mmuGlobalPageMap( )** must occur before any virtual memory contexts are created; changes made to global virtual memory after virtual memory contexts are created are not guaranteed to be reflected in all virtual memory contexts.

Most MMU architectures will dedicate some fixed amount of virtual memory to a minimal section of the translation table (a "segment", or "block"). This creates a problem in that the user may map a small section of virtual memory into the global translation tables, and then attempt to use the virtual memory after this section as private virtual memory. The problem is that the translation table entries for this virtual memory are contained in the global translation tables, and are thus shared by all translation tables. This condition is detected by **vmMap( )**, and an error is returned, thus, the lower level routines in **mmuPro32Lib.c** (**mmuPageMap( )**, **mmuGlobalPageMap( )**) need not perform any error checking.

A global variable **mmuPageBlockSize** should be defined which is equal to the minimum virtual segment size. **mmuLib** must provide a routine **mmuGlobalInfoGet( )**, which returns a pointer to the **globalPageBlock[]** array. This provides the user with enough information to be able to allocate virtual memory space that does not conflict with the global memory space.

This module supports the PentiumPro/2/3/4 MMU:

```
                              PDBR
                               |
                               |
                 -------------------------------------
 top level       |pde   |pde   |pde   |pde   |pde   |pde   | ...
                 -------------------------------------
                    |      |      |      |      |      |
                    |      |      |      |      |      |
      ----------           |      v      v      v      v
      |           ------        NULL   NULL   NULL   NULL
      |         |
      v         v
     ----      ----
l   |pte |    |pte |
o    ----      ----
w   |pte |    |pte |
e    ----      ----
r   |pte |    |pte |
l    ----      ----
e   |pte |    |pte |
v    ----      ----
e     .         .
l     .         .
      .         .
```

where the top level consists of an array of pointers (Page Directory Entry) held within a single 4k page. These point to arrays of Page Table Entry arrays in the lower level. Each of these lower level arrays is also held within a single 4k page, and describes a virtual space of 4 MB (each Page Table Entry is 4 bytes, so we get 1000 of these in each array, and each Page Table Entry maps a 4KB page - thus 1000 * 4096 = 4MB.)

To implement global virtual memory, a separate translation table called **mmuGlobalTransTbl** is created when the module is initialized. Calls to **mmuGlobalPageMap( )** will augment and modify this translation table. When new translation tables are created, memory for the top level array of sftd's is allocated and initialized by duplicating the pointers in **mmuGlobalTransTbl**'s top level sftd array. Thus, the new translation table will use the global translation table's state information for portions of virtual memory that are defined as global. Here's a picture to illustrate:

```
                  GLOBAL TRANS TBL                            NEW TRANS TBL
                        PDBR                                       PDBR
                         |                                          |
                         |                                          |
            -------------------------                  -------------------------
 top level  |pde   |pde   | NULL| NULL|                |pde   |pde   | NULL| NULL|
            -------------------------                  -------------------------
               |     |     |     |                        |     |     |     |
               |     |     |     |                        |     |     |     |
      ----------     |     v     v               ----------     |     v     v
      |        ------     NULL  NULL             |              |    NULL  NULL
      |        |                                 |              |
      o-------------------------------------------              |
      |        |                                                |
      |        o-------------------------------------------------
      |        |
      v        v
     ----     ----
l   |pte |   |pte |
o    ----     ----
w   |pte |   |pte |
e    ----     ----
r   |pte |   |pte |
l    ----     ----
e   |pte |   |pte |
v    ----     ----
e     .        .
l     .        .
      .        .
```

Note that with this scheme, the global memory granularity is 4MB. Each time you map a section of global virtual memory, you dedicate at least 4MB of the virtual space to global virtual memory that will be shared by all virtual memory contexts.

The physical memory that holds these data structures is obtained from the system memory manager via **memalign( )** to ensure that the memory is page aligned. We want to protect this memory from being corrupted, so we invalidate the descriptors that we set up in the global translation that correspond to the memory containing the translation table data structures. This creates a “chicken and the egg” paradox, in that the only way we can modify these data structures is through virtual memory that is now invalidated, and we can’t validate it because the page descriptors for that memory are in invalidated memory (confused yet?) So, you will notice that anywhere that page table descriptors (pte’s) are modified, we do so by locking out interrupts, momentarily disabling the MMU, accessing the memory with its physical address, enabling the MMU, and then re-enabling interrupts (see **mmuStateSet( )**, for example.)

Support for two new page attribute bits are added for PentiumPro's enhanced MMU. They are Global bit (G) and Page-level write-through/back bit (PWT). Global bit indicates a global page when set. When a page is marked global and the page global enable (PGE) bit in register CR4 is set, the page-table or page-directory entry for the page is not invalidated in the TLB when register CR3 is loaded or a task switch occurs. This bit is provided to prevent frequently used pages (such as pages that contain kernel or other operating system or executive code) from being flushed from the TLB. Page-level write-through/back bit (PWT) controls the write-through or write- back caching policy of individual pages or page tables. When the PWT bit is set, write-through caching is enabled for the associated page or page table. When the bit is clear, write-back caching is enabled for the associated page and page table. Following macros are used to describe these attribute bits in the physical memory descriptor table **sysPhysMemDesc[]** in **sysLib.c**.

VM_STATE_WBACK - use write-back cache policy for the page
VM_STATE_WBACK_NOT - use write-through cache policy for the page
VM_STATE_GLOBAL - set page global bit
VM_STATE_GLOBAL_NOT - not set page global bit

Support for two page size (4KB and 4MB) are added also. The linear address for 4KB pages is divided into three sections:

Page directory entry - bits 22 through 31.
Page table entry - Bits 12 through 21.
Page offset - Bits 0 through 11.

The linear address for 4MB pages is divided into two sections:

Page directory entry - Bits 22 through 31.
Page offset - Bits 0 through 21.

These two page size is configurable by **VM_PAGE_SIZE** macro in **config.h**.

mmuSh7700Lib

NAME **mmuSh7700Lib** – Hitachi SH7700 MMU support library

ROUTINES **mmuSh7700LibInit( )** - initialize module

DESCRIPTION **mmuLib.c** provides the architecture dependent routines that directly control the memory management unit. It provides 10 routines that are called by the higher level architecture independent routines in **vmLib.c**:

– **mmuLibInit( )** - initialize module

– **mmuTransTblCreate( )** - create a new translation table

– **mmuTransTblDelete( )** - delete a translation table.

– **mmuEnable( )** - turn mmu on or off

– **mmuStateSet( )** - set state of virtual memory page

– **mmuStateGet( )** - get state of virtual memory page

– **mmuPageMap( )** - map physical memory page to virtual memory page

– **mmuGlobalPageMap( )** - map physical memory page to global virtual memory page

– **mmuTranslate( )** - translate a virtual address to a physical address

– **mmuCurrentSet( )** - change active translation table

Applications using the mmu will never call these routines directly; the visible interface is supported in **vmLib.c**.

mmuLib supports the creation and maintenance of multiple translation tables, one of which is the active translation table when the mmu is enabled. Note that VxWorks does not include a translation table as part of the task context; individual tasks do not reside in private virtual memory. However, we include the facilities to create multiple translation tables so that the user may create "private" virtual memory contexts and switch them in an application specific manner. New translation tables are created with a call to **mmuTransTblCreate( )**, and installed as the active translation table with **mmuCurrentSet( )**. Translation tables are modified and potentially augmented with calls to **mmuPageMap( )** and **mmuStateSet( )**. The state of portions of the translation table can be read with calls to **mmuStateGet( )** and **mmuTranslate( )**.

The traditional VxWorks architecture and design philosophy requires that all objects and operating systems resources be visible and accessible to all agents (tasks, isrs, watchdog timers, *etc.*) in the system. This has traditionally been insured by the fact that all objects and data structures reside in physical memory; thus, a data structure created by one agent may be accessed by any other agent using the same pointer (object identifiers in VxWorks are often pointers to data structures.) This creates a potential problem if you have multiple virtual memory contexts. For example, if a semaphore is created in one virtual memory context, you must guarantee that that semaphore will be visible in all virtual memory contexts if the semaphore is to be accessed at interrupt level, when a virtual memory context other than the one in which it was created may be active. Another example is that code loaded using the incremental loader from the shell must be accessible in all virtual memory contexts, since code is shared by all agents in the system.

This problem is resolved by maintaining a global "transparent" mapping of virtual to physical memory for all the contiguous segments of physical memory (on board memory, i/o space, sections of vme space, *etc*.) that is shared by all translation tables; all available physical memory appears at the same address in virtual memory in all virtual memory contexts. This technique provides an environment that allows resources that rely on a globally accessible physical address to run without modification in a system with multiple virtual memory contexts.

An additional requirement is that modifications made to the state of global virtual memory in one translation table appear in all translation tables. For example, memory containing the text segment is made read only (to avoid accidental corruption) by setting the appropriate writable bits in the translation table entries corresponding to the virtual memory containing the text segment. This state information must be shared by all virtual memory contexts, so that no matter what translation table is active, the text segment is protected from corruption. The mechanism that implements this feature is architecture dependent, but usually entails building a section of a translation table that corresponds to the global memory, that is shared by all other translation tables. Thus, when changes to the state of the global memory are made in one translation table, the changes are reflected in all other translation tables.

mmuLib provides a separate call for constructing global virtual memory - **mmuGlobalPageMap( )** - which creates translation table entries that are shared by all translation tables. Initialization code in **usrConfig** makes calls to **vmGlobalMap( )** (which in turn calls **mmuGlobalPageMap( )**) to set up global transparent virtual memory for all available physical memory. All calls made to **mmuGlobaPageMap( )** must occur before any virtual memory contexts are created; changes made to global virtual memory after virtual memory contexts are created are not guaranteed to be reflected in all virtual memory contexts.

Most mmu architectures will dedicate some fixed amount of virtual memory to a minimal section of the translation table (a "segment", or "block"). This creates a problem in that the user may map a small section of virtual memory into the global translation tables, and then attempt to use the virtual memory after this section as private virtual memory. The problem is that the translation table entries for this virtual memory are contained in the global translation tables, and are thus shared by all translation tables. This condition is detected by **vmMap( )**, and an error is returned, thus, the lower level routines in **mmuLib.c** (**mmuPageMap( )**, **mmuGlobalPageMap( )**) need not perform any error checking.

A global variable called **mmuPageBlockSize** should be defined which is equal to the minimum virtual segment size.

This module supports the SH7700 mmu with a two level translation table:

```
                            root
                             |
                             |
            --------------------------------
 top level  | td | td | td | td | td | td | ...
            --------------------------------
               |    |    |    |    |    |
               |    |    |    |    |    |
      ----------    |    v    v    v    v
      |         -----   NULL NULL NULL NULL
      |         |
      v         v
     ------     ------
l   | ptel |   | ptel |
o    ------     ------
w   | ptel |   | ptel |
e    ------     ------
r   | ptel |   | ptel |
l    ------     ------
e   | ptel |   | ptel |
v    ------     ------
e     .         .
l     .         .
      .         .
```

where the top level consists of an array of pointers (Table Descriptors) held within a single 4k page. These point to arrays of PTEL (Page Table Entry Low) arrays in the lower level. Each of these lower level arrays is also held within a single 4k page, and describes a virtual space of 4MB (each page descriptor is 4 bytes, so we get 1024 of these in each array, and each page descriptor maps a 4KB page - thus 1024 * 4096 = 4MB.)

To implement global virtual memory, a separate translation table called **mmuGlobalTransTbl** is created when the module is initialized. Calls to **mmuGlobalPageMap( )** will augment and modify this translation table. When new translation tables are created, memory for the top level array of td's is allocated and initialized by duplicating the pointers in **mmuGlobalTransTbl**'s top level td array. Thus, the new translation table will use the global translation table's state information for portions of virtual memory that are defined as global. Here's a picture to illustrate:

```
                 GLOBAL TRANS TBL                     NEW TRANS TBL
                       root                                root
                        |                                   |
                        |                                   |
            -------------------------           -------------------------
 top level  | td1 | td2 | NULL| NULL|           | td1 | td2 | NULL| NULL|
            -------------------------           -------------------------
               |     |     |     |                 |     |     |     |
               |     |     |     |                 |     |     |     |
      ----------     |     v     v        ----------     |     v     v
      |         ------    NULL  NULL      |              |    NULL  NULL
      |         |                         |              |
      o------------------------------------              |
      |         |                                        |
      |         o-----------------------------------------
      |         |
      v         v
     ------     ------
l   | ptel |   | ptel |
o    ------     ------
w   | ptel |   | ptel |
e    ------     ------
r   | ptel |   | ptel |
l    ------     ------
e   | ptel |   | ptel |
v    ------     ------
e     .         .
l     .         .
      .         .
```

Note that with this scheme, the global memory granularity is 4MB. Each time you map a section of global virtual memory, you dedicate at least 4MB of the virtual space to global virtual memory that will be shared by all virtual memory contexts.

The physical memory that holds these data structures is obtained from the system memory manager via **memalign( )** to ensure that the memory is page aligned. We want to protect this memory from being corrupted, so we invalidate the descriptors that we set up in the global translation that correspond to the memory containing the translation table data structures. This creates a “chicken and the egg” paradox, in that the only way we can modify these data structures is through virtual memory that is now invalidated, and we can’t validate it because the page descriptors for that memory are in invalidated memory (confused yet?) So, you will notice that anywhere that page table descriptors (ptel’s) are modified, we do so by locking out interrupts, momentarily disabling the mmu, accessing the memory with its physical address, enabling the mmu, and then re-enabling interrupts (see **mmuStateSet( )**, for example.)

M

USER-MODIFIABLE OPTIONS

1) Memory fragmentation - **mmuLib** obtains memory from the system memory manager via **memalign( )** to contain the mmu's translation tables. This memory was allocated a page at a time on page boundaries. Unfortunately, in the current memory management scheme, the memory manager is not able to allocate these pages contiguously. Building large translation tables (*i.e.*, when mapping large portions of virtual memory) causes excessive fragmentation of the system memory pool. An attempt to alleviate this has been installed by providing a local buffer of page aligned memory; the user may control the buffer size by manipulating the global variable **mmuNumPagesInFreeList**. By default, **mmuPagesInFreeList** is set to 8.

2) Alternate memory source - A customer has special purpose hardware that includes separate static RAM for the mmu's translation tables. Thus, they require the ability to specify an alternate source of memory other than **memalign( )**. A global variable has been created that points to the memory partition to be used as the source for translation table memory; by default, it points to the system memory partition. The user may modify this to point to another memory partition before **mmuSh7700LibInit( )** is called.

mmuSh7750Lib

NAME

mmuSh7750Lib – Hitachi SH7750 MMU support library

ROUTINES

mmuSh7750LibInit( ) - initialize module

DESCRIPTION

mmuLib.c provides the architecture dependent routines that directly control the memory management unit. It provides 10 routines that are called by the higher level architecture independent routines in **vmLib.c**:

– **mmuLibInit( )** - initialize module
– **mmuTransTblCreate( )** - create a new translation table
– **mmuTransTblDelete( )** - delete a translation table.
– **mmuEnable( )** - turn mmu on or off
– **mmuStateSet( )** - set state of virtual memory page
– **mmuStateGet( )** - get state of virtual memory page
– **mmuPageMap( )** - map physical memory page to virtual memory page
– **mmuGlobalPageMap( )** - map physical memory page to global virtual memory page
– **mmuTranslate( )** - translate a virtual address to a physical address
– **mmuCurrentSet( )** - change active translation table

Applications using the mmu will never call these routines directly; the visible interface is supported in **vmLib.c**.

mmuLib supports the creation and maintenance of multiple translation tables, one of which is the active translation table when the mmu is enabled. Note that VxWorks does not include a translation table as part of the task context; individual tasks do not reside in private virtual memory. However, we include the facilities to create multiple translation tables so that the user may create "private" virtual memory contexts and switch them in an application specific manner. New translation tables are created with a call to **mmuTransTblCreate( )**, and installed as the active translation table with **mmuCurrentSet( )**. Translation tables are modified and potentially augmented with calls to **mmuPageMap( )** and **mmuStateSet( )**. The state of portions of the translation table can be read with calls to **mmuStateGet( )** and **mmuTranslate( )**.

The traditional VxWorks architecture and design philosophy requires that all objects and operating systems resources be visible and accessible to all agents (tasks, isrs, watchdog timers, *etc*.) in the system. This has traditionally been insured by the fact that all objects and data structures reside in physical memory; thus, a data structure created by one agent may be accessed by any other agent using the same pointer (object identifiers in VxWorks are often pointers to data structures.) This creates a potential problem if you have multiple virtual memory contexts. For example, if a semaphore is created in one virtual memory context, you must guarantee that that semaphore will be visible in all virtual memory contexts if the semaphore is to be accessed at interrupt level, when a virtual memory context other than the one in which it was created may be active. Another example is that code loaded using the incremental loader from the shell must be accessible in all virtual memory contexts, since code is shared by all agents in the system.

This problem is resolved by maintaining a global "transparent" mapping of virtual to physical memory for all the contiguous segments of physical memory (on board memory, i/o space, sections of vme space, *etc*.) that is shared by all translation tables; all available physical memory appears at the same address in virtual memory in all virtual memory contexts. This technique provides an environment that allows resources that rely on a globally accessible physical address to run without modification in a system with multiple virtual memory contexts.

An additional requirement is that modifications made to the state of global virtual memory in one translation table appear in all translation tables. For example, memory containing the text segment is made read only (to avoid accidental corruption) by setting the appropriate writable bits in the translation table entries corresponding to the virtual memory containing the text segment. This state information must be shared by all virtual memory contexts, so that no matter what translation table is active, the text segment is protected from corruption. The mechanism that implements this feature is architecture dependent, but usually entails building a section of a translation table that corresponds to the global memory, that is shared by all other translation tables. Thus, when changes to the state of the global memory are made in one translation table, the changes are reflected in all other translation tables.

mmuLib provides a separate call for constructing global virtual memory - **mmuGlobalPageMap( )** - which creates translation table entries that are shared by all translation tables. Initialization code in **usrConfig** makes calls to **vmGlobalMap( )** (which in turn calls **mmuGlobalPageMap( )**) to set up global transparent virtual memory for all available physical memory. All calls made to **mmuGlobaPageMap( )** must occur before any virtual memory contexts are created; changes made to global virtual memory after virtual memory contexts are created are not guaranteed to be reflected in all virtual memory contexts.

Most mmu architectures will dedicate some fixed amount of virtual memory to a minimal section of the translation table (a "segment", or "block"). This creates a problem in that the user may map a small section of virtual memory into the global translation tables, and then attempt to use the virtual memory after this section as private virtual memory. The problem is that the translation table entries for this virtual memory are contained in the global translation tables, and are thus shared by all translation tables. This condition is detected by **vmMap( )**, and an error is returned, thus, the lower level routines in **mmuLib.c** (**mmuPageMap( )**, **mmuGlobalPageMap( )**) need not perform any error checking.

A global variable called **mmuPageBlockSize** should be defined which is equal to the minimum virtual segment size.

This module supports the SH7750 mmu with a two level translation table:

```
                           root
                            |
                            |
           --------------------------------
 top level | td | td | td | td | td | td | ...
           --------------------------------
              |    |    |    |    |    |
              |    |    |    |    |    |
     ----------    |    v    v    v    v
     |        -----   NULL NULL NULL NULL
     |        |
     v        v
    ------    ------
l  | ptel |  | ptel |
o   ------    ------
w  | ptel |  | ptel |
e   ------    ------
r  | ptel |  | ptel |
l   ------    ------
e  | ptel |  | ptel |
v   ------    ------
e    .        .
l    .        .
     .        .
```

where the top level consists of an array of pointers (Table Descriptors) held within a single 4k page. These point to arrays of PTEL (Page Table Entry Low) arrays in the lower level.

Each of these lower level arrays is also held within a single 4k page, and describes a virtual space of 4MB (each page descriptor is 4 bytes, so we get 1024 of these in each array, and each page descriptor maps a 4KB page - thus 1024 * 4096 = 4MB.)

To implement global virtual memory, a separate translation table called **mmuGlobalTransTbl** is created when the module is initialized. Calls to **mmuGlobalPageMap( )** will augment and modify this translation table. When new translation tables are created, memory for the top level array of td's is allocated and initialized by duplicating the pointers in **mmuGlobalTransTbl**'s top level td array. Thus, the new translation table will use the global translation table's state information for portions of virtual memory that are defined as global. Here's a picture to illustrate:

```
                 GLOBAL TRANS TBL                     NEW TRANS TBL
                       root                                root
                        |                                   |
                        |                                   |
            -------------------------           -------------------------
 top level  | td1 | td2 | NULL| NULL|           | td1 | td2 | NULL| NULL|
            -------------------------           -------------------------
               |     |     |     |                 |     |     |     |
               |     |     |     |                 |     |     |     |
      ----------     |     v     v        ----------     |     v     v
      |         ------    NULL  NULL      |              |    NULL  NULL
      |         |                         |              |
      o------------------------------------              |
      |         |                                        |
      |         o-----------------------------------------
      |         |
      v         v
     ------     ------
l   | ptel |   | ptel |
o    ------     ------
w   | ptel |   | ptel |
e    ------     ------
r   | ptel |   | ptel |
l    ------     ------
e   | ptel |   | ptel |
v    ------     ------
e     .         .
l     .         .
      .         .
```

Note that with this scheme, the global memory granularity is 4MB. Each time you map a section of global virtual memory, you dedicate at least 4MB of the virtual space to global virtual memory that will be shared by all virtual memory contexts.

The physical memory that holds these data structures is obtained from the system memory manager via **memalign( )** to ensure that the memory is page aligned. We want to protect this memory from being corrupted, so we invalidate the descriptors that we set up in the global translation that correspond to the memory containing the translation table data structures. This creates a "chicken and the egg" paradox, in that the only way we can modify these data structures is through virtual memory that is now invalidated, and we can't validate it because the page descriptors for that memory are in invalidated memory (confused yet?) So, you will notice that anywhere that page table descriptors (ptel's) are modified, we do so by locking out interrupts, momentarily disabling the mmu, accessing the memory with its physical address, enabling the mmu, and then re-enabling interrupts (see **mmuStateSet( )**, for example.)

USER MODIFIABLE OPTIONS

1) Memory fragmentation - **mmuLib** obtains memory from the system memory manager via **memalign( )** to contain the mmu's translation tables. This memory was allocated a page at a time on page boundaries. Unfortunately, in the current memory management scheme, the memory manager is not able to allocate these pages contiguously. Building large translation tables (*i.e.*, when mapping large portions of virtual memory) causes excessive fragmentation of the system memory pool. An attempt to alleviate this has been installed by providing a local buffer of page aligned memory; the user may control the buffer size by manipulating the global variable **mmuNumPagesInFreeList**. By default, **mmuPagesInFreeList** is set to 8.

2) Alternate memory source - A customer has special purpose hardware that includes separate static RAM for the mmu's translation tables. Thus, they require the ability to specify an alternate source of memory other than **memalign( )**. A global variable has been created that points to the memory partition to be used as the source for translation table memory; by default, it points to the system memory partition. The user may modify this to point to another memory partition before **mmuSh7750LibInit( )** is called.

moduleLib

NAME

moduleLib – object module management library

ROUTINES

moduleCreate( ) - create and initialize a module
moduleDelete( ) - delete module ID information (use **unld( )** to reclaim space)
moduleShow( ) - show the current status for all the loaded modules
moduleSegGet( ) - get (delete and return) the first segment from a module
moduleSegFirst( ) - find the first segment in a module
moduleSegNext( ) - find the next segment in a module
moduleCreateHookAdd( ) - add a routine to be called when a module is added

moduleCreateHookDelete( ) - delete a previously added module create hook routine
moduleFindByName( ) - find a module by name
moduleFindByNameAndPath( ) - find a module by file name and path
moduleFindByGroup( ) - find a module by group number
moduleIdListGet( ) - get a list of loaded modules
moduleInfoGet( ) - get information about an object module
moduleCheck( ) - verify checksums on all modules
moduleNameGet( ) - get the name associated with a module ID
moduleFlagsGet( ) - get the flags associated with a module ID

DESCRIPTION

This library is a class manager, using the standard VxWorks class/object facilities. The library is used to keep track of which object modules have been loaded into VxWorks, to maintain information about object module segments associated with each module, and to track which symbols belong to which module. Tracking modules makes it possible to list which modules are currently loaded, and to unload them when they are no longer needed.

The module object contains the following information:

- name
- linked list of segments, including base addresses and sizes
- symbol group number
- format of the object module (a.out, COFF, ECOFF, *etc*.)
- the *symFlag* passed to **ld( )** when the module was loaded. (For more information about *symFlag* and the loader, see the manual entry for **loadLib**.)

Multiple modules with the same name are allowed (the same module may be loaded without first being unloaded) but "find" functions find the most recently created module.

The symbol group number is a unique number for each module, used to identify the module's symbols in the symbol table. This number is assigned by **moduleLib** when a module is created.

In general, users will not access these routines directly, with the exception of **moduleShow( )**, which displays information about currently loaded modules. Most calls to this library will be from routines in **loadLib** and **unldLib**.

INCLUDE FILES

moduleLib.h

SEE ALSO

loadLib, *Tornado User's Guide: Cross-Development*

mountLib

NAME
mountLib – mount protocol library

ROUTINES
mountdInit( ) - initialize the mount daemon
nfsExport( ) - specify a file system to be NFS exported
nfsUnexport( ) - remove a file system from the list of exported file systems

DESCRIPTION
This library implements a mount server to support mounting VxWorks file systems remotely. The mount server is an implementation of version 1 of the mount protocol as defined in RFC 1094. It is closely connected with version 2 of the Network File System Protocol Specification, which in turn is implemented by the library **nfsdLib**.

NOTE: The only routines in this library that are normally called by applications are **nfsExport( )** and **nfsUnexport( )**. The mount daemon is normally initialized indirectly by **nfsdInit( )**.

The mount server is initialized by calling **mountdInit( )**. Normally, this is done by **nfsdInit( )**, although it is possible to call **mountdInit( )** directly if the NFS server is not being initialized. Defining **INCLUDE_NFS_SERVER** enables the call to **nfsdInit( )** during the boot process, which in turn calls **mountdInit( )**, so there is normally no need to call either routine manually. **mountdInit( )** spawns one task, **tMountd**, which registers as an RPC service with the portmapper.

Currently, only the **dosFsLib** file system is supported. File systems are exported with the **nfsExport( )** call.

To export VxWorks file systems via NFS, you need facilities from both this library and from **nfsdLib**. To include both, add **INCLUDE_NFS_SERVER** and rebuild VxWorks.

Example

The following example illustrates how to export an existing dosFs file system.

First, initialize the block device containing your file system.

Then assuming the dosFs system is called **/export** execute the following code on the target:

```
nfsExport ("/export", 0, FALSE, 0);           /* make available remotely */
```

This makes it available to all clients to be mounted using the client's NFS mounting command. (On UNIX systems, mounting file systems normally requires root privileges.)

VxWorks does not normally provide authentication services for NFS requests, and the DOS file system does not provide file permissions. If you need to authenticate incoming requests, see the documentation for **nfsdInit( )** and **mountdInit( )** for information about authorization hooks.

The following requests are accepted from clients. For details of their use, see Appendix A of RFC 1094, "NFS: Network File System Protocol Specification."

Procedure Name	Procedure Number
MOUNTPROC_NULL	0
MOUNTPROC_MNT	1
MOUNTPROC_DUMP	2
MOUNTPROC_UMNT	3
MOUNTPROC_UMNTALL	4
MOUNTPROC_EXPORT	5

SEE ALSO **dosFsLib**, **nfsdLib**, RFC 1094

mqPxLib

NAME **mqPxLib** – message queue library (POSIX)

ROUTINES

mqPxLibInit( ) - initialize the POSIX message queue library
mq_open( ) - open a message queue (POSIX)
mq_receive( ) - receive a message from a message queue (POSIX)
mq_send( ) - send a message to a message queue (POSIX)
mq_close( ) - close a message queue (POSIX)
mq_unlink( ) - remove a message queue (POSIX)
mq_notify( ) - notify a task that a message is available on a queue (POSIX)
mq_setattr( ) - set message queue attributes (POSIX)
mq_getattr( ) - get message queue attributes (POSIX)

DESCRIPTION

This library implements the message-queue interface defined in the POSIX 1003.1b standard, as an alternative to the VxWorks-specific message queue design in **msgQLib**. These 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 be notified asynchronously of the availability of a message on the queue. The purpose of this feature is to let the task to perform other functions and yet still be notified that a message has become available on the queue.

M

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.

The routines in this library conform to POSIX 1003.1b.

INCLUDE FILES **mqueue.h**

SEE ALSO POSIX 1003.1b document, **msgQLib**, *VxWorks Programmer's Guide: Basic OS*

mqPxShow

NAME **mqPxShow** – POSIX message queue show

ROUTINES **mqPxShowInit( )** - initialize the POSIX message queue show facility

DESCRIPTION This library provides a show routine for POSIX objects.

msgQDistGrpLib

NAME **msgQDistGrpLib** – distributed message queue group library (VxFusion Opt.)

ROUTINES
msgQDistGrpAdd( ) - add a distributed message queue to a group (VxFusion Opt.)
msgQDistGrpDelete( ) - delete a distributed message queue from a group (VxFusion Opt.)

DESCRIPTION This library provides the grouping facility for distributed message queues. Single distributed message queues can join one or more groups. A message sent to a group is sent to all message queues that are members of that group. A group, however, is prohibited from sending messages. Also, it is an error to call **msgQDistNumMsgs( )** with a distributed message queue group ID.

Groups are created with symbolic names and identified by a unique ID, **MSG_Q_ID**, as with normal message queues.

If the group is new to the distributed system, the group agreement protocol (GAP) is employed to determine a globally unique identifier. As part of the protocol's negotiation, all group databases throughout the system are updated.

The distributed message queue group library is initialized by calling **distInit( )**.

AVAILABILITY
This module is distributed as a component of the unbundled distributed message queues option, VxFusion.

INCLUDE FILES
msgQDistGrpLib.h

SEE ALSO
distLib, **msgQDistGrpShow**

msgQDistGrpShow

NAME
msgQDistGrpShow – distributed message queue group show routines (VxFusion Opt.)

ROUTINES
msgQDistGrpShow( ) - display all or one group with its members (VxFusion Opt.)

DESCRIPTION
This library provides a routine to show either the contents of the entire message queue group database or the contents of single message queue group.

AVAILABILITY
This module is distributed as a component of the unbundled distributed message queues option, VxFusion.

INCLUDE FILES
msgQDistGrpShow.h

SEE ALSO
msgQDistGrpLib

msgQDistLib

NAME
msgQDistLib – distributed objects message queue library (VxFusion Opt.)

ROUTINES
msgQDistCreate( ) - create a distributed message queue (VxFusion Opt.)
msgQDistSend( ) - send a message to a distributed message queue (VxFusion Opt.)
msgQDistReceive( ) - receive a message from a distributed message queue (VxFusion Opt.)
msgQDistNumMsgs( ) - get the number of messages in a distributed message queue (VxFusion Opt.)

DESCRIPTION
This library provides the interface to distributed message queues. Any task on any node in the system can send messages to or receive from a distributed message queue. Full

M

duplex communication between two tasks generally requires two distributed message queues, one for each direction.

Distributed message queues are created with **msgQDistCreate( )**. After creation, they can be manipulated using the generic routines for local message queues; for more information on the use of these routines, see the manual entry for **msgQLib**. The **msgQDistLib** library also provides the **msgQDistSend( )**, **msgQDistReceive( )**, and **msgQDistNumMsgs( )** routines which support additional parameters that are useful for working with distributed message queues.

The distributed objects message queue library is initialized by calling **distInit( )**.

AVAILABILITY This module is distributed as a component of the unbundled distributed message queues option, VxFusion.

INCLUDE FILES **msgQDistLib.h**

SEE ALSO **msgQLib**, **msgQDistShow**, **distLib**

msgQDistShow

NAME **msgQDistShow** – distributed message queue show routines (VxFusion Opt.)

ROUTINES **msgQDistShowInit( )** - initialize the distributed message queue show package (VxFusion Opt.)

DESCRIPTION This library provides show routines for distributed message queues. The user does not call these show routines directly. Instead, he uses the **msgQShow** library routine **msgQShow( )** to display the contents of a message queue, regardless of its type. The **msgQShow( )** routine calls the distributed show routines, as necessary.

AVAILABILITY This module is distributed as a component of the unbundled distributed message queues option, VxFusion.

INCLUDE FILES **msgQDistShow.h**

SEE ALSO **msgQDistLib**, **msgQShow**

msgQEvLib

NAME

msgQEvLib – VxWorks events support for message queues

ROUTINES

msgQEvStart( ) - start event notification process for a message queue
msgQEvStop( ) - stop 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, *VxWorks Programmer's Guide: Basic OS*

msgQLib

NAME

msgQLib – message queue library

ROUTINES

msgQCreate( ) - create and initialize a message queue
msgQDelete( ) - delete a message queue
msgQSend( ) - send a message to a message queue
msgQReceive( ) - receive a message from a message queue
msgQNumMsgs( ) - get the number of messages queued to a message queue

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.

To provide message queue support for a system, VxWorks must be configured with the **INCLUDE_MSG_Q** component.

CREATING AND USING MESSAGE QUEUES

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

A task or interrupt service routine sends a message to a message queue with **msgQSend( )**. If no tasks are waiting for messages on the message queue, the message is simply 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 will block and be 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 should return immediately; **WAIT_FOREVER** means the routine should never time out.

URGENT MESSAGES

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

VXWORKS EVENTS If a task has registered with a message queue via **msgQEvStart( )**, events will be sent to that task when a message arrives on that message queue, on the condition that no other task is pending on the queue.

INCLUDE FILES **msgQLib.h**

SEE ALSO **pipeDrv**, **msgQSmLib**, **msgQEvLib**, **eventLib**, *VxWorks Programmer's Guide: Basic OS*

msgQShow

NAME **msgQShow** – message queue show routines

ROUTINES **msgQShowInit( )** - initialize the message queue show facility
msgQInfoGet( ) - get information about a message queue
msgQShow( ) - show information about a message queue

DESCRIPTION This library provides routines to show message queue statistics, such as the task queuing method, messages queued, receivers blocked, *etc*.

The routine **msgQshowInit( )** links the message queue show facility into the VxWorks system. It is called automatically when the message queue show facility is configured into VxWorks using either of the following methods:

If you use the configuration header files, define **INCLUDE_SHOW_ROUTINES** in **config.h**.

If you use the Tornado project facility, select **INCLUDE_MSG_Q_SHOW**.

INCLUDE FILES **msgQLib.h**

SEE ALSO **pipeDrv**, *VxWorks Programmer's Guide: Basic OS*

msgQSmLib

NAME **msgQSmLib** – shared memory message queue library (VxMP Opt.)

ROUTINES **msgQSmCreate( )** - create and initialize a shared memory message queue (VxMP Opt.)

DESCRIPTION This library provides the interface to shared memory message queues. Shared memory message queues allow a variable number of messages (varying in length) to be queued in first-in-first-out order. Any task running on any CPU in the system can send messages to or receive messages from a shared message queue. Tasks can also send to and receive from the same shared message queue. Full-duplex communication between two tasks generally requires two shared message queues, one for each direction.

Shared memory message queues are created with **msgQSmCreate( )**. Once created, they can be manipulated using the generic routines for local message queues; for more information on the use of these routines, see the manual entry for **msgQLib**.

MEMORY REQUIREMENTS

The shared memory message queue structure is allocated from a dedicated shared memory partition. This shared memory partition is initialized by the shared memory objects master CPU. The size of this partition is defined by the maximum number of shared message queues, **SM_OBJ_MAX_MSG_Q**.

The message queue buffers are allocated from the shared memory system partition.

RESTRICTIONS Shared memory message queues differ from local message queues in the following ways:

Interrupt Use:
Shared memory message queues may not be used (sent to or received from) at interrupt level.

Deletion:
There is no way to delete a shared memory message queue and free its associated shared memory. Attempts to delete a shared message queue return **ERROR** and set **errno** to **S_smObjLib_NO_OBJECT_DESTROY**.

Queuing Style:
The shared message queue task queueing order specified when a message queue is created must be FIFO.

CONFIGURATION Before routines in this library can be called, the shared memory objects facility must be initialized by calling **usrSmObjInit( )**. This is done automatically during VxWorks initialization if the component **INCLUDE_SM_OBJ** is included.

AVAILABILITY This module is distributed as a component of the unbundled shared objects memory support option, VxMP.

INCLUDE FILES **msgQSmLib.h**, **msgQLib.h**, **smMemLib.h**, **smObjLib.h**

SEE ALSO **msgQLib**, **smObjLib**, **msgQShow**, **usrSmObjInit( )**, *VxWorks Programmer's Guide: Shared Memory Objects*

muxLib

NAME **muxLib** – MUX network interface library

ROUTINES
muxLibInit( ) - initialize global state for the MUX
muxDevLoad( ) - load a driver into the MUX
muxDevStart( ) - start a device by calling its start routine
muxDevStop( ) - stop a device by calling its stop routine
muxShow( ) - display configuration of devices registered with the MUX
muxBind( ) - create a binding between a network service and an END
muxSend( ) - send a packet out on a network interface

muxPollSend( ) - now **deprecated**, see **muxTkPollSend( )**
muxPollReceive( ) - now **deprecated**, see **muxTkPollReceive( )**
muxIoctl( ) - send control information to the MUX or to a device
muxMCastAddrAdd( ) - add a multicast address to a device's multicast table
muxMCastAddrDel( ) - delete a multicast address from a device's multicast table
muxMCastAddrGet( ) - get the multicast address table from the MUX/Driver
muxUnbind( ) - detach a network service from the specified device
muxDevUnload( ) - unloads a device from the MUX
muxLinkHeaderCreate( ) - attach a link-level header to a packet
muxAddressForm( ) - form a frame with a link-layer address
muxPacketDataGet( ) - return the data from a packet
muxPacketAddrGet( ) - get addressing information from a packet
endFindByName( ) - find a device using its string name
muxDevExists( ) - tests whether a device is already loaded into the MUX
muxAddrResFuncAdd( ) - replace the default address resolution function
muxAddrResFuncGet( ) - get the address resolution function for ifType/protocol
muxAddrResFuncDel( ) - delete an address resolution function
muxTaskDelaySet( ) - set the inter-cycle delay on the polling task
muxTaskDelayGet( ) - get the delay on the polling task
muxTaskPrioritySet( ) - reset the priority of **tMuxPollTask**
muxTaskPriorityGet( ) - get the priority of **tMuxPollTask**
muxPollStart( ) - initialize and start the MUX poll task
muxPollEnd( ) - shuts down **tMuxPollTask** and returns devices to interrupt mode
muxPollDevAdd( ) - adds a device to list polled by **tMuxPollTask**
muxPollDevDel( ) - removes a device from the list polled by **tMuxPollTask**
muxPollDevStat( ) - reports whether device is on list polled by **tMuxPollTask**

DESCRIPTION

This library provides the routines that define the MUX interface, a facility that handles communication between the data link layer and the network protocol layer. Using the MUX, the VxWorks network stack has decoupled the data link and network layers. Drivers and services no longer need knowledge of each other's internals. This independence makes it much easier to add new drivers or services. For example, if you add a new MUX-based "END" driver, all existing MUX-based services can use the new driver. Likewise, if you add a new MUX-based service, any existing END can use the MUX to access the new service.

INCLUDE FILES

errno.h, **lstLib.h**, **logLib.h**, **string.h**, **m2Lib.h**, **bufLib.h**, **if.h**, **end.h**, **muxLib.h**, **vxWorks.h**, **taskLib.h**, **stdio.h**, **errnoLib.h**, **if_ether.h**, **netLib.h**, **semLib.h**, **rebootLib.h**

To use this feature, include the following component: **INCLUDE_MUX**

SEE ALSO

VxWorks AE Network Programmer's Guide

M

muxTkLib

NAME

muxTkLib – MUX toolkit Network Interface Library

ROUTINES

muxTkDrvCheck( ) - checks if the device is an NPT or an END interface
muxTkCookieGet( ) - returns the cookie for a device
muxTkBind( ) - bind an NPT protocol to a driver
muxTkReceive( ) - receive a packet from a NPT driver
muxTkSend( ) - send a packet out on a Toolkit or END network interface
muxTkPollSend( ) - send a packet out in polled mode to an END or NPT interface
muxTkPollReceive( ) - poll for a packet from a NPT or END driver

DESCRIPTION

This library provides additional APIs offered by the Network Protocol Toolkit (NPT) architecture. These APIs extend the original release of the MUX interface.

A NPT driver is an enhanced END but retains all of the END's functionality. NPT also introduces the term "network service sublayer" or simply "service sublayer" which is the component that interfaces between the network service (or network protocol) and the MUX. This service sublayer may be built in to the network service or protocol rather than being a separate component.

INCLUDE FILES

vxWorks.h, **taskLib.h**, **stdio.h**, **errno.herrnoLib.h**, **lstlib.h**, **logLib.h**, **string.h**, **m2Lib.h**, **net/if.h**, **bufLib.h**, **semlib.h**, **end.h**, **muxLib.h**, **muxTkLib.h**, **netinet/if_ether.h**, **net/mbuf.h**

netBufLib

NAME

netBufLib – network buffer library

ROUTINES

netBufLibInit( ) - initialize **netBufLib**
netPoolInit( ) - initialize a **netBufLib**-managed memory pool
netPoolKheapInit( ) - kernel heap version of **netPoolInit( )**
netPoolDelete( ) - delete a memory pool
netMblkFree( ) - free an **mBlk** back to its memory pool
netClBlkFree( ) - free a **clBlk**-cluster construct back to the memory pool
netClFree( ) - free a cluster back to the memory pool
netMblkClFree( ) - free an **mBlk-clBlk**-cluster construct
netMblkClChainFree( ) - free a chain of **mBlk-clBlk**-cluster constructs
netMblkGet( ) - get an **mBlk** from a memory pool
netClBlkGet( ) - get a **clBlk**
netClusterGet( ) - get a cluster from the specified cluster pool
netMblkClGet( ) - get a **clBlk**-cluster and join it to the specified **mBlk**
netTupleGet( ) - get an **mBlk-clBlk**-cluster
netClBlkJoin( ) - join a cluster to a **clBlk** structure
netMblkClJoin( ) - join an **mBlk** to a **clBlk**-cluster construct
netClPoolIdGet( ) - return a **CL_POOL_ID** for a specified buffer size
netMblkToBufCopy( ) - copy data from an **mBlk** to a buffer
netMblkDup( ) - duplicate an **mBlk**
netMblkChainDup( ) - duplicate an **mBlk** chain

DESCRIPTION

This library contains routines that you can use to organize and maintain a memory pool that consists of pools of **mBlk** structures, pools of **clBlk** structures, and pools of clusters. The **mBlk** and **clBlk** structures are used to manage the clusters. The clusters are containers for the data described by the **mBlk** and **clBlk** structures.

These structures and the various routines of this library constitute a buffering API that has been designed to meet the needs both of network protocols and network device drivers.

The **mBlk** structure is the primary vehicle for passing data between a network driver and a protocol. However, the **mBlk** structure must first be properly joined with a **clBlk** structure that was previously joined with a cluster. Thus, the actual vehicle for passing data is not merely an **mBlk** structure but an **mBlk-clBlk**-cluster construct.

To use this feature, include the following component: **INCLUDE_NETWRS_NETBUFLIB**

INCLUDE FILES

netBufLib.h

N

netDrv

NAME

netDrv – network remote file I/O driver

ROUTINES

netDrv( ) - install the network remote file driver
netDevCreate( ) - create a remote file device
netDevCreate2( ) - create a remote file device with fixed buffer size
netDrvDebugLevelSet( ) - set the debug level of the **netDrv** library routines
netDrvFileDoesNotExistInstall( ) - install an applette to test if a file exists

DESCRIPTION

This driver provides facilities for accessing files transparently over the network via FTP or RSH. By creating a network device with **netDevCreate( )**, files on a remote UNIX machine may be accessed as if they were local.

When a remote file is opened, the entire file is copied over the network to a local buffer. When a remote file is created, an empty local buffer is opened. Any reads, writes, or **ioctl( )** calls are performed on the local copy of the file. If the file was opened with the flags **O_WRONLY** or **O_RDWR** and modified, the local copy is sent back over the network to the UNIX machine when the file is closed.

Note that this copying of the entire file back and forth can make **netDrv** devices awkward to use. A preferable mechanism is NFS as provided by **nfsDrv**.

USER-CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. However, two routines must be called directly: **netDrv( )** to initialize the driver and **netDevCreate( )** to create devices.

FILE OPERATIONS

This driver supports the creation, deletion, opening, reading, writing, and appending of files. The renaming of files is not supported.

INITIALIZATION

Before using the driver, it must be initialized by calling the routine **netDrv( )**. This routine should be called only once, before any reads, writes, **netDevCreate( )**, or **netDevCreate2( )** calls. Initialization is performed automatically when **INCLUDE_NET_DRV** is defined.

CREATING NETWORK DEVICES

To access files on a remote host, a network device must be created by calling **netDevCreate( )** or **netDevCreate2( )**. The arguments to **netDevCreate( )** are the name of the device, the name of the host the device will access, and the remote file access protocol to be used -- RSH or FTP. The arguments to **netDevCreate2( )** are ones described above and a size of buffer used in the network device as a fourth argument. By convention, a network device name is the remote machine name followed by a colon ":". For example, for a UNIX host on the network "wrs", files can be accessed by creating a device called "wrs:". For more information, see the manual entry for **netDevCreate( )** and **netDevCreate2( )**.

IOCTL FUNCTIONS

The network driver responds to the following **ioctl( )** functions:

FIOGETNAME
Gets the file name of the file descriptor *fd* and copies it to the buffer specified by *nameBuf*:

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

FIONREAD
Copies to *nBytesUnread* the number of bytes remaining in the file specified by *fd*:

```
status = ioctl (fd, FIONREAD, &nBytesUnread);
```

FIOSEEK
Sets the current byte offset in the file to the position specified by *newOffset*. If the seek goes beyond the end-of-file, the file grows. The end-of-file pointer changes to the new position, and the new space is filled with zeroes:

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

FIOWHERE
Returns the current byte position in the file. This is the byte offset of the next byte to be read or written. It takes no additional argument:

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

FIOFSTATGET
Gets file status information. The argument *statStruct* is a pointer to a stat structure that is filled with data describing the specified file. Normally, the **stat( )** or **fstat( )** routine is used to obtain file information, rather than using the **FIOFSTATGET** function directly. **netDrv** only fills in three fields of the stat structure: **st_dev**, **st_mode**, and **st_size**. **st_mode** is always filled with **S_IFREG**.

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

LIMITATIONS

The **netDrv** implementation strategy implies that directories cannot always be distinguished from plain files. Thus, **opendir( )** does not work for directories mounted on **netDrv** devices, and **ll( )** does not flag subdirectories with the label "DIR" in listings from **netDrv** devices.

When the access method is FTP, operations can only be done on files that the FTP server allows to download. In particular it is not possible to **stat** a directory, doing so will result in "*dirname*: not a plain file" error.

INCLUDE FILES

netDrv.h

SEE ALSO

remLib, **netLib**, **sockLib**, **hostAdd( )**

N

netLib

NAME
netLib – network interface library

ROUTINES
netLibInit( ) - initialize the network package
netTask( ) - network task entry point

DESCRIPTION
This library contains the network task that runs low-level network interface routines in a task context. The network task executes and removes routines that were added to the job queue. This facility is used by network interfaces in order to have interrupt-level processing at task level.

The routine **netLibInit( )** initializes the network and spawns the network task **netTask( )**. This is done automatically when **INCLUDE_NET_LIB** is defined.

The routine **netHelp( )** in **usrLib** displays a summary of the network facilities available from the VxWorks shell.

INCLUDE FILES
netLib.h

SEE ALSO
routeLib, **hostLib**, **netDrv**, **netHelp( )**,

netShow

NAME
netShow – network information display routines

ROUTINES
ifShow( ) - display the attached network interfaces
inetstatShow( ) - display all active connections for Internet protocol sockets
ipstatShow( ) - display IP statistics
netPoolShow( ) - show pool statistics
netStackDataPoolShow( ) - show network stack data pool statistics
netStackSysPoolShow( ) - show network stack system pool statistics
mbufShow( ) - report **mbuf** statistics
netShowInit( ) - initialize network show routines
arpShow( ) - display entries in the system ARP table
arptabShow( ) - display the known ARP entries
routestatShow( ) - display routing statistics
routeShow( ) - display all IP routes (summary information)
hostShow( ) - display the host table
mRouteShow( ) - display all IP routes (verbose information)

DESCRIPTION

This library provides routines to show various network-related statistics, such as configuration parameters for network interfaces, protocol statistics, socket statistics, and so on.

Interpreting these statistics requires detailed knowledge of Internet network protocols. Information on these protocols can be found in the following books:

Internetworking with TCP/IP Volume III, by Douglas Comer and David Stevens

UNIX Network Programming, by Richard Stevens

The Design and Implementation of the 4.3 BSD UNIX Operating System, by Leffler, McKusick, Karels and Quarterman

The **netShowInit( )** routine links the network show facility into the VxWorks system. This is performed automatically if **INCLUDE_NET_SHOW** is defined. If you want **inetstatShow( )** to display TCP socket status, then **INCLUDE_TCP_SHOW** needs to be included.

SEE ALSO

ifLib, **icmpShow**, **igmpShow**, **tcpShow**, **udpShow**

nfsdLib

N

NAME

nfsdLib – Network File System (NFS) server library

ROUTINES

nfsdInit( ) - initialize the NFS server
nfsdStatusGet( ) - get the status of the NFS server
nfsdStatusShow( ) - show the status of the NFS server

DESCRIPTION

This library is an implementation of version 2 of the Network File System Protocol Specification as defined in RFC 1094. It is closely connected with version 1 of the mount protocol, also defined in RFC 1094 and implemented in turn by **mountLib**.

The NFS server is initialized by calling **nfsdInit( )**. This is done automatically at boot time if **INCLUDE_NFS_SERVER** is defined.

Currently, only the **dosFsLib** file system is supported. File systems are exported with the **nfsExport( )** call.

To create and export a file system, define **INCLUDE_NFS_SERVER** and rebuild VxWorks.

To export VxWorks file systems via NFS, you need facilities from both this library and from **mountLib**. To include both, define **INCLUDE_NFS_SERVER** and rebuild VxWorks.

Use the **mountLib** routine **nfsExport( )** to export file systems. For an example, see the manual page for **mountLib**.

VxWorks does not normally provide authentication services for NFS requests, and the DOS file system does not provide file permissions. If you need to authenticate incoming requests, see the documentation for **nfsdInit( )** and **mountdInit( )** for information about authorization hooks.

The following requests are accepted from clients. For details of their use, see RFC 1094, "NFS: Network File System Protocol Specification."

Procedure Name	Procedure Number
NFSPROC_NULL	0
NFSPROC_GETATTR	1
NFSPROC_SETATTR	2
NFSPROC_ROOT	3
NFSPROC_LOOKUP	4
NFSPROC_READLINK	5
NFSPROC_READ	6
NFSPROC_WRITE	8
NFSPROC_CREATE	9
NFSPROC_REMOVE	10
NFSPROC_RENAME	11
NFSPROC_LINK	12
NFSPROC_SYMLINK	13
NFSPROC_MKDIR	14
NFSPROC_RMDIR	15
NFSPROC_READDIR	16
NFSPROC_STATFS	17

AUTHENTICATION AND PERMISSIONS

Currently, no authentication is done on NFS requests. **nfsdInit( )** describes the authentication hooks that can be added should authentication be necessary.

Note that the DOS file system does not provide information about ownership or permissions on individual files. Before initializing a dosFs file system, three global variables--**dosFsUserId**, **dosFsGroupId**, and **dosFsFileMode**--can be set to define the user ID, group ID, and permissions byte for all files in all dosFs volumes initialized after setting these variables. To arrange for different dosFs volumes to use different user and group ID numbers, reset these variables before each volume is initialized. See the manual entry for **dosFsLib** for more information.

TASKS

Several NFS tasks are created by **nfsdInit( )**. They are:

tMountd
: The mount daemon, which handles all incoming mount requests. This daemon is created by **mountdInit( )**, which is automatically called from **nfsdInit( )**.

tNfsd
The NFS daemon, which queues all incoming NFS requests.

tNfsdX
The NFS request handlers, which dequeues and processes all incoming NFS requests.

Performance of the NFS file system can be improved by increasing the number of servers specified in the **nfsdInit( )** call, if there are several different dosFs volumes exported from the same target system. The **spy( )** utility can be called to determine whether this is useful for a particular configuration.

nfsDrv

NAME

nfsDrv – Network File System (NFS) I/O driver

ROUTINES

nfsDrv( ) - install the NFS driver
nfsDrvNumGet( ) - return the IO system driver number for the NFS driver
nfsMount( ) - mount an NFS file system
nfsMountAll( ) - mount all file systems exported by a specified host
nfsDevShow( ) - display the mounted NFS devices
nfsUnmount( ) - unmount an NFS device
nfsDevListGet( ) - create list of all the NFS devices in the system
nfsDevInfoGet( ) - read configuration information from the requested NFS device

DESCRIPTION

This driver provides facilities for accessing files transparently over the network via NFS (Network File System). By creating a network device with **nfsMount( )**, files on a remote NFS system (such as a UNIX system) can be handled as if they were local.

USER-CALLABLE ROUTINES

The **nfsDrv( )** routine initializes the driver. The **nfsMount( )** and **nfsUnmount( )** routines mount and unmount file systems. The **nfsMountAll( )** routine mounts all file systems exported by a specified host.

INITIALIZATION

Before using the network driver, it must be initialized by calling **nfsDrv( )**. This routine must be called before any reads, writes, or other NFS calls. This is done automatically when **INCLUDE_NFS** is defined.

CREATING NFS DEVICES

In order to access a remote file system, an NFS device must be created by calling **nfsMount( )**. For example, to create the device **/myd0/** for the file system **/d0/** on the host **wrs**, call:

```
nfsMount ("wrs", "/d0/", "/myd0/");
```

The file **/d0/dog** on the host **wrs** can now be accessed as **/myd0/dog**.

If the third parameter to **nfsMount( )** is **NULL**, VxWorks creates a device with the same name as the file system. For example, the call:

```
nfsMount ("wrs", "/d0/", NULL);
```

or from the shell:

```
nfsMount "wrs", "/d0/"
```

creates the device **/d0/**. The file **/d0/dog** is accessed by the same name, **/d0/dog**.

Before mounting a file system, the host must already have been created with **hostAdd( )**. The routine **nfsDevShow( )** displays the mounted NFS devices.

IOCTL FUNCTIONS
The NFS driver responds to the following **ioctl( )** functions:

FIOGETNAME
Gets the file name of *fd* and copies it to the buffer referenced by *nameBuf*:

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

FIONREAD
Copies to *nBytesUnread* the number of bytes remaining in the file specified by *fd*:

```
status = ioctl (fd, FIONREAD, &nBytesUnread);
```

FIOSEEK
Sets the current byte offset in the file to the position specified by *newOffset*. If the seek goes beyond the end-of-file, the file grows. The end-of-file pointer gets moved to the new position, and the new space is filled with zeros:

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

FIOSYNC
Flush data to the remote NFS file. It takes no additional argument:

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

FIOWHERE
Returns the current byte position in the file. This is the byte offset of the next byte to be read or written. It takes no additional argument:

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

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

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

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

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

FIOFSTATFSGET
Gets the file system parameters for and open file descriptor. The argument *statfsStruct* is a pointer to a **statfs** structure that is filled with data describing the underlying file system. Normally, the **stat( )** or **fstat( )** routine is used to obtain file information, rather than using the **FIOFSTATGET** function directly. See the manual entry for **dirLib**:

```
statfs statfsStruct;
fd = open ("directory", O_RDONLY);
status = ioctl (fd, FIOFSTATFSGET, &statfsStruct);
```

DEFICIENCIES
There is only one client handle/cache per task. Performance is poor if a task is accessing two or more NFS files.

Changing *nfsCacheSize* after a file is open could cause adverse effects. However, changing it before opening any NFS file descriptors should not pose a problem.

INCLUDE FILES
nfsDrv.h, **ioLib.h**, **dirent.h**

SEE ALSO
dirLib, **nfsLib**, **hostAdd( )**, **ioctl( )**,

N

nfsLib

NAME
nfsLib – Network File System (NFS) library

ROUTINES
nfsHelp( ) - display the NFS help menu
nfsExportShow( ) - display the exported file systems of a remote host
nfsAuthUnixPrompt( ) - modify the NFS UNIX authentication parameters
nfsAuthUnixShow( ) - display the NFS UNIX authentication parameters
nfsAuthUnixSet( ) - set the NFS UNIX authentication parameters
nfsAuthUnixGet( ) - get the NFS UNIX authentication parameters
nfsIdSet( ) - set the ID number of the NFS UNIX authentication parameters

DESCRIPTION

This library provides the client side of services for NFS (Network File System) devices. Most routines in this library should not be called by users, but rather by device drivers. The driver is responsible for keeping track of file pointers, mounted disks, and cached buffers. This library uses Remote Procedure Calls (RPC) to make the NFS calls.

VxWorks is delivered with NFS disabled. To use this feature, include the following component: **INCLUDE_NFS**

In the same file, **NFS_USER_ID** and **NFS_GROUP_ID** should be defined to set the default user ID and group ID at system start-up. For information about creating NFS devices, see the *WindNet TCP/IP Network Programmer's Guide.*

Normal use of NFS requires no more than 2000 bytes of stack. This requirement may change depending on how the maximum file name path length parameter, **NFS_MAXPATH**, is configured. As many as 4 character arrays of length **NFS_MAXPATH** may be allocated off the stack during client operation. Therefore any increase in the parameter can increase stack usage by a factor of four times the deviation from default **NFS_MAXPATH**. For example, a change from 255 to 1024 will increase peak stack usage by (1024 -255) * 4 which is 3076 bytes.

NFS USER IDENTIFICATION

NFS is built on top of RPC and uses a type of RPC authentication known as **AUTH_UNIX**, which is passed on to the NFS server with every NFS request. **AUTH_UNIX** is a structure that contains necessary information for NFS, including the user ID number and a list of group IDs to which the user belongs. On UNIX systems, a user ID is specified in the file **/etc/passwd**. The list of groups to which a user belongs is specified in the file **/etc/group**.

To change the default authentication parameters, use **nfsAuthUnixPrompt( )**. To change just the **AUTH_UNIX** ID, use **nfsIdSet( )**. Usually, only the user ID needs to be changed to indicate a new NFS user.

INCLUDE FILES

nfsLib.h

SEE ALSO

rpcLib, **ioLib**, **nfsDrv**

ntPassFsLib

NAME

ntPassFsLib – pass-through (to Windows NT) file system library

ROUTINES

ntPassFsDevInit( ) - associate a device with ntPassFs file system functions
ntPassFsInit( ) - prepare to use the ntPassFs library

DESCRIPTION

This module is only used with VxSim simulated versions of VxWorks.

This library provides services for file-oriented device drivers to use the Windows NT file standard. In general, the routines in this library are not to be called directly by users, but rather by the VxWorks I/O System.

INITIALIZING PASSFSLIB

Before any other routines in **ntPassFsLib** can be used, the routine **ntPassFsInit( )** must be called to initialize this library. The **ntPassFsDevInit( )** routine associates a device name with the **ntPassFsLib** functions. The parameter expected by **ntPassFsDevInit( )** is a pointer to a name string, to be used to identify the volume/device. This will be part of the pathname for I/O operations which operate on the device. This name will appear in the I/O system device table, which may be displayed using the **iosDevShow( )** routine.

As an example:

```
ntPassFsInit (1);
ntPassFsDevInit ("host:");
```

After the **ntPassFsDevInit( )** call has been made, when **ntPassFsLib** receives a request from the I/O system, it calls the Windows NT I/O system to service the request. Only one volume may be created.

READING DIRECTORY ENTRIES

Directories on a ntPassFs volume may be searched using the **opendir( )**, **readdir( )**, **rewinddir( )**, and **closedir( )** routines. These calls allow the names of files and sub-directories to be determined.

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

FILE DATE AND TIME

Windows NT file date and time are passed through to VxWorks.

INCLUDE FILES **ntPassFsLib.h**

SEE ALSO **ioLib**, **iosLib**, **dirLib**, **ramDrv**

passFsLib

NAME

passFsLib – pass-through (to UNIX) file system library (VxSim)

ROUTINES

passFsDevInit( ) - associate a device with passFs file system functions
passFsInit( ) - prepare to use the passFs library

DESCRIPTION

This module is only used with VxSim simulated versions of VxWorks.

This library provides services for file-oriented device drivers to use the UNIX file standard. This module takes care of all the buffering, directory maintenance, and file system details that are necessary. In general, the routines in this library are not to be called directly by users, but rather by the VxWorks I/O System.

INITIALIZING PASSFSLIB

Before any other routines in **passFsLib** can be used, the routine **passFsInit( )** must be called to initialize this library. The **passFsDevInit( )** routine associates a device name with the **passFsLib** functions. The parameter expected by **passFsDevInit( )** is a pointer to a name string, to be used to identify the volume/device. This will be part of the pathname for I/O operations which operate on the device. This name will appear in the I/O system device table, which may be displayed using the **iosDevShow( )** routine.

As an example:

```
passFsInit (1);
passFsDevInit ("host:");
```

After the **passFsDevInit( )** call has been made, when **passFsLib** receives a request from the I/O system, it calls the UNIX I/O system to service the request. Only one volume may be created.

READING DIRECTORY ENTRIES

Directories on a passFs volume may be searched using the **opendir( )**, **readdir( )**, **rewinddir( )**, and **closedir( )** routines. These calls allow the names of files and sub-directories to be determined.

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

FILE DATE AND TIME

UNIX file date and time are passed though to VxWorks.

INCLUDE FILES

passFsLib.h

SEE ALSO

ioLib, **iosLib**, **dirLib**, **ramDrv**

pentiumALib

NAME

pentiumALib – Pentium and PentiumPro specific routines

ROUTINES

pentiumCr4Get( ) - get contents of CR4 register
pentiumCr4Set( ) - sets specified value to the CR4 register
pentiumP6PmcStart( ) - start both PMC0 and PMC1
pentiumP6PmcStop( ) - stop both PMC0 and PMC1
pentiumP6PmcStop1( ) - stop PMC1
pentiumP6PmcGet( ) - get the contents of PMC0 and PMC1
pentiumP6PmcGet0( ) - get the contents of PMC0
pentiumP6PmcGet1( ) - get the contents of PMC1
pentiumP6PmcReset( ) - reset both PMC0 and PMC1
pentiumP6PmcReset0( ) - reset PMC0
pentiumP6PmcReset1( ) - reset PMC1
pentiumP5PmcStart0( ) - start PMC0
pentiumP5PmcStart1( ) - start PMC1
pentiumP5PmcStop( ) - stop both P5 PMC0 and PMC1
pentiumP5PmcStop0( ) - stop P5 PMC0
pentiumP5PmcStop1( ) - stop P5 PMC1
pentiumP5PmcGet( ) - get the contents of P5 PMC0 and PMC1
pentiumP5PmcGet0( ) - get the contents of P5 PMC0
pentiumP5PmcGet1( ) - get the contents of P5 PMC1
pentiumP5PmcReset( ) - reset both PMC0 and PMC1
pentiumP5PmcReset0( ) - reset PMC0
pentiumP5PmcReset1( ) - reset PMC1
pentiumTscGet64( ) - get 64Bit TSC (Timestamp Counter)
pentiumTscGet32( ) - get the lower half of the 64Bit TSC (Timestamp Counter)
pentiumTscReset( ) - reset the TSC (Timestamp Counter)
pentiumMsrGet( ) - get the contents of the specified MSR (Model Specific Register)
pentiumMsrSet( ) - set a value to the specified MSR (Model Specific Registers)
pentiumTlbFlush( ) - flush TLBs (Translation Lookaside Buffers)
pentiumSerialize( ) - execute a serializing instruction CPUID
pentiumBts( ) - execute atomic compare-and-exchange instruction to set a bit
pentiumBtc( ) - execute atomic compare-and-exchange instruction to clear a bit

DESCRIPTION

This module contains Pentium and PentiumPro specific routines written in assembly language.

MCA (Machine Check Architecture)

The Pentium processor introduced a new exception called the machine-check exception (interrupt-18). This exception is used to signal hardware-related errors, such as a parity error on a read cycle. The PentiumPro processor extends the types of errors that can be detected and that generate a machine- check exception. It also provides a new machine-check architecture that records information about a machine-check error and provides the basis for an extended error logging capability.

MCA is enabled and its status registers are cleared zero in **sysHwInit( )**. Its registers are accessed by **pentiumMsrSet( )** and **pentiumMsrGet( )**.

PMC (Performance Monitoring Counters)

The P5 and P6 family of processor has two performance-monitoring counters for use in monitoring internal hardware operations. These counters are duration or event counters that can be programmed to count any of approximately 100 different types of events, such as the number of instructions decoded, number of interrupts received, or number of cache loads. However, the set of events can be counted with PMC is different in the P5 and P6 family of processors; and the locations and bit definitions of the related counter and control registers are also different. So there are two set of PMC routines, one for P6 family and one for p5 family respectively.

There are nine routines to interface the PMC of P6 family processors. These nine routines are:

```
STATUS pentiumP6PmcStart
       (
       int pmcEvtSel0;        /* performance event select register 0 */
       int pmcEvtSel1;        /* performance event select register 1 */
       )
void   pentiumP6PmcStop (void)
void   pentiumP6PmcStop1 (void)
void   pentiumP6PmcGet
       (
       long long int * pPmc0; /* performance monitoring counter 0 */
       long long int * pPmc1; /* performance monitoring counter 1 */
       )
void   pentiumP6PmcGet0
       (
       long long int * pPmc0; /* performance monitoring counter 0 */
       )
void   pentiumP6PmcGet1
       (
       long long int * pPmc1; /* performance monitoring counter 1 */
       )
void   pentiumP6PmcReset (void)
void   pentiumP6PmcReset0 (void)
void   pentiumP6PmcReset1 (void)
```

pentiumP6PmcStart( ) starts both PMC0 and PMC1. **pentiumP6PmcStop( )** stops them, and **pentiumP6PmcStop1( )** stops only PMC1. **pentiumP6PmcGet( )** gets contents of PMC0 and PMC1. **pentiumP6PmcGet0( )** gets contents of PMC0, and **pentiumP6PmcGet1( )** gets contents of PMC1. **pentiumP6PmcReset( )** resets both PMC0 and PMC1. **pentiumP6PmcReset0( )** resets PMC0, and **pentiumP6PmcReset1( )** resets PMC1. PMC is enabled in **sysHwInit( )**. Selected events in the default configuration are PMC0 = number of hardware interrupts received and PMC1 = number of misaligned data memory references.

There are ten routines to interface the PMC of P5 family processors. These ten routines are:

```
STATUS pentiumP5PmcStart0
       (
       int pmc0Cesr;  /* PMC0 control and event select */
       )
STATUS pentiumP5PmcStart1
       (
       int pmc1Cesr;  /* PMC1 control and event select */
       )
void   pentiumP5PmcStop0 (void)
void   pentiumP5PmcStop1 (void)
void   pentiumP5PmcGet
       (
       long long int * pPmc0; /* performance monitoring counter 0 */
       long long int * pPmc1; /* performance monitoring counter 1 */
       )
void   pentiumP5PmcGet0
       (
       long long int * pPmc0; /* performance monitoring counter 0 */
       )
void   pentiumP5PmcGet1
       (
       long long int * pPmc1; /* performance monitoring counter 1 */
       )
void   pentiumP5PmcReset (void)
void   pentiumP5PmcReset0 (void)
void   pentiumP5PmcReset1 (void)
```

pentiumP5PmcStart0( ) starts PMC0, and **pentiumP5PmcStart1( )** starts PMC1. **pentiumP5PmcStop0( )** stops PMC0, and **pentiumP5PmcStop1( )** stops PMC1. **pentiumP5PmcGet( )** gets contents of PMC0 and PMC1. **pentiumP5PmcGet0( )** gets contents of PMC0, and **pentiumP5PmcGet1( )** gets contents of PMC1. **pentiumP5PmcReset( )** resets both PMC0 and PMC1. **pentiumP5PmcReset0( )** resets PMC0, and **pentiumP5PmcReset1( )** resets PMC1. PMC is enabled in **sysHwInit( )**. Selected events in the default configuration are PMC0 = number of hardware interrupts received and PMC1 = number of misaligned data memory references.

MSR (Model Specific Register)

The concept of model-specific registers (MSRs) to control hardware functions in the processor or to monitor processor activity was introduced in the PentiumPro processor.

The new registers control the debug extensions, the performance counters, the machine-check exception capability, the machine check architecture, and the MTRRs. The MSRs can be read and written to using the RDMSR and WRMSR instructions, respectively.

There are two routines to interface the MSR. These two routines are:

```
void pentiumMsrGet
     (
     int address,              /* MSR address */
     long long int * pData     /* MSR data */
     )
void pentiumMsrSet
     (
     int address,              /* MSR address */
     long long int * pData     /* MSR data */
     )
```

pentiumMsrGet( ) get contents of the specified MSR, and **pentiumMsrSet( )** sets value to the specified MSR.

TSC (Time Stamp Counter)

The PentiumPro processor provides a 64-bit time-stamp counter that is incremented every processor clock cycle. The counter is incremented even when the processor is halted by the HLT instruction or the external STPCLK# pin. The time-stamp counter is set to 0 following a hardware reset of the processor. The RDTSC instruction reads the time stamp counter and is guaranteed to return a monotonically increasing unique value whenever executed, except for 64-bit counter wraparound. Intel guarantees, architecturally, that the time-stamp counter frequency and configuration will be such that it will not wraparound within 10 years after being reset to 0. The period for counter wrap is several thousands of years in the PentiumPro and Pentium processors.

There are three routines to interface the TSC. These three routines are:

```
void pentiumTscReset (void)
void pentiumTscGet32 (void)
void pentiumTscGet64
     (
     long long int * pTsc      /* TSC */
     )
```

pentiumTscReset( ) resets the TSC. **pentiumTscGet32( )** gets the lower half of the 64Bit TSC, and **pentiumTscGet64( )** gets the entire 64Bit TSC.

Four other routines are provided in this library. They are:

```
void   pentiumTlbFlush (void)
void   pentiumSerialize (void)
STATUS pentiumBts
```

```
        (
        char * pFlag                    /* flag address */
        )
STATUS pentiumBtc (pFlag)
        (
        char * pFlag                    /* flag address */
        )
```

pentiumTlbFlush( ) flushes TLBs (Translation Lookaside Buffers). **pentiumSerialize( )** does serialization by executing CPUID instruction. **pentiumBts( )** executes an atomic compare-and-exchange instruction to set a bit. **pentiumBtc( )** executes an atomic compare-and-exchange instruction to clear a bit.

SEE ALSO *Pentium, PentiumPro Family Developer's Manual*

---

pentiumLib

NAME **pentiumLib** – Pentium and Pentium[234] library

ROUTINES

pentiumMtrrEnable( ) - enable MTRR (Memory Type Range Register)
pentiumMtrrDisable( ) - disable MTRR (Memory Type Range Register)
pentiumMtrrGet( ) - get MTRRs to a specified MTRR table
pentiumMtrrSet( ) - set MTRRs from specified MTRR table with WRMSR instruction.
pentiumPmcStart( ) - start both PMC0 and PMC1
pentiumPmcStart0( ) - start PMC0
pentiumPmcStart1( ) - start PMC1
pentiumPmcStop( ) - stop both PMC0 and PMC1
pentiumPmcStop0( ) - stop PMC0
pentiumPmcStop1( ) - stop PMC1
pentiumPmcGet( ) - get the contents of PMC0 and PMC1
pentiumPmcGet0( ) - get the contents of PMC0
pentiumPmcGet1( ) - get the contents of PMC1
pentiumPmcReset( ) - reset both PMC0 and PMC1
pentiumPmcReset0( ) - reset PMC0
pentiumPmcReset1( ) - reset PMC1
pentiumMsrInit( ) - initialize all the MSRs (Model Specific Register)
pentiumMcaEnable( ) - enable/disable the MCA (Machine Check Architecture)

DESCRIPTION This library provides Pentium and Pentium[234] specific routines.

MTRR (Memory Type Range Register)

MTRR (Memory Type Range Register) are a new feature introduced in the P6 family

P

processor that allow the processor to optimize memory operations for different types of memory, such as RAM, ROM, frame buffer memory, and memory-mapped IO. MTRRs configure an internal map of how physical address ranges are mapped to various types of memory. The processor uses this internal map to determine the cacheability of various physical memory locations and the optimal method of accessing memory locations. For example, if a memory location is specified in an MTRR as write-through memory, the processor handles accesses to this location as follows. It reads data from that location in lines and caches the read data or maps all writes to that location to the bus and updates the cache to maintain cache coherency. In mapping the physical address space with MTRRs, the processor recognizes five types of memory: uncacheable (UC), write-combining (WC), write-through (WT), write-protected (WP), and write-back (WB).

There is one table - **sysMtrr[]** in **sysLib.c** - and four routines to interface the MTRR. These four routines are:

```
void pentiumMtrrEnable (void)
void pentiumMtrrDisable (void)
STATUS pentiumMtrrGet
    (
    MTRR * pMtrr              /* MTRR table */
    )
STATUS pentiumMtrrSet (void)
    (
    MTRR * pMtrr              /* MTRR table */
    )
```

pentiumMtrrEnable( ) enables MTRR, **pentiumMtrrDisable( )** disables MTRR. **pentiumMtrrGet( )** gets MTRRs to the specified MTRR table. **pentiumMtrrGet( )** sets MTRRs from the specified MTRR table. The MTRR table is defined as follows:

```
typedef struct mtrr_fix         /* MTRR - fixed range register */
    {
    char type[8];               /* address range: [0]=0-7 ... [7]=56-63 */
    } MTRR_FIX;
typedef struct mtrr_var         /* MTRR - variable range register */
    {
    long long int base;         /* base register */
    long long int mask;         /* mask register */
    } MTRR_VAR;
typedef struct mtrr             /* MTRR */
    {
    int cap[2];                 /* MTRR cap register */
    int deftype[2];             /* MTRR defType register */
    MTRR_FIX fix[11];           /* MTRR fixed range registers */
    MTRR_VAR var[8];            /* MTRR variable range registers */
    } MTRR;
```

Fixed Range Register's type array can be one of following memory types. **MTRR_UC** (uncacheable), **MTRR_WC** (write-combining), **MTRR_WT** (write-through), **MTRR_WP** (write-protected), and **MTRR_WB** (write-back). MTRR is enabled in **sysHwInit( )**.

PMC (Performance Monitoring Counters)

The P5 and P6 family of processors has two performance-monitoring counters for use in monitoring internal hardware operations. These counters are duration or event counters that can be programmed to count any of approximately 100 different types of events, such as the number of instructions decoded, number of interrupts received, or number of cache loads. However, the set of events can be counted with PMC is different in the P5 and P6 family of processors; and the locations and bit definitions of the related counter and control registers are also different. So there are two set of PMC routines, one for P6 family and one for P5 family respectively in **pentiumALib**. For convenience, the PMC routines here are acting as wrappers to those routines in **pentiumALib**. They will call the P5 or P6 routine depending on the processor type.

There are twelve routines to interface the PMC. These twelve routines are:

```
STATUS pentiumPmcStart
       (
       int pmcEvtSel0;        /* performance event select register 0 */
       int pmcEvtSel1;        /* performance event select register 1 */
       )
STATUS pentiumPmcStart0
       (
       int pmcEvtSel0;        /* performance event select register 0 */
       )
STATUS pentiumPmcStart1
       (
       int pmcEvtSel1;        /* performance event select register 1 */
       )
void   pentiumPmcStop (void)
void   pentiumPmcStop0 (void)
void   pentiumPmcStop1 (void)
void   pentiumPmcGet
       (
       long long int * pPmc0; /* performance monitoring counter 0 */
       long long int * pPmc1; /* performance monitoring counter 1 */
       )
void   pentiumPmcGet0
       (
       long long int * pPmc0; /* performance monitoring counter 0 */
       )
void   pentiumPmcGet1
       (
       long long int * pPmc1; /* performance monitoring counter 1 */
```

P

```
        )
void   pentiumPmcReset (void)
void   pentiumPmcReset0 (void)
void   pentiumPmcReset1 (void)
```

pentiumPmcStart( ) starts both PMC0 and PMC1. **pentiumPmcStart0( )** starts PMC0, and **pentiumPmcStart1( )** starts PMC1. **pentiumPmcStop( )** stops both PMC0 and PMC1. **pentiumPmcStop0( )** stops PMC0, and **pentiumPmcStop1( )** stops PMC1. **pentiumPmcGet( )** gets contents of PMC0 and PMC1. **pentiumPmcGet0( )** gets contents of PMC0, and **pentiumPmcGet1( )** gets contents of PMC1. **pentiumPmcReset( )** resets both PMC0 and PMC1. **pentiumPmcReset0( )** resets PMC0, and **pentiumPmcReset1( )** resets PMC1. PMC is enabled in **sysHwInit( )**. Selected events in the default configuration are PMC0 = number of hardware interrupts received and PMC1 = number of misaligned data memory references.

MSR (Model Specific Registers)

The P5(Pentium), P6(PentiumPro, II, III), and P7(Pentium4) family processors contain a model-specific registers (MSRs). These registers are implementation specific. They are provided to control a variety of hardware and software related features including the performance monitoring, the debug extensions, the machine check architecture, *etc*.

There is one routine - **pentiumMsrInit( )** - to initialize all the MSRs. This routine initializes all the MSRs in the processor and works on either P5, P6 or P7 family processors.

MCA (Machine Check Architecture)

The P5(Pentium), P6(PentiumPro, II, III), and P7(Pentium4) family processors have a machine-check architecture that provides a mechanism for detecting and reporting hardware (machine) errors, such as system bus errors, ECC errors, parity errors, cache errors and TLB errors. It consists of a set of model-specific registers (MSRs) that are used to set up machine checking and additional banks of MSRs for recording errors that are detected. The processor signals the detection of a machine-check error by generating a machine-check exception, which an abort class exception. The implementation of the machine-check architecture, does not ordinarily permit the processor to be restarted reliably after generating a machine-check exception. However, the machine-check exception handler can collect information about the machine-check error from the machine-check MSRs.

There is one routine - **pentiumMcaEnable( )** - to enable or disable the MCA. The routine enables or disables 1) the Machine Check Architecture and its Error Reporting register banks 2) the Machine Check Exception by toggling the MCE bit in the CR4. This routine works on either P5, P6 or P7 family.

SEE ALSO **PentiumALib**, *Pentium*, *Pentium[234] Family Developer's Manual*

pentiumShow

NAME **pentiumShow** – Pentium and Pentium[234] specific show routines

ROUTINES
pentiumMcaShow( ) - show MCA (Machine Check Architecture) registers
pentiumPmcShow( ) - show PMCs (Performance Monitoring Counters)
pentiumMsrShow( ) - show all the MSR (Model Specific Register)

DESCRIPTION This library provides Pentium and Pentium[234] specific show routines.

pentiumMcaShow( ) shows Machine Check Global Control Registers and Error Reporting Register Banks. **pentiumPmcShow( )** shows PMC0 and PMC1, and reset them if the parameter zap is **TRUE**.

SEE ALSO *VxWorks Programmer's Guide: Configuration*

pingLib

NAME **pingLib** – Packet InterNet Groper (PING) library

ROUTINES
pingLibInit( ) - initialize the **ping( )** utility
ping( ) - test that a remote host is reachable

DESCRIPTION This library contains the **ping( )** utility, which tests the reachability of a remote host.

The routine **ping( )** is typically called from the VxWorks shell to check the network connection to another VxWorks target or to a UNIX host. **ping( )** may also be used programmatically by applications that require such a test. The remote host must be running TCP/IP networking code that responds to ICMP echo request packets. The **ping( )** routine is re-entrant, thus may be called by many tasks concurrently.

The routine **pingLibInit( )** initializes the **ping( )** utility and allocates resources used by this library. It is called automatically when **INCLUDE_PING** is defined.

pipeDrv

NAME

pipeDrv – pipe I/O driver

ROUTINES

pipeDrv( ) - initialize the pipe driver
pipeDevCreate( ) - create a pipe device
pipeDevDelete( ) - delete a pipe device

DESCRIPTION

The pipe driver provides a mechanism that lets tasks communicate with each other through the standard I/O interface. Pipes can be read and written with normal **read( )** and **write( )** calls. The pipe driver is initialized with **pipeDrv( )**. Pipe devices are created with **pipeDevCreate( )**.

The pipe driver uses the VxWorks message queue facility to do the actual buffering and delivering of messages. The pipe driver simply provides access to the message queue facility through the I/O system. The main differences between using pipes and using message queues directly are:

– pipes are named (with I/O device names).

– pipes use the standard I/O functions -- **open( )**, **close( )**, **read( )**, **write( )** -- while message queues use the functions **msgQSend( )** and **msgQReceive( )**.

– pipes respond to standard **ioctl( )** functions.

– pipes can be used in a **select( )** call.

– message queues have more flexible options for timeouts and message priorities.

– pipes are less efficient than message queues because of the additional overhead of the I/O system.

INSTALLING THE DRIVER

Before using the driver, it must be initialized and installed by calling **pipeDrv( )**. This routine must be called before any pipes are created. It is called automatically by the root task, **usrRoot( )**, in **usrConfig.c** when the configuration macro **INCLUDE_PIPES** is defined.

CREATING PIPES

Before a pipe can be used, it must be created with **pipeDevCreate( )**. For example, to create a device pipe **/pipe/demo** with up to 10 messages of size 100 bytes, the proper call is:

```
pipeDevCreate ("/pipe/demo", 10, 100);
```

USING PIPES

Once a pipe has been created it can be opened, closed, read, and written just like any other I/O device. Often the data that is read and written to a pipe is a structure of some type. Thus, the following example writes to a pipe and reads back the same data:

```
{
int fd;
struct msg outMsg;
struct msg inMsg;
int len;
fd = open ("/pipe/demo", O_RDWR);
write (fd, &outMsg, sizeof (struct msg));
len = read (fd, &inMsg, sizeof (struct msg));
close (fd);
}
```

The data written to a pipe is kept as a single message and will be read all at once in a single read. If **read( )** is called with a buffer that is smaller than the message being read, the remainder of the message will be discarded. Thus, pipe I/O is "message oriented" rather than "stream oriented." In this respect, VxWorks pipes differ significantly from UNIX pipes which are stream oriented and do not preserve message boundaries.

WRITING TO PIPES FROM INTERRUPT SERVICE ROUTINES

Interrupt service routines (ISR) can write to pipes, providing one of several ways in which ISRs can communicate with tasks. For example, an interrupt service routine may handle the time-critical interrupt response and then send a message on a pipe to a task that will continue with the less critical aspects. However, the use of pipes to communicate from an ISR to a task is now discouraged in favor of the direct message queue facility, which offers lower overhead (see the manual entry for **msgQLib** for more information).

SELECT CALLS

An important feature of pipes is their ability to be used in a **select( )** call. The **select( )** routine allows a task to wait for input from any of a selected set of I/O devices. A task can use **select( )** to wait for input from any combination of pipes, sockets, or serial devices. See the manual entry for **select( )**.

IOCTL FUNCTIONS

Pipe devices respond to the following **ioctl( )** functions. These functions are defined in the header file **ioLib.h**.

FIOGETNAME
Gets the file name of *fd* and copies it to the buffer referenced by *nameBuf*:

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

FIONREAD
Copies to *nBytesUnread* the number of bytes remaining in the first message in the pipe:

```
status = ioctl (fd, FIONREAD, &nBytesUnread);
```

FIONMSGS
Copies to *nMessages* the number of discrete messages remaining in the pipe:

```
status = ioctl (fd, FIONMSGS, &nMessages);
```

P

FIOFLUSH
Discards all messages in the pipe and releases the memory block that contained them:

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

INCLUDE FILES **ioLib.h**, **pipeDrv.h**

SEE ALSO **select( )**, **msgQLib**, *VxWorks Programmer's Guide: I/O System*

pppHookLib

NAME **pppHookLib** – PPP hook library

ROUTINES
pppHookAdd( ) - add a hook routine on a unit basis
pppHookDelete( ) - delete a hook routine on a unit basis

DESCRIPTION This library provides routines to add and delete connect and disconnect routines. The connect routine, added on a unit basis, is called before the initial phase of link option negotiation. The disconnect routine, added on a unit basis is called before the PPP connection is closed. These connect and disconnect routines can be used to hook up additional software. If either connect or disconnect hook returns **ERROR**, the connection is terminated immediately.

This library is automatically linked into the VxWorks system image when the configuration macro **INCLUDE_PPP** is defined.

INCLUDE FILES **pppLib.h**

SEE ALSO **pppLib**, *VxWorks Programmer's Guide: Network*

pppLib

NAME **pppLib** – Point-to-Point Protocol library

ROUTINES
pppInit( ) - initialize a PPP network interface
pppDelete( ) - delete a PPP network interface

DESCRIPTION This library implements the VxWorks Point-to-Point Protocol (PPP) facility. PPP allows VxWorks to communicate with other machines by sending encapsulated multi-protocol datagrams over a point-to-point serial link. VxWorks may have up to 16 PPP interfaces active at any one time. Each individual interface (or "unit") operates independent of the state of other PPP units.

USER-CALLABLE ROUTINES

PPP network interfaces are initialized using the **pppInit( )** routine. This routine's parameters specify the unit number, the name of the serial interface (*tty*) device, Internet (IP) addresses for both ends of the link, the interface baud rate, an optional pointer to a configuration options structure, and an optional pointer to a configuration options file. The **pppDelete( )** routine deletes a specified PPP interface.

DATA ENCAPSULATION

PPP uses HDLC-like framing, in which five header and three trailer octets are used to encapsulate each datagram. In environments where bandwidth is at a premium, the total encapsulation may be shortened to four octets with the available address/control and protocol field compression options.

LINK CONTROL PROTOCOL

PPP incorporates a link-layer protocol called Link Control Protocol (LCP), which is responsible for the link set up, configuration, and termination. LCP provides for automatic negotiation of several link options, including datagram encapsulation format, user authentication, and link monitoring (LCP echo request/reply).

NETWORK CONTROL PROTOCOLS

PPP's Network Control Protocols (NCP) allow PPP to support different network protocols. VxWorks supports only one NCP, the Internet Protocol Control Protocol (IPCP), which allows the establishment and configuration of IP over PPP links. IPCP supports the negotiation of IP addresses and TCP/IP header compression (commonly called "VJ" compression).

AUTHENTICATION

The VxWorks PPP implementation supports two separate user authentication protocols: the Password Authentication Protocol (PAP) and the Challenge-Handshake Authentication Protocol (CHAP). While PAP only authenticates at the time of link establishment, CHAP may be configured to periodically require authentication throughout the life of the link. Both protocols are independent of one another, and either may be configured in through the PPP options structure or options file.

IMPLEMENTATION

Each VxWorks PPP interface is handled by two tasks: the daemon task (tPPP*unit*) and the write task (tPPP*unit*Wrt).

The daemon task controls the various PPP control protocols (LCP, IPCP, CHAP, and PAP). Each PPP interface has its own daemon task that handles link set up, negotiation of link options, link-layer user authentication, and link termination. The daemon task is not used for the actual sending and receiving of IP datagrams.

The write task controls the transmit end of a PPP driver interface. Each PPP interface has its own write task that handles the actual sending of a packet by writing data to the *tty* device. Whenever a packet is ready to be sent out, the PPP driver activates this task by giving a semaphore. The write task then completes the packet framing and writes the packet data to the *tty* device.

The receive end of the PPP interface is implemented as a "hook" into the *tty* device driver. The *tty* driver's receive interrupt service routine (ISR) calls the PPP driver's ISR every time a character is received on the serial channel. When the correct PPP framing character sequence is received, the PPP ISR schedules the **tNetTask** task to call the PPP input routine. The PPP input routine reads a whole PPP packet out of the *tty* ring buffer and processes it according to PPP framing rules. The packet is then queued either to the IP input queue or to the PPP daemon task input queue.

INCLUDE FILES **pppLib.h**

SEE ALSO **ifLib**, **tyLib**, **pppSecretLib**, **pppShow**, *VxWorks Programmer's Guide: Network*, *RFC-1332: The PPP Internet Protocol Control Protocol (IPCP)*, *RFC-1334: PPP Authentication Protocols*, *RFC-1548: The Point-to-Point Protocol (PPP)*, *RFC-1549: PPP in HDLC Framing*

ACKNOWLEDGEMENT

This program is based on original work done by Paul Mackerras of Australian National University, Brad Parker, Greg Christy, Drew D. Perkins, Rick Adams, and Chris Torek.

pppSecretLib

NAME **pppSecretLib** – PPP authentication secrets library

ROUTINES

pppSecretAdd( ) - add a secret to the PPP authentication secrets table
pppSecretDelete( ) - delete a secret from the PPP authentication secrets table

DESCRIPTION

This library provides routines to create and manipulate a table of "secrets" for use with Point-to-Point Protocol (PPP) user authentication protocols. The secrets in the secrets table can be searched by peers on a PPP link so that one peer (client) can send a secret word to the other peer (server). If the client cannot find a suitable secret when required to do so, or the secret received by the server is not valid, the PPP link may be terminated.

This library is automatically linked into the VxWorks system image when the configuration macro **INCLUDE_PPP** is defined.

INCLUDE FILES **pppLib.h**

SEE ALSO **pppLib**, **pppShow**, *VxWorks Programmer's Guide: Network*

pppShow

NAME **pppShow** – Point-to-Point Protocol show routines

ROUTINES
pppInfoShow( ) - display PPP link status information
pppInfoGet( ) - get PPP link status information
pppstatShow( ) - display PPP link statistics
pppstatGet( ) - get PPP link statistics
pppSecretShow( ) - display the PPP authentication secrets table

DESCRIPTION This library provides routines to show Point-to-Point Protocol (PPP) link status information and statistics. Also provided are routines that programmatically access this same information.

This library is automatically linked into the VxWorks system image when the configuration macro **INCLUDE_PPP** is defined.

INCLUDE FILES **pppLib.h**

SEE ALSO **pppLib**, *VxWorks Programmer's Guide: Network*

proxyArpLib

NAME **proxyArpLib** – proxy Address Resolution Protocol (ARP) server library

ROUTINES
proxyArpLibInit( ) - initialize proxy ARP
proxyNetCreate( ) - create a proxy ARP network
proxyNetDelete( ) - delete a proxy network
proxyNetShow( ) - show proxy ARP networks
proxyPortFwdOn( ) - enable broadcast forwarding for a particular port
proxyPortFwdOff( ) - disable broadcast forwarding for a particular port
proxyPortShow( ) - show ports enabled for broadcast forwarding

DESCRIPTION This library implements a proxy ARP server that uses the Address Resolution Protocol (ARP) to make physically distinct networks appear as one logical network (that is, the networks share the same address space). The server forwards ARP messages between the separate networks so that hosts on the main network can access hosts on the proxy network without altering their routing tables.

The **proxyArpLibInit( )** initializes the server and adds this library to the VxWorks image. This happens automatically if **INCLUDE_PROXY_SERVER** is defined at the time the image is built. The **proxyNetCreate( )** and **proxyNetDelete( )** routines will enable and disable the forwarding of ARP messages between networks. The **proxyNetShow( )** routine displays the current set of proxy networks and the main network and known clients for each.

By default, this server automatically adds a client when it first detects an ARP message from that host. A VxWorks target can also register as a client with the **proxyReg( )** routine and remove that registration with the **proxyUnreg( )** routine. See the **proxyLib** manual pages for details.

To minimize traffic on the main network, the proxy server will only forward broadcast packets to the specified destination ports visible with the **proxyPortShow( )** routine. The **proxyPortFwdOn( )** and **proxyPortFwdOff( )** routines will alter the current settings. Initially, broadcast forwarding is not active for any ports.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, the functions you assign for either **proxyArpHook** or **proxyBroadcastHook** must be valid within the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

INCLUDE FILES **proxyArpLib.h**

SEE ALSO **proxyLib**, RFC 925, RFC 1027, RFC 826

proxyLib

NAME **proxyLib** – proxy Address Resolution Protocol (ARP) client library

ROUTINES

proxyReg( ) - register a proxy client
proxyUnreg( ) - unregister a proxy client

DESCRIPTION

This library implements the client side of the proxy Address Resolution Protocol (ARP). It allows a VxWorks target to register itself as a proxy client by calling **proxyReg( )** and to unregister itself by calling **proxyUnreg( )**.

Both commands take an interface name and an IP address as arguments. The interface, *ifName*, specifies the interface through which to send the message. *ifName* must be a backplane interface. *proxyAddr* is the IP address associated with the interface *ifName*.

To use this feature, include **INCLUDE_PROXY_CLIENT**.

INCLUDE FILES **proxyArpLib.h**

SEE ALSO **proxyArpLib**

pthreadLib

NAME

pthreadLib – POSIX 1003.1c thread library interfaces

ROUTINES

pthreadLibInit( ) - initialize POSIX threads support
pthread_sigmask( ) - change and/or examine calling thread's signal mask (POSIX)
pthread_kill( ) - send a signal to a thread (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 attributes 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_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_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)

P

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_attr_init( ) - initialize thread attributes object (POSIX)
pthread_attr_destroy( ) - destroy a thread attributes object (POSIX)
pthread_attr_setname( ) - set name in thread attribute object
pthread_attr_getname( ) - get name of thread attribute object
pthread_attr_setstacksize( ) - set *stacksize* attribute in thread attributes
pthread_attr_getstacksize( ) - get stack value of *stacksize* attribute from thread attributes object (POSIX)
pthread_attr_setstackaddr( ) - set *stackaddr* attribute in thread attributes object (POSIX)
pthread_attr_getstackaddr( ) - get value of *stackaddr* attribute from thread attributes object (POSIX)
pthread_attr_setdetachstate( ) - set *detachstate* attribute in thread attributes object (POSIX)
pthread_attr_getdetachstate( ) - get value of *detachstate* attribute from thread attributes object (POSIX)
pthread_create( ) - create a thread (POSIX)
pthread_detach( ) - dynamically detach a thread (POSIX)
pthread_join( ) - wait for a thread to terminate (POSIX)
pthread_exit( ) - terminate a thread (POSIX)
pthread_equal( ) - compare thread IDs (POSIX)
pthread_self( ) - get the calling thread's ID (POSIX)
pthread_once( ) - dynamic package initialization (POSIX)
pthread_key_create( ) - create a thread specific data key (POSIX)
pthread_setspecific( ) - set thread specific data (POSIX)
pthread_getspecific( ) - get thread specific data (POSIX)
pthread_key_delete( ) - delete a thread specific data key (POSIX)
pthread_cancel( ) - cancel execution of a thread (POSIX)
pthread_setcancelstate( ) - set cancellation state for calling thread (POSIX)
pthread_setcanceltype( ) - set cancellation type for calling thread (POSIX)
pthread_testcancel( ) - create a cancellation point in the calling thread (POSIX)
pthread_cleanup_push( ) - pushes a routine onto the cleanup stack (POSIX)
pthread_cleanup_pop( ) - pop a cleanup routine off the top of the stack (POSIX)

DESCRIPTION

This library provides an implementation of POSIX 1003.1c threads for VxWorks. This provides an increased level of compatibility between VxWorks applications and those written for other operating systems that support the POSIX threads model (often called *pthreads*).

VxWorks is a task based operating system, rather than one implementing the process model in the POSIX sense. 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

To add POSIX threads support to a system, the component **INCLUDE_POSIX_PTHREADS** must be added.

Threads support also requires the POSIX scheduler to be included (see **schedPxLib** for more detail).

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.

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 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 two types of locking protocols available, **PTHREAD_PRIO_INHERIT** and **PTHREAD_PRIO_PROTECT**. **PTHREAD_PRIO_INHERIT** maps to a semaphore create with **SEM_PRIO_INHERIT** set (see **semMCreate** for more detail). A thread locking a mutex created with its protocol attribute set to **PTHREAD_PRIO_PROTECT** has its priority elevated to that of the *prioceiling* attribute of the mutex. When the mutex is unlocked, the priority of the calling thread is restored to its previous value.

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 signalling. 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 signalled by another thread. When the condition is signalled, the function will re-acquire the mutex and return to the caller.

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

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

RESOURCE COMPETITION

All tasks, and therefore all POSIX threads, compete for CPU time together. For that reason the contention scope thread attribute is always **PTHREAD_SCOPE_SYSTEM**.

NO VXWORKS EQUIVALENT

Since there is no notion of a process (in the POSIX sense), there is no notion of sharing of locks (mutexes) and condition variables between processes. As a result, the POSIX symbol **_POSIX_THREAD_PROCESS_SHARED** is not defined in this implementation, and the routines **pthread_condattr_getpshared( )**, **pthread_condattr_setpshared( )**, **pthread_mutexattr_getpshared( )** are not implemented.

Also, since there are no processes in VxWorks, **fork( )**, **wait( )**, and **pthread_atfork( )** are unimplemented.

VxWorks does not have password, user, or group databases, therefore there are no implementations of **getlogin( )**, **getgrgid( )**, **getpwnam( )**, **getpwuid( )**, **getlogin_r( )**, **getgrgid_r( )**, **getpwnam_r( )**, and **getpwuid_r( )**.

SCHEDULING

The default scheduling policy for a created thread is inherited from the system setting at the time of creation.

Scheduling policies under VxWorks are global; they are not set per-thread, as the POSIX model describes. As a result, the *pthread* scheduling routines, as well as the POSIX scheduling routines native to VxWorks, do not allow you to change the scheduling policy. Under VxWorks you may set the scheduling policy in a thread, but if it does not match the system's scheduling policy, an error is returned.

The detailed explanation for why this error occurs is a bit convoluted: technically the scheduling policy is an attribute of a thread (in that there are **pthread_attr_getschedpolicy( )** and **pthread_attr_setschedpolicy( )** functions that define what the thread's scheduling policy will be once it is created, and not what any thread should do at the time they are called). A situation arises where the scheduling policy in force at the time of a thread's creation is not the same as set in its attributes. In this case **pthread_create( )** fails with an otherwise undocumented error **ENOTTY**.

The bottom line is that under VxWorks, if you wish to specify the scheduling policy of a thread, you must set the desired global scheduling policy to match. Threads must then adhere to that scheduling policy, or use the **PTHREAD_INHERIT_SCHED** mode to inherit the current mode and creator's priority.

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, **SIGCANCEL**, has been set aside in this version of VxWorks for this purpose. Applications should take care not to block or handle **SIGCANCEL**.

SUMMARY MATRIX

pthread function	Implemented?	Note(s)
pthread_attr_destroy	Yes	
pthread_attr_getdetachstate	Yes	
pthread_attr_getinheritsched	Yes	
pthread_attr_getschedparam	Yes	
pthread_attr_getschedpolicy	Yes	
pthread_attr_getscope	Yes	
pthread_attr_getstackaddr	Yes	
pthread_attr_getstacksize	Yes	
pthread_attr_init	Yes	
pthread_attr_setdetachstate	Yes	
pthread_attr_setinheritsched	Yes	
pthread_attr_setschedparam	Yes	
pthread_attr_setschedpolicy	Yes	
pthread_attr_setscope	Yes	2
pthread_attr_setstackaddr	Yes	
pthread_attr_setstacksize	Yes	
pthread_atfork	No	1
pthread_cancel	Yes	5
pthread_cleanup_pop	Yes	
pthread_cleanup_push	Yes	
pthread_condattr_destroy	Yes	
pthread_condattr_getpshared	No	3
pthread_condattr_init	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_getschedparam	Yes	4

P

pthread function	Implemented?	Note(s)
pthread_getspecific	Yes	
pthread_join	Yes	
pthread_key_create	Yes	
pthread_key_delete	Yes	
pthread_kill	Yes	
pthread_once	Yes	
pthread_self	Yes	
pthread_setcancelstate	Yes	
pthread_setcanceltype	Yes	
pthread_setschedparam	Yes	4
pthread_setspecific	Yes	
pthread_sigmask	Yes	
pthread_testcancel	Yes	
pthread_mutexattr_destroy	Yes	
pthread_mutexattr_getprioceiling	Yes	
pthread_mutexattr_getprotocol	Yes	
pthread_mutexattr_getpshared	No	3
pthread_mutexattr_init	Yes	
pthread_mutexattr_setprioceiling	Yes	
pthread_mutexattr_setprotocol	Yes	
pthread_mutexattr_setpshared	No	3
pthread_mutex_destroy	Yes	
pthread_mutex_getprioceiling	Yes	
pthread_mutex_init	Yes	
pthread_mutex_lock	Yes	
pthread_mutex_setprioceiling	Yes	
pthread_mutex_trylock	Yes	
pthread_mutex_unlock	Yes	
getlogin_r	No	6
getgrgid_r	No	6
getpwnam_r	No	6
getpwuid_r	No	6

NOTES

1 The **pthread_atfork( )** function is not implemented since **fork( )** is not implemented in VxWorks.

2 The contention scope thread scheduling attribute is always **PTHREAD_SCOPE_SYSTEM**, since threads (*i.e.*, tasks) contend for resources with all other threads in the system.

3 The routines **pthread_condattr_getpshared( )**, **pthread_attr_setpshared( )**, **pthread_mutexattr_getpshared( )** and **pthread_mutexattr_setpshared( )** are not

supported, since these interfaces describe how condition variables and mutexes relate to a process, and VxWorks does not implement a process model.

4 The default scheduling policy is inherited from the current system setting. The POSIX model of per-thread scheduling policies is not supported, since a basic tenet of the design of VxWorks is a system-wide scheduling policy.

5 Thread cancellation is supported in appropriate *pthread* routines and those routines already supported by VxWorks. However, the complete list of cancellation points specified by POSIX is not supported because routines such as **msync( )**, **fcntl( )**, **tcdrain( )**, and **wait( )** are not implemented by VxWorks.

6 The routines **getlogin_r( )**, **getgrgid_r( )**, **getpwnam_r( )**, and **getpwuid_r( )** are not implemented.

INCLUDE FILES **pthread.h**

SEE ALSO **taskLib**, **semMLib**, **semPxLib**, *VxWorks Programmer's Guide: Multitasking*

ptyDrv

NAME **ptyDrv** – pseudo-terminal driver

ROUTINES
ptyDrv( ) - initialize the pseudo-terminal driver
ptyDevCreate( ) - create a pseudo terminal
ptyDevRemove( ) - destroy a pseudo terminal
ptyShow( ) - show the state of the Pty Buffers

DESCRIPTION The pseudo-terminal driver provides a *tty*-like interface between a master and slave process, typically in network applications. The master process simulates the "hardware" side of the driver (*e.g.*, a USART serial chip), while the slave process is the application program that normally talks to the driver.

USER-CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. However, the following routines must be called directly: **ptyDrv( )** to initialize the driver, **ptyDevCreate( )** to create devices, and **ptyDevRemove( )** to remove an existing device.

INITIALIZING THE DRIVER

Before using the driver, it must be initialized by calling **ptyDrv( )**. This routine must be called before any reads, writes, or calls to **ptyDevCreate( )**.

P

CREATING PSEUDO-TERMINAL DEVICES

Before a pseudo-terminal can be used, it must be created by calling **ptyDevCreate( )**:

```
STATUS ptyDevCreate
    (
    char  *name,      /* name of pseudo terminal      */
    int   rdBufSize,  /* size of terminal read buffer */
    int   wrtBufSize  /* size of write buffer         */
    )
```

For instance, to create the device pair **/pty/0.M** and **/pty/0.S**, with read and write buffer sizes of 512 bytes, the proper call would be:

```
ptyDevCreate ("/pty/0.", 512, 512);
```

When **ptyDevCreate( )** is called, two devices are created, a master and slave. One is called *name*M and the other *name*S. They can then be opened by the master and slave processes. Data written to the master device can then be read on the slave device, and vice versa. Calls to **ioctl( )** may be made to either device, but they should only apply to the slave side, since the master and slave are the same device.

The **ptyDevRemove( )** routine will delete an existing pseudo-terminal device and reclaim the associated memory. Any file descriptors associated with the device will be closed.

IOCTL FUNCTIONS

Pseudo-terminal drivers respond to the same **ioctl( )** functions used by *tty* devices. These functions are defined in **ioLib.h** and documented in the manual entry for **tyLib**.

INCLUDE FILES

ioLib.h, **ptyDrv.h**

SEE ALSO

tyLib, *VxWorks Programmer's Guide: I/O System*

ramDiskCbio

NAME **ramDiskCbio** – RAM Disk Cached Block Driver

ROUTINES **ramDiskDevCreate( )** - Initialize a RAM Disk device

DESCRIPTION This module implements a RAM-disk driver with a CBIO interface which can be directly utilized by **dosFsLib** without the use of the Disk Cache module **dcacheCbio**. This results in an ultra-compact RAM footprint. This module is implemented using the CBIO API (see **cbioLib( )**)

This module is delivered in source as a functional example of a basic CBIO module.

WARNING: This module may be used for SRAM or other non-volatile RAM cards to store a file system, but that configuration will be susceptible to data corruption in events of system failure which are not normally observed with magnetic disks, *i.e.*, using this driver with an SRAM card can not guard against interruptions in midst of updating a particular sector, resulting in that sector become internally inconsistent.

SEE ALSO **dosFsLib**, **cbioLib**

ramDrv

NAME **ramDrv** – RAM disk driver

ROUTINES **ramDrv( )** - prepare a RAM disk driver for use (optional)
ramDevCreate( ) - create a RAM disk device

DESCRIPTION This driver emulates a disk driver, but actually keeps all data in memory. The memory location and size are specified when the “disk” is created. The RAM disk feature is useful when data must be preserved between boots of VxWorks or when sharing data between CPUs.

USER-CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. Two routines, however, can be called directly by the user. The first, **ramDrv( )**, provides no real function except to parallel the initialization function found in true disk device drivers. A call to **ramDrv( )** is not required to use the RAM disk driver. However, the second routine, **ramDevCreate( )**, must be called directly to create RAM disk devices.

Once the device has been created, it must be associated with a name and file system (dosFs, rt11Fs, or rawFs). This is accomplished by passing the value returned by **ramDevCreate( )**, a pointer to a block device structure, to the file system's device initialization routine or make-file-system routine. See the manual entry **ramDevCreate( )** for a more detailed discussion.

IOCTL FUNCTIONS
The RAM driver is called in response to **ioctl( )** codes in the same manner as a normal disk driver. When the file system is unable to handle a specific **ioctl( )** request, it is passed to the **ramDrv** driver. Although there is no physical device to be controlled, **ramDrv** does handle a **FIODISKFORMAT** request, which always returns **OK**. All other **ioctl( )** requests return an error and set the task's **errno** to **S_ioLib_UNKNOWN_REQUEST**.

INCLUDE FILE
ramDrv.h

SEE ALSO
dosFsDevInit( ), **dosFsMkfs( )**, **rt11FsDevInit( )**, **rt11FsMkfs( )**, **rawFsDevInit( )**, *VxWorks Programmer's Guide: I/O System, Local File Systems*

rawFsLib

NAME
rawFsLib – raw block device file system library

ROUTINES
rawFsDevInit( ) - associate a block device with raw volume functions
rawFsInit( ) - prepare to use the raw volume library
rawFsModeChange( ) - modify the mode of a raw device volume
rawFsReadyChange( ) - notify **rawFsLib** of a change in ready status
rawFsVolUnmount( ) - disable a raw device volume

DESCRIPTION
This library provides basic services for disk devices that do not use a standard file or directory structure. The disk volume is treated much like a large file. Portions of it may be read, written, or the current position within the disk may be changed. However, there is no high-level organization of the disk into files or directories.

USING THIS LIBRARY
The various routines provided by the VxWorks raw "file system" (rawFs) may be separated into three broad groups: general initialization, device initialization, and file system operation.

The **rawFsInit( )** routine is the principal initialization function; it need only be called once, regardless of how many rawFs devices will be used.

A separate rawFs routine is used for device initialization. For each rawFs device, **rawFsDevInit( )** must be called to install the device.

Several routines are provided to inform the file system of changes in the system environment. The **rawFsModeChange( )** routine may be used to modify the readability or writability of a particular device. The **rawFsReadyChange( )** routine is used to inform the file system that a disk may have been swapped and that the next disk operation should first remount the disk. The **rawFsVolUnmount( )** routine informs the file system that a particular device should be synchronized and unmounted, generally in preparation for a disk change.

INITIALIZATION

Before any other routines in **rawFsLib** can be used, **rawFsInit( )** must be called to initialize the library. This call specifies the maximum number of raw device file descriptors that can be open simultaneously and allocates memory for that many raw file descriptors. Any attempt to open more raw device file descriptors than the specified maximum will result in errors from **open( )** or **creat( )**.

During the **rawFsInit( )** call, the raw device library is installed as a driver in the I/O system driver table. The driver number associated with it is then placed in a global variable, **rawFsDrvNum**.

This initialization is enabled when the configuration macro **INCLUDE_RAWFS** is defined; **rawFsInit( )** is then called from the root task, **usrRoot( )**, in **usrConfig.c**.

DEFINING A RAW DEVICE

To use this library for a particular device, the device structure used by the device driver must contain, as the very first item, a CBIO device description structure (**CBIO_DEV**) or block device description structure (**BLK_DEV**). This must be initialized before calling **rawFsDevInit( )**.

The **rawFsDevInit( )** routine is used to associate a device with the **rawFsLib** functions. The *pVolName* parameter expected by **rawFsDevInit( )** is a pointer to a name string, to be used to identify the device. This will serve as the pathname for I/O operations which operate on the device. This name will appear in the I/O system device table, which may be displayed using **iosDevShow( )**.

The syntax of the **rawFsDevInit( )** routine is as follows:

```
rawFsDevInit
    (
    char     *pVolName, /* name to be used for volume - iosDevAdd  */
    BLK_DEV  *pDevice   /* pointer to BLK_DEV device or a CBIO_DEV_ID */
    )
```

Unlike the VxWorks DOS file system, raw volumes do not require an **FIODISKINIT ioctl( )** function to initialize volume structures. (Such an **ioctl( )** call can be made for a raw volume, but it has no effect.) As a result, there is no “make file system” routine for raw volumes (for comparison, see the manual entry for **rawFsMkfs( )**).

When **rawFsLib** receives a request from the I/O system, after **rawFsDevInit( )** has been called, it calls the appropriate device driver routines to access the device.

R

MULTIPLE LOGICAL DEVICES

The block number passed to the block read and write routines is an absolute number, starting from block 0 at the beginning of the device. If desired, the driver may add an offset from the beginning of the physical device before the start of the logical device. This would normally be done by keeping an offset parameter in the driver's device-specific structure, and adding the proper number of blocks to the block number passed to the read and write routines. See the **ramDrv** manual entry for an example.

UNMOUNTING VOLUMES (CHANGING DISKS)

A disk should be unmounted before it is removed. When unmounted, any modified data that has not been written to the disk will be written out. A disk may be unmounted by either calling **rawFsVolUnmount( )** directly or calling **ioctl( )** with a **FIODISKCHANGE** function code.

There may be open file descriptors to a raw device volume when it is unmounted. If this is the case, those file descriptors will be marked as obsolete. Any attempts to use them for further I/O operations will return an **S_rawFsLib_FD_OBSOLETE** error. To free such file descriptors, use the **close( )** call, as usual. This will successfully free the descriptor, but will still return **S_rawFsLib_FD_OBSOLETE**.

SYNCHRONIZING VOLUMES

A disk should be "synchronized" before it is unmounted. To synchronize a disk means to write out all buffered data (the write buffers associated with open file descriptors), so that the disk is updated. It may or may not be necessary to explicitly synchronize a disk, depending on how (or if) the driver issues the **rawFsVolUnmount( )** call.

When **rawFsVolUnmount( )** is called, an attempt will be made to synchronize the device before unmounting. However, if the **rawFsVolUnmount( )** call is made by a driver in response to a disk being removed, it is obviously too late to synchronize. Therefore, a separate **ioctl( )** call specifying the **FIOSYNC** function should be made before the disk is removed. (This could be done in response to an operator command.)

If the disk will still be present and writable when **rawFsVolUnmount( )** is called, it is not necessary to first synchronize the disk. In all other circumstances, failure to synchronize the volume before unmounting may result in lost data.

IOCTL FUNCTIONS

The VxWorks raw block device file system supports the following **ioctl( )** functions. The functions listed are defined in the header **ioLib.h**.

FIODISKFORMAT

No file system is initialized on the disk by this request. This ioctl is passed directly down to the driver-provided function:

```
fd = open ("DEV1:", O_WRONLY);
status = ioctl (fd, FIODISKFORMAT, 0);
```

FIODISKINIT

Initializes a raw file system on the disk volume. Since there are no file system structures, this functions performs no action. It is provided only for compatibility with other VxWorks file systems.

FIODISKCHANGE
Announces a media change. It performs the same function as **rawFsReadyChange( )**. This function may be called from interrupt level:

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

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

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

FIOGETNAME
Gets the file name of the file descriptor and copies it to the buffer *nameBuf*:

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

FIOSEEK
Sets the current byte offset on the disk to the position specified by *newOffset*:

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

FIOWHERE
Returns the current byte position from the start of the device for the specified file descriptor. This is the byte offset of the next byte to be read or written. It takes no additional argument:

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

FIOFLUSH
Writes all modified file descriptor buffers to the physical device.

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

FIOSYNC
Performs the same function as **FIOFLUSH**.

FIONREAD
Copies to *unreadCount* the number of bytes from the current file position to the end of the device:

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

INCLUDE FILES **rawFsLib.h**

SEE ALSO **ioLib**, **iosLib**, **rawFsLib**, **ramDrv**, *VxWorks Programmer's Guide: I/O System, Local File Systems*

rBuffLib

NAME **rBuffLib** – dynamic ring buffer (rBuff) library

ROUTINES **wvRBuffMgrPrioritySet( )** - set the priority of the WindView rBuff manager (WindView)

DESCRIPTION This library contains a routine for changing the default priority of the rBuff manager task.

SEE ALSO **memLib**, **rngLib**, *VxWorks Programmer's Guide: Basic OS*

rdiscLib

NAME **rdiscLib** – ICMP router discovery server library

ROUTINES

rdiscLibInit( ) - Initialize router discovery
rdiscInit( ) - initialize the ICMP router discovery function
sendAdvert( ) - send an advertisement to one location
sendAdvertAll( ) - send an advertisement to all active locations
rdiscTimerEvent( ) - called after watchdog timeout
rdisc( ) - implement the ICMP router discovery function
rdCtl( ) - implement the ICMP router discovery control function
rdiscIfReset( ) - check for new or removed interfaces for router discovery

DESCRIPTION **rdiscLib** contains code to implement ICMP Router Discovery. This feature allows routers to advertise an address to the hosts on each of the routers interfaces. This address is placed by the host into its route table as a default router. A host may also solicit the address by multicasting the request to the **ALL_ROUTERS** address (224.0.0.2), to which a router would respond with a unicast version of the advertisement.

There are three routines in this implementation of router discovery: **rdiscInit( )**, **rdisc( )** and **rdCtl( )**. **rdiscInit( )** is the initialization routine, **rdisc( )** handles the periodic transmission of advertisements and processing of solicitations, and **rdCtl( )** sets/gets user parameters.

rebootLib

NAME **rebootLib** – reboot support library

ROUTINES **reboot( )** - reset network devices and transfer control to boot ROMs
rebootHookAdd( ) - add a routine to be called at reboot

DESCRIPTION This library provides reboot support. To restart VxWorks, the routine **reboot( )** can be called at any time by typing CTRL-X from the shell. Shutdown routines can be added with **rebootHookAdd( )**. These are typically used to reset or synchronize hardware. For example, **netLib** adds a reboot hook to cause all network interfaces to be reset. Once the reboot hooks have been run, **sysToMonitor( )** is called to transfer control to the boot ROMs. For more information, see the manual entry for **bootInit**.

DEFICIENCIES The order in which hooks are added is the order in which they are run. As a result, **netLib** will kill the network, and no user-added hook routines will be able to use the network. There is no **rebootHookDelete( )** routine.

INCLUDE FILES **rebootLib.h**

SEE ALSO **sysLib**, **bootConfig**, **bootInit**

remLib

R

NAME **remLib** – remote command library

ROUTINES **rcmd( )** - execute a shell command on a remote machine
rresvport( ) - open a socket with a privileged port bound to it
remCurIdGet( ) - get the current user name and password
remCurIdSet( ) - set the remote user name and password
iam( ) - set the remote user name and password
whoami( ) - display the current remote identity
bindresvport( ) - bind a socket to a privileged IP port

DESCRIPTION This library provides routines that support remote command functions. The **rcmd( )** and **rresvport( )** routines use protocols implemented in BSD 4.3; they support remote command execution, and the opening of a socket with a bound privileged port, respectively. For more information, see *Unix Network Programming* by W. Richard Stevens. This library also includes routines that authorize network file access via **netDrv**.

To include **remLib** in a VxWorks image, include the **NETWRS_REMLIB** configuration component. This component contains one parameter, **RSH_STDERR_SETUP_TIMEOUT**. Use this parameter to specify how long an **rcmd( )** call should wait for a return from its internal call to **select( )**. Valid values for **RSH_STDERR_SETUP_TIMEOUT** are 0 (**NO_WAIT**), -1 (**WAIT_FOREVER**), or a positive integer from 1 to 2147483647 inclusive. This positive integer specifies the wait in seconds. The default value for **RSH_STDERR_SETUP_TIMEOUT** is -1 (**WAIT_FOREVER**).

INCLUDE FILES **remLib.h**

SEE ALSO **inetLib**

remShellLib

NAME **remShellLib** – remote access to target shell

ROUTINES No Callable Routines

DESCRIPTION This library contains the support routines for remote access to the VxWorks target shell for clients using the **telnet** or **rlogin** protocols. It supplies file descriptors to connection **telnet** or **rlogin** sessions to the shell's command interpreter.

INCLUDE FILES **remShellLib.h**, **shellLib.h**

resolvLib

NAME **resolvLib** – DNS resolver library

ROUTINES

resolvInit( ) - initialize the resolver library
resolvGetHostByName( ) - query the DNS server for the IP address of a host
resolvGetHostByAddr( ) - query the DNS server for the host name of an IP address
resolvParamsSet( ) - set the parameters which control the resolver library
resolvParamsGet( ) - get the parameters which control the resolver library
resolvDNExpand( ) - expand a DNS compressed name from a DNS packet
resolvDNComp( ) - compress a DNS name in a DNS packet
resolvQuery( ) - construct a query, send it, wait for a response
resolvMkQuery( ) - create all types of DNS queries
resolvSend( ) - send a pre-formatted query and return the answer

DESCRIPTION

This library provides the client-side services for DNS (Domain Name Service) queries. DNS queries come from applications that require translation of IP addresses to host names and back. If you include this library in VxWorks, it extends the services of the host library. The interface to this library is described in **hostLib**. The **hostLib** interface uses resolver services to get IP and host names. In addition, the resolver can query multiple DNS servers, if necessary, to add redundancy for queries.

There are two interfaces available for the resolver library. One is a high-level interface suitable for most applications. The other is also a low-level interface for more specialized applications, such as mail protocols.

USING THIS LIBRARY

By default, a VxWorks build does not include the resolver code. In addition, VxWorks is delivered with the resolver library disabled. To include the resolver library in the VxWorks image, edit **config/all/configAll.h** and include the definition:

```
#define INCLUDE_DNS_RESOLVER
```

To enable the resolver services, you need to redefine only one DNS server IP address, changing it from a place-holder value to an actual value. Additional DNS server IP addresses can be configured using **resolvParamsSet( )**. To do the initial configuration, edit **configAll.h**, and enter the correct IP address for your domain server in the definition:

```
#define RESOLVER_DOMAIN_SERVER  "90.0.0.3"
```

If you do not provide a valid IP address, resolver initialization fails. You also need to configure the domain to which your resolver belongs. To do this, edit **configAll.h** and enter the correct domain name for your organization in the definition:

```
#define RESOLVER_DOMAIN  "wrs.com"
```

The last and most important step is to make sure that you have a route to the configured DNS server. If your VxWorks image includes a routing protocol, such as RIP or OSPF, the routes are created for you automatically. Otherwise, you must use **routeAdd( )** or **mRouteAdd( )** to add the routes to the routing table.

The resolver library comes with a debug option. To turn on debugging, edit **configAll.h** to include the define:

```
#define INCLUDE_DNS_DEBUG
```

This include makes VxWorks print a log of the resolver queries to the console. This feature assumes a single task. Thus, if you are running multiple tasks, your output to the console is a garble of messages from all the tasks.

The resolver library uses UDP to send queries to the DNS server and expects the DNS server to handle recursion. You can change the resolver parameters at any time after the library has been initialized with **resolvInit( )**. However, it is strongly recommended that you change parameters only shortly after initialization, or when there are no other tasks accessing the resolver library.

Your procedure for changing any of the resolver parameter should start with a call to **resolvParamsGet( )** to retrieve the active parameters. Then you can change the query order (defaults to query DNS server only), the domain name, or add DNS server IP addresses. After the parameters are changed, call **resolvParamsSet( )**. For the values you can use when accessing resolver library services, see the header files **resolvLib.h**, **resolv/resolv.h**, and **resolv/nameser.h**.

INCLUDE FILES **resolvLib.h**

SEE ALSO **hostLib**

ripLib

NAME **ripLib** – Routing Information Protocol (RIP) v1 and v2 library

ROUTINES

ripLibInit( ) - initialize the RIP routing library
ripAddrsXtract( ) - extract socket address pointers from the route message
ripRouteShow( ) - display the internal routing table maintained by RIP
ripIfShow( ) - display the internal interface table maintained by RIP
ripAuthHookAdd( ) - add an authentication hook to a RIP interface
ripAuthHookDelete( ) - remove an authentication hook from a RIP interface
ripAuthHook( ) - sample authentication hook
ripLeakHookAdd( ) - add a hook to bypass the RIP and kernel routing tables
ripLeakHookDelete( ) - remove a table bypass hook from a RIP interface
ripSendHookAdd( ) - add an update filter to a RIP interface
ripSendHookDelete( ) - remove an update filter from a RIP interface
ripRouteHookAdd( ) - add a hook to install static and non-RIP routes into RIP
ripRouteHookDelete( ) - remove the route hook
ripIfSearch( ) - add new interfaces to the internal list
ripIfReset( ) - alter the RIP configuration after an interface changes
ripFilterEnable( ) - activate strict border gateway filtering
ripFilterDisable( ) - prevent strict border gateway filtering
ripShutdown( ) - terminate all RIP processing
ripDebugLevelSet( ) - specify amount of debugging output
ripAuthKeyShow( ) - show current authentication configuration
ripAuthKeyAdd( ) - add a new RIP authentication key
ripAuthKeyDelete( ) - delete an existing RIP authentication key
ripAuthKeyFind( ) - find a RIP authentication key
ripAuthKeyFindFirst( ) - find a RIP authentication key
ripAuthKeyInMD5( ) - authenticate an incoming RIP-2 message using MD5
ripAuthKeyOut1MD5( ) - start MD5 authentication of an outgoing RIP-2 message
ripAuthKeyOut2MD5( ) - authenticate an outgoing RIP-2 message using MD5
ripIfExcludeListAdd( ) - Add an interface to the RIP exclusion list

ripIfExcludeListDelete( ) - Delete an interface from RIP exclusion list
ripIfExcludeListShow( ) - Show the RIP interface exclusion list

DESCRIPTION This library implements versions 1 and 2 of the Routing Information Protocol (RIP). The protocol is intended to operate as an interior gateway protocol within a relatively small network with a longest path of 15 hops.

HIGH-LEVEL INTERFACE

The **ripLibInit( )** routine links this library into the VxWorks image and begins a RIP session. This happens automatically if **INCLUDE_RIP** is defined at the time the image is built. Once started, RIP will maintain the network routing table until deactivated by a call to the **ripShutdown( )** routine, which will remove all route entries and disable the RIP library routines. All RIP requests and responses are handled as defined in the RFC specifications. RFC 1058 defines the basic protocol operation and RFC 1723 details the extensions that constitute version 2.

When acting as a supplier, outgoing route updates are filtered using simple split horizon. Split horizon with poisoned reverse is not currently available. Additional route entries may be excluded from the periodic update with the **ripSendHookAdd( )** routine.

If a RIP session is terminated, the networking subsystem may not function correctly until RIP is restarted with a new call to **ripLibInit( )** unless routing information is provided by some other method.

CONFIGURATION INTERFACE

By default, a RIP session only uses the network interfaces created before it started. The **ripIfSearch( )** routine allows RIP to recognize any interfaces added to the system after that point. If the address or netmask of an existing interface is changed during a RIP session, the **ripIfReset( )** routine must be used to update the RIP configuration appropriately. The current RIP implementation also automatically performs the border gateway filtering required by the RFC specification. Those restrictions provide correct operation in a mixed environment of RIP-1 and RIP-2 routers. The **ripFilterDisable( )** routine will remove those limitations, and can produce more efficient routing for some topologies. However, you must not use that routine if any version 1 routers are present. The **ripFilterEnable( )** routine will restore the default behavior.

AUTHENTICATION INTERFACE

By default, authentication is disabled, but may be activated by an SNMP agent on an interface-specific basis. While authentication is disabled, any RIP-2 messages containing authentication entries are discarded. When enabled, all RIP-2 messages without authentication entries are automatically rejected. To fully support authentication, an authentication routine should be specified with the **ripAuthHookAdd( )** routine. The specified function will be called to screen every RIP-1 message and all unverified RIP-2 messages containing authentication entries. It may be removed with the **ripAuthHookDelete( )** routine. All RIP-1 and unverified RIP-2 messages will be discarded while authentication is enabled unless a hook is present.

OPTIONAL INTERFACE

The **ripLeakHookAdd( )** routine allows the use of an alternative routing protocol that uses RIP as a transport mechanism. The specified function can prevent the RIP session from creating any table entries from the received messages. The **ripLeakHookDelete( )** routine will restore the default operation.

DEBUGGING INTERFACE

As required by the RFC specification, the obsolete **traceon** and **traceoff** messages are not supported by this implementation. The **ripRouteShow( )** routine will display the contents of the internal RIP routing table. Routines such as **mRouteShow( )** to display the corresponding kernel routing table will also be available if **INCLUDE_NET_SHOW** is defined when the image is built. If additional information is required, the **ripDebugLevelSet( )** routine will enable predefined debugging messages that will be sent to the standard output.

INCLUDE FILES **ripLib.h**

SEE ALSO RFC 1058, RFC 1723

rlogLib

NAME **rlogLib** – remote login library

ROUTINES

rlogInit( ) - initialize the remote login facility
rlogind( ) - the VxWorks remote login daemon
rlogin( ) - log in to a remote host

DESCRIPTION

This library provides a remote login facility for VxWorks based on the UNIX **rlogin** protocol (as implemented in UNIX BSD 4.3). On a VxWorks terminal, this command gives users the ability to log in to remote systems on the network.

Reciprocally, the remote login daemon, **rlogind( )**, allows remote users to log in to VxWorks. The daemon is started by calling **rlogInit( )**, which is called automatically when **INCLUDE_RLOGIN** is defined. The remote login daemon accepts remote login requests from another VxWorks or UNIX system, and causes the shell's input and output to be redirected to the remote user.

Internally, **rlogind( )** provides a *tty*-like interface to the remote user through the use of the VxWorks pseudo-terminal driver **ptyDrv**.

INCLUDE FILES **rlogLib.h**

SEE ALSO **ptyDrv**, **telnetLib**, UNIX BSD 4.3 manual entries for **rlogin**, **rlogind**, and **pty**

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**

routeEntryLib

NAME **routeEntryLib** – route interface library for multiple matching entries

ROUTINES
routeModify( ) - change an entry in the routing table
routeEntryAdd( ) - insert a route in the routing table
routeEntryDel( ) - remove a route from the routing table
routeTableWalk( ) - traverse the IP routing table
routeEntryLookup( ) - find a matching route for a destination

routeLib

NAME **routeLib** – network route manipulation library

ROUTINES
routeAdd( ) - add a route
routeNetAdd( ) - add a route to a destination that is a network
routeDelete( ) - delete a route
mRouteAdd( ) - add multiple routes to the same destination
mRouteEntryAdd( ) - add a protocol-specific route to the routing table
mRouteEntryDelete( ) - delete route from the routing table
mRouteDelete( ) - delete a route from the routing table

DESCRIPTION This library contains the routines for inspecting the routing table, as well as routines for adding and deleting routes from that table. If you do not configure VxWorks to include a routing protocol, such as RIP or OSPF, you can use these routines to maintain the routing tables manually.

To use this feature, include the following component: **INCLUDE_NETWRS_ROUTELIB**

INCLUDE FILES **routeLib.h**

SEE ALSO **hostLib**

routeMessageLib

NAME **routeMessageLib** – message routines for the routing interface library

ROUTINES
routeStorageUnbind( ) - remove a registered handler from the routing system

rpcLib

NAME
rpcLib – Remote Procedure Call (RPC) support library

ROUTINES
rpcInit( ) - initialize the RPC package
rpcTaskInit( ) - initialize a task's access to the RPC package

DESCRIPTION
This library supports Sun Microsystems' Remote Procedure Call (RPC) facility. RPC provides facilities for implementing distributed client/server-based architectures. The underlying communication mechanism can be completely hidden, permitting applications to be written without any reference to network sockets. The package is structured such that lower-level routines can optionally be accessed, allowing greater control of the communication protocols.

For more information and a tutorial on RPC, see Sun Microsystems' *Remote Procedure Call Programming Guide.* For an example of RPC usage, see **/target/unsupported/demo/sprites**.

The RPC facility is enabled when **INCLUDE_RPC** is defined.

VxWorks supports Network File System (NFS), which is built on top of RPC. If NFS is configured into the VxWorks system, RPC is automatically included as well.

IMPLEMENTATION
A task must call **rpcTaskInit( )** before making any calls to other routines in the RPC library. This routine creates task-specific data structures required by RPC. These task-specific data structures are automatically deleted when the task exits.

Because each task has its own RPC context, RPC-related objects (such as SVCXPRTs and CLIENTs) cannot be shared among tasks; objects created by one task cannot be passed to another for use. Such additional objects must be explicitly deleted (for example, using task deletion hooks).

INCLUDE FILES
rpc.h

SEE ALSO
nfsLib, **nfsDrv**, Sun Microsystems' *Remote Procedure Call Programming Guide*

rt11FsLib

NAME
rt11FsLib – RT-11 media-compatible file system library

ROUTINES
rt11FsDevInit( ) - initialize the rt11Fs device descriptor
rt11FsInit( ) - prepare to use the rt11Fs library
rt11FsMkfs( ) - initialize a device and create an rt11Fs file system

rt11FsDateSet( ) - set the rt11Fs file system date
rt11FsReadyChange( ) - notify rt11Fs of a change in ready status
rt11FsModeChange( ) - modify the mode of an rt11Fs volume

DESCRIPTION

This library provides services for file-oriented device drivers which use the RT-11 file standard. This module takes care of all the necessary buffering, directory maintenance, and RT-11-specific details.

USING THIS LIBRARY

The various routines provided by the VxWorks RT-11 file system (rt11Fs) may be separated into three broad groups: general initialization, device initialization, and file system operation.

The **rt11FsInit( )** routine is the principal initialization function; it need only be called once, regardless of how many rt11Fs devices will be used.

Other rt11Fs routines are used for device initialization. For each rt11Fs device, either **rt11FsDevInit( )** or **rt11FsMkfs( )** must be called to install the device and define its configuration.

Several functions are provided to inform the file system of changes in the system environment. The **rt11FsDateSet( )** routine is used to set the date. The **rt11FsModeChange( )** routine is used to modify the readability or writability of a particular device. The **rt11FsReadyChange( )** routine is used to inform the file system that a disk may have been swapped, and that the next disk operation should first remount the disk.

INITIALIZING RT11FSLIB

Before any other routines in **rt11FsLib** can be used, **rt11FsInit( )** must be called to initialize this library. This call specifies the maximum number of rt11Fs files that can be open simultaneously and allocates memory for that many rt11Fs file descriptors. Attempts to open more files than the specified maximum will result in errors from **open( )** or **creat( )**.

This initialization is enabled when the configuration macro **INCLUDE_RT11FS** is defined.

DEFINING AN RT-11 DEVICE

To use this library for a particular device, the device structure must contain, as the very first item, a **BLK_DEV** structure. This must be initialized before calling **rt11FsDevInit( )**. In the **BLK_DEV** structure, the driver includes the addresses of five routines which it must supply: one that reads one or more sectors, one that writes one or more sectors, one that performs I/O control on the device (using **ioctl( )**), one that checks the status of the device, and one that resets the device. This structure also specifies various physical aspects of the device (*e.g.*, number of sectors, sectors per track, whether the media is removable). For more information about defining block devices, see the *VxWorks Programmer's Guide: I/O System*.

The device is associated with the rt11Fs file system by the **rt11FsDevInit( )** call. The arguments to **rt11FsDevInit( )** include the name to be used for the rt11Fs volume, a pointer to the **BLK_DEV** structure, whether the device uses RT-11 standard skew and interleave, and the maximum number of files that can be contained in the device directory.

Thereafter, when the file system receives a request from the I/O system, it simply calls the provided routines in the device driver to fulfill the request.

RTFMT

The RT-11 standard defines a peculiar software interleave and track-to-track skew as part of the format. The *rtFmt* parameter passed to **rt11FsDevInit( )** should be **TRUE** if this formatting is desired. This should be the case if strict RT-11 compatibility is desired, or if files must be transferred between the development and target machines using the VxWorks-supplied RT-11 tools. Software interleave and skew will automatically be dealt with by **rt11FsLib**.

When *rtFmt* has been passed as **TRUE** and the maximum number of files is specified **RT_FILES_FOR_2_BLOCK_SEG**, the driver does not need to do anything else to maintain RT-11 compatibility (except to add the track offset as described above).

Note that if the number of files specified is different than **RT_FILES_FOR_2_BLOCK_SEG** under either a VxWorks system or an RT-11 system, compatibility is lost because VxWorks allocates a contiguous directory, whereas RT-11 systems create chained directories.

MULTIPLE LOGICAL DEVICES AND RT-11 COMPATIBILITY

The sector number passed to the sector read and write routines is an absolute number, starting from sector 0 at the beginning of the device. If desired, the driver may add an offset from the beginning of the physical device before the start of the logical device. This would normally be done by keeping an offset parameter in the device-specific structure of the driver, and adding the proper number of sectors to the sector number passed to the read and write routines.

The RT-11 standard defines the disk to start on track 1. Track 0 is set aside for boot information. Therefore, in order to retain true compatibility with RT-11 systems, a one-track offset (*i.e.*, the number of sectors in one track) needs to be added to the sector numbers passed to the sector read and write routines, and the device size needs to be declared as one track smaller than it actually is. This must be done by the driver using **rt11FsLib**; the library does not add such an offset automatically.

In the VxWorks RT-11 implementation, the directory is a fixed size, able to contain at least as many files as specified in the call to **rt11FsDevInit( )**. If the maximum number of files is specified to be **RT_FILES_FOR_2_BLOCK_SEG**, strict RT-11 compatibility is maintained, because this is the initial allocation in the RT-11 standard.

RT-11 FILE NAMES

File names in the RT-11 file system use six characters, followed by a period (.), followed by an optional three-character extension.

DIRECTORY ENTRIES

An **ioctl( )** call with the **FIODIRENTRY** function returns information about a particular directory entry. A pointer to a **REQ_DIR_ENTRY** structure is passed as the parameter. The field **entryNum** in the **REQ_DIR_ENTRY** structure must be set to the desired entry number. The name of the file, its size (in bytes), and its creation date are returned in the structure. If the specified entry is empty (*i.e.*, if it represents an unallocated section of the disk), the name will be an empty string, the size will be the size of the available disk section, and the date will be meaningless. Typically, the entries are accessed sequentially, starting with **entryNum** = 0, until the terminating entry is reached, indicated by a return code of **ERROR**.

DIRECTORIES IN MEMORY

A copy of the directory for each volume is kept in memory (in the **RT_VOL_DESC** structure). This speeds up directory accesses, but requires that **rt11FsLib** be notified when disks are changed (*i.e.*, floppies are swapped). If the driver can find this out (by interrogating controller status or by receiving an interrupt), the driver simply calls **rt11FsReadyChange( )** when a disk is inserted or removed. The library **rt11FsLib** will automatically try to remount the device next time it needs it.

If the driver does not have access to the information that disk volumes have been changed, the *changeNoWarn* parameter should be set to **TRUE** when the device is defined using **rt11FsDevInit( )**. This will cause the disk to be automatically remounted before each **open( )**, **creat( )**, **delete( )**, and directory listing.

The routine **rt11FsReadyChange( )** can also be called by user tasks, by issuing an **ioctl( )** call with **FIODISKCHANGE** as the function code.

ACCESSING THE RAW DISK

As a special case in **open( )** and **creat( )** calls, **rt11FsLib** recognizes a **NULL** file name to indicate access to the entire “raw” disk, as opposed to a file on the disk. Access in raw mode is useful for a disk that has no file system. For example, to initialize a new file system on the disk, use an **ioctl( )** call with **FIODISKINIT**. To read the directory of a disk for which no file names are known, open the raw disk and use an **ioctl( )** call with the function **FIODIRENTRY**.

HINTS

The RT-11 file system is much simpler than the more common UNIX or MS-DOS file systems. The advantage of RT-11 is its speed; file access is made in at most one seek because all files are contiguous. Some of the most common errors for users with a UNIX background are:

Only a single create at a time may be active per device.

File size is set by the first create and close sequence; use **lseek( )** to ensure a specific file size; there is no append function to expand a file.

Files are strictly block oriented; unused portions of a block are filled with NULLs -- there is no end-of-file marker other than the last block.

IOCTL FUNCTIONS The rt11Fs file system supports the following **ioctl( )** functions. The functions listed are defined in the header **ioLib.h**. Unless stated otherwise, the file descriptor used for these functions can be any file descriptor open to a file or to the volume itself.

FIODISKFORMAT
Formats the entire disk with appropriate hardware track and sector marks. No file system is initialized on the disk by this request. Note that this is a driver-provided function:

```
fd = open ("DEV1:", O_WRONLY);
status = ioctl (fd, FIODISKFORMAT, 0);
```

FIODISKINIT
Initializes an rt11Fs file system on the disk volume. This routine does not format the disk; formatting must be done by the driver. The file descriptor should be obtained by opening the entire volume in raw mode:

```
fd = open ("DEV1:", O_WRONLY);
status = ioctl (fd, FIODISKINIT, 0);
```

FIODISKCHANGE
Announces a media change. It performs the same function as **rt11FsReadyChange( )**. This function may be called from interrupt level:

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

FIOGETNAME
Gets the file name of the file descriptor and copies it to the buffer *nameBuf*:

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

FIORENAME
Renames the file to the string *newname*:

```
status = ioctl (fd, FIORENAME, "newname");
```

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

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

FIOFLUSH
Flushes the file output buffer. It guarantees that any output that has been requested is actually written to the device.

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

FIOSEEK
Sets the current byte offset in the file to the position specified by *newOffset*:

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

FIOWHERE
Returns the current byte position in the file. This is the byte offset of the next byte to

be read or written. It takes no additional argument:

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

FIOSQUEEZE
Coalesces fragmented free space on an rt11Fs volume:

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

FIODIRENTRY
Copies information about the specified directory entries to a **REQ_DIR_ENTRY** structure that is defined in **ioLib.h**. The argument *req* is a pointer to a **REQ_DIR_ENTRY** structure. On entry, the structure contains the number of the directory entry for which information is requested. On return, the structure contains the information on the requested entry. For example, after the following:

```
REQ_DIR_ENTRY req;
req.entryNum = 0;
status = ioctl (fd, FIODIRENTRY, &req);
```

the request structure contains the name, size, and creation date of the file in the first entry (0) of the directory.

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

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

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

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

Any other **ioctl( )** function codes are passed to the block device driver for handling.

INCLUDE FILES **rt11FsLib.h**

SEE ALSO **ioLib**, **iosLib**, **ramDrv**, *VxWorks Programmer's Guide: I/O System, Local File Systems*

schedPxLib

NAME

schedPxLib – scheduling library (POSIX)

ROUTINES

sched_setparam( ) - set a task's priority (POSIX)
sched_getparam( ) - get the scheduling parameters for a specified task (POSIX)
sched_setscheduler( ) - set scheduling policy and scheduling parameters (POSIX)
sched_getscheduler( ) - get the current scheduling policy (POSIX)
sched_yield( ) - relinquish the CPU (POSIX)
sched_get_priority_max( ) - get the maximum priority (POSIX)
sched_get_priority_min( ) - get the minimum priority (POSIX)
sched_rr_get_interval( ) - get the current time slice (POSIX)

DESCRIPTION

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

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

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

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

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

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

INCLUDE FILES

sched.h

SEE ALSO

POSIX 1003.1b document, **taskLib**

S

scsi1Lib

NAME

scsi1Lib – Small Computer System Interface (SCSI) library (SCSI-1)

ROUTINES

No Callable Routines

DESCRIPTION

This library implements the Small Computer System Interface (SCSI) protocol in a controller-independent manner. It implements only the SCSI initiator function; the library does not support a VxWorks target acting as a SCSI target. Furthermore, in the current implementation, a VxWorks target is assumed to be the only initiator on the SCSI bus, although there may be multiple targets (SCSI peripherals) on the bus.

The implementation is transaction based. A transaction is defined as the selection of a SCSI device by the initiator, the issuance of a SCSI command, and the sequence of data, status, and message phases necessary to perform the command. A transaction normally completes with a "Command Complete" message from the target, followed by disconnection from the SCSI bus. If the status from the target is "Check Condition," the transaction continues; the initiator issues a "Request Sense" command to gain more information on the exception condition reported.

Many of the subroutines in **scsi1Lib** facilitate the transaction of frequently used SCSI commands. Individual command fields are passed as arguments from which SCSI Command Descriptor Blocks are constructed, and fields of a **SCSI_TRANSACTION** structure are filled in appropriately. This structure, along with the **SCSI_PHYS_DEV** structure associated with the target SCSI device, is passed to the routine whose address is indicated by the **scsiTransact** field of the **SCSI_CTRL** structure associated with the relevant SCSI controller.

The function variable **scsiTransact** is set by the individual SCSI controller driver. For off-board SCSI controllers, this routine rearranges the fields of the **SCSI_TRANSACTION** structure into the appropriate structure for the specified hardware, which then carries out the transaction through firmware control. Drivers for an on-board SCSI-controller chip can use the **scsiTransact( )** routine in **scsiLib** (which invokes the **scsi1Transact( )** routine in **scsi1Lib**), as long as they provide the other functions specified in the **SCSI_CTRL** structure.

Note that no disconnect/reconnect capability is currently supported.

SUPPORTED SCSI DEVICES

The **scsi1Lib** library supports use of SCSI peripherals conforming to the standards specified in *Common Command Set (CCS) of the SCSI, Rev. 4.B.* Most SCSI peripherals currently offered support CCS. While an attempt has been made to have **scsi1Lib** support non-CCS peripherals, not all commands or features of this library are guaranteed to work with them. For example, auto-configuration may be impossible with non-CCS devices, if they do not support the INQUIRY command.

Not all classes of SCSI devices are supported. However, the **scsiLib** library provides the capability to transact any SCSI command on any SCSI device through the **FIOSCSICOMMAND** function of the **scsiIoctl( )** routine.

Only direct-access devices (disks) are supported by a file system. For other types of devices, additional, higher-level software is necessary to map user-level commands to SCSI transactions.

CONFIGURING SCSI CONTROLLERS

The routines to create and initialize a specific SCSI controller are particular to the controller and normally are found in its library module. The normal calling sequence is:

```
xxCtrlCreate (...); /* parameters are controller specific */
xxCtrlInit (...);   /* parameters are controller specific */
```

The conceptual difference between the two routines is that **xxCtrlCreate( )** calloc's memory for the **xx_SCSI_CTRL** data structure and initializes information that is never expected to change (for example, clock rate). The remaining fields in the **xx_SCSI_CTRL** structure are initialized by **xxCtrlInit( )** and any necessary registers are written on the SCSI controller to effect the desired initialization. This routine can be called multiple times, although this is rarely required. For example, the bus ID of the SCSI controller can be changed without rebooting the VxWorks system.

CONFIGURING PHYSICAL SCSI DEVICES

Before a device can be used, it must be "created," that is, declared. This is done with **scsiPhysDevCreate( )** and can only be done after a **SCSI_CTRL** structure exists and has been properly initialized.

```
SCSI_PHYS_DEV *scsiPhysDevCreate
    (
    SCSI_CTRL * pScsiCtrl,/* ptr to SCSI controller info */
    int  devBusId,        /* device's SCSI bus ID */
    int  devLUN,          /* device's logical unit number */
    int  reqSenseLength,  /* length of REQUEST SENSE data dev returns */
    int  devType,         /* type of SCSI device */
    BOOL removable,       /* whether medium is removable */
    int  numBlocks,       /* number of blocks on device */
    int  blockSize        /* size of a block in bytes */
    )
```

Several of these parameters can be left unspecified, as follows:

reqSenseLength
: If 0, issue a **REQUEST_SENSE** to determine a request sense length.

devType
: If -1, issue an INQUIRY to determine the device type.

numBlocks, *blockSize*
: If 0, issue a **READ_CAPACITY** to determine the number of blocks.

The above values are recommended, unless the device does not support the required commands, or other non-standard conditions prevail.

LOGICAL PARTITIONS ON BLOCK DEVICES

It is possible to have more than one logical partition on a SCSI block device. This capability is currently not supported for removable media devices. A partition is an array of contiguously addressed blocks with a specified starting block address and a specified number of blocks. The **scsiBlkDevCreate( )** routine is called once for each block device partition. Under normal usage, logical partitions should not overlap.

```
SCSI_BLK_DEV *scsiBlkDevCreate
    (
    SCSI_PHYS_DEV *  pScsiPhysDev,  /* ptr to SCSI physical device info */
    int              numBlocks,     /* number of blocks in block device */
    int              blockOffset    /* address of first block in volume */
    )
```

Note that if *numBlocks* is 0, the rest of the device is used.

ATTACHING FILE SYSTEMS TO LOGICAL PARTITIONS

Files cannot be read or written to a disk partition until a file system (such as dosFs or rt11Fs) has been initialized on the partition. For more information, see the documentation in **dosFsLib** or **rt11FsLib**.

TRANSMITTING ARBITRARY COMMANDS TO SCSI DEVICES

The **scsi1Lib** library provides routines that implement many common SCSI commands. Still, there are situations that require commands that are not supported by **scsi1Lib** (for example, writing software to control non-direct access devices). Arbitrary commands are handled with the **FIOSCSICOMMAND** option to **scsiIoctl( )**. The *arg* parameter for **FIOSCSICOMMAND** is a pointer to a valid **SCSI_TRANSACTION** structure. Typically, a call to **scsiIoctl( )** is written as a subroutine of the form:

```
STATUS myScsiCommand
    (
    SCSI_PHYS_DEV *  pScsiPhysDev,  /* ptr to SCSI physical device     */
    char *           buffer,        /* ptr to data buffer              */
    int              bufLength,     /* length of buffer in bytes       */
    int              someParam      /* param. specifiable in cmd block */
    )
    {
    SCSI_COMMAND myScsiCmdBlock;        /* SCSI command byte array */
    SCSI_TRANSACTION myScsiXaction;     /* info on a SCSI transaction */
    /* fill in fields of SCSI_COMMAND structure */
    myScsiCmdBlock [0] = MY_COMMAND_OPCODE;     /* the required opcode */
```

```
.
myScsiCmdBlock [X] = (UINT8) someParam;     /* for example */
.
myScsiCmdBlock [N-1] = MY_CONTROL_BYTE;     /* typically == 0 */
/* fill in fields of SCSI_TRANSACTION structure */
myScsiXaction.cmdAddress    = myScsiCmdBlock;
myScsiXaction.cmdLength     = <# of valid bytes in myScsiCmdBlock>;
myScsiXaction.dataAddress   = (UINT8 *) buffer;
myScsiXaction.dataDirection = <O_RDONLY (0) or O_WRONLY (1)>;
myScsiXaction.dataLength    = bufLength;
myScsiXaction.cmdTimeout    = timeout in usec;
/* if dataDirection is O_RDONLY, and the length of the input data is
 * variable, the following parameter specifies the byte # (min == 0)
 * of the input data which will specify the additional number of
 * bytes available
 */
myScsiXaction.addLengthByte = X;
if (scsiIoctl (pScsiPhysDev, FIOSCSICOMMAND, &myScsiXaction) == OK)
    return (OK);
else
    /* optionally perform retry or other action based on value of
     *  myScsiXaction.statusByte
     */
    return (ERROR);
}
```

INCLUDE FILES **scsiLib.h**, **scsi1Lib.h**

SEE ALSO **dosFsLib**, **rt11FsLib**, *American National Standards for Information Systems - Small Computer System Interface (SCSI), ANSI X3.131-1986, VxWorks Programmer's Guide: I/O System, Local File Systems*

S

scsi2Lib

NAME **scsi2Lib** – Small Computer System Interface (SCSI) library (SCSI-2)

ROUTINES
scsi2IfInit( ) - initialize the SCSI-2 interface to **scsiLib**
scsiTargetOptionsSet( ) - set options for one or all SCSI targets
scsiTargetOptionsGet( ) - get options for one or all SCSI targets
scsiTargetOptionsShow( ) - display options for specified SCSI target
scsiPhysDevShow( ) - show status information for a physical device
scsiCacheSynchronize( ) - synchronize the caches for data coherency

scsiIdentMsgBuild( ) - build an identification message
scsiIdentMsgParse( ) - parse an identification message
scsiMsgOutComplete( ) - perform post-processing after a SCSI message is sent
scsiMsgOutReject( ) - perform post-processing when an outgoing message is rejected
scsiMsgInComplete( ) - handle a complete SCSI message received from the target
scsiSyncXferNegotiate( ) - initiate or continue negotiating transfer parameters
scsiWideXferNegotiate( ) - initiate or continue negotiating wide parameters
scsiThreadInit( ) - perform generic SCSI thread initialization
scsiCacheSnoopEnable( ) - inform SCSI that hardware snooping of caches is enabled
scsiCacheSnoopDisable( ) - inform SCSI that hardware snooping of caches is disabled

DESCRIPTION

This library implements the Small Computer System Interface (SCSI) protocol in a controller-independent manner. It implements only the SCSI initiator function as defined in the SCSI-2 ANSI specification. This library does not support a VxWorks target acting as a SCSI target.

The implementation is transaction based. A transaction is defined as the selection of a SCSI device by the initiator, the issuance of a SCSI command, and the sequence of data, status, and message phases necessary to perform the command. A transaction normally completes with a "Command Complete" message from the target, followed by disconnection from the SCSI bus. If the status from the target is "Check Condition," the transaction continues; the initiator issues a "Request Sense" command to gain more information on the exception condition reported.

Many of the subroutines in **scsi2Lib** facilitate the transaction of frequently used SCSI commands. Individual command fields are passed as arguments from which SCSI Command Descriptor Blocks are constructed, and fields of a **SCSI_TRANSACTION** structure are filled in appropriately. This structure, along with the **SCSI_PHYS_DEV** structure associated with the target SCSI device, is passed to the routine whose address is indicated by the **scsiTransact** field of the **SCSI_CTRL** structure associated with the relevant SCSI controller. The above mentioned structures are defined in **scsi2Lib.h**.

The function variable **scsiTransact** is set by the individual SCSI controller driver. For off-board SCSI controllers, this routine rearranges the fields of the **SCSI_TRANSACTION** structure into the appropriate structure for the specified hardware, which then carries out the transaction through firmware control. Drivers for an on-board SCSI-controller chip can use the **scsiTransact( )** routine in **scsiLib** (which invokes the **scsi2Transact( )** routine in **scsi2Lib**), as long as they provide the other functions specified in the **SCSI_CTRL** structure.

SCSI TRANSACTION TIMEOUT

Associated with each transaction is a time limit (specified in microseconds, but measured with the resolution of the system clock). If the transaction has not completed within this time limit, the SCSI library aborts it; the called routine fails with a corresponding error code. The timeout period includes time spent waiting for the target device to become free to accept the command.

The semantics of the timeout should guarantee that the caller waits no longer than the transaction timeout period, but in practice this may depend on the state of the SCSI bus and the connected target device when the timeout occurs. If the target behaves correctly according to the SCSI specification, proper timeout behavior results. However, in certain unusual cases--for example, when the target does not respond to an asserted ATN signal--the caller may remain blocked for longer than the timeout period.

If the transaction timeout causes problems in your system, you can set the value of either or both the global variables "scsi{Min,Max}Timeout". These specify (in microseconds) the global minimum and maximum timeout periods, which override (clip) the value specified for a transaction. They may be changed at any time and affect all transactions issued after the new values are set. The range of both these variable is 0 to 0xffffffff (zero to about 4295 seconds).

SCSI TRANSACTION PRIORITY

Each transaction also has an associated priority used by the SCSI library when selecting the next command to issue when the SCSI system is idle. It chooses the highest priority transaction that can be dispatched on an available physical device. If there are several equal-priority transactions available, the SCSI library uses a simple round-robin scheme to avoid favoring the same physical device.

Priorities range from 0 (highest) to 255 (lowest), which is the same as task priorities. The priority **SCSI_THREAD_TASK_PRIORITY** can be used to give the transaction the same priority as the calling task (this is the method used internally by this SCSI-2 library).

SUPPORTED SCSI DEVICES

This library requires peripherals that conform to the SCSI-2 ANSI standard; in particular, the INQUIRY, REQUEST SENSE, and TEST UNIT READY commands must be supported as specified by this standard. In general, the SCSI library is self-configuring to work with any device that meets these requirements.

Peripherals that support identification and the SCSI message protocol are strongly recommended as these provide maximum performance.

In theory, all classes of SCSI devices are supported. The **scsiLib** library provides the capability to transact any SCSI command on any SCSI device through the **FIOSCSICOMMAND** function of the **scsiIoctl( )** routine (which invokes the **scsi2Ioctl( )** routine in **scsi2Lib**).

Only direct-access devices (disks) are supported by file systems like dosFs, rt11Fs and rawFs. These file systems employ routines in **scsiDirectLib** (most of which are described in **scsiLib** but defined in **scsiDirectLib**). In the case of sequential-access devices (tapes), higher-level tape file systems, like tapeFs, make use of **scsiSeqLib**. For other types of devices, additional, higher-level software is necessary to map user-level commands to SCSI transactions.

DISCONNECT/RECONNECT SUPPORT

The target device can be disconnected from the SCSI bus while it carries out a SCSI command; in this way, commands to multiple SCSI devices can be overlapped to improve overall SCSI throughput. There are no restrictions on the number of pending, disconnected commands or the order in which they are resumed. The SCSI library serializes access to the device according to the capabilities and status of the device (see the following section).

Use of the disconnect/reconnect mechanism is invisible to users of the SCSI library. It can be enabled and disabled separately for each target device (see **scsiTargetOptionsSet( )**). Note that support for disconnect/reconnect depends on the capabilities of the controller and its driver (see below).

TAGGED COMMAND QUEUEING SUPPORT

If the target device conforms to the ANSI SCSI-2 standard and indicates (using the INQUIRY command) that it supports command queuing, the SCSI library allows new commands to be started on the device whenever the SCSI bus is idle. That is, it executes multiple commands concurrently on the target device. By default, commands are tagged with a SIMPLE QUEUE TAG message. Up to 256 commands can be executing concurrently.

The SCSI library correctly handles contingent allegiance conditions that arise while a device is executing tagged commands. (A contingent allegiance condition exists when a target device is maintaining sense data that the initiator should use to correctly recover from an error condition.) It issues an untagged REQUEST SENSE command, and stops issuing tagged commands until the sense recovery command has completed.

For devices that do not support command queuing, the SCSI library only issues a new command when the previous one has completed. These devices can only execute a single command at once.

Use of tagged command queuing is normally invisible to users of the SCSI library. If necessary, the default tag type and maximum number of tags may be changed on a per-target basis, using **scsiTargetOptionsSet( )**.

SYNCHRONOUS TRANSFER PROTOCOL SUPPORT

If the SCSI controller hardware supports the synchronous transfer protocol, **scsiLib** negotiates with the target device to determine whether to use synchronous or asynchronous transfers. Either VxWorks or the target device may start a round of negotiation. Depending on the controller hardware, synchronous transfer rates up to the maximum allowed by the SCSI-2 standard (10 Mtransfers/second) can be used.

Again, this is normally invisible to users of the SCSI library, but synchronous transfer parameters may be set or disabled on a per-target basis by using **scsiTargetOptionsSet( )**.

WIDE DATA TRANSFER SUPPORT

If the SCSI controller supports the wide data transfer protocol, **scsiLib** negotiates wide data transfer parameters with the target device, if that device also supports wide transfers. Either VxWorks or the target device may start a round of negotiation. Wide data transfer parameters are negotiated prior to the synchronous data transfer parameters, as specified by the SCSI-2 ANSI specification. In conjunction with synchronous transfer, up to a maximum of 20MB/sec. can be attained.

Wide data transfer negotiation is invisible to users of this library, but it is possible to enable or disable wide data transfers and the parameters on a per-target basis by using **scsiTargetOptionsSet( )**.

SCSI BUS RESET The SCSI library implements the ANSI "hard reset" option. Any transactions in progress when a SCSI bus reset is detected fail with an error code indicating termination due to bus reset. Any transactions waiting to start executing are then started normally.

CONFIGURING SCSI CONTROLLERS

The routines to create and initialize a specific SCSI controller are particular to the controller and normally are found in its library module. The normal calling sequence is:

```
xxCtrlCreate (...); /* parameters are controller specific */
xxCtrlInit (...);   /* parameters are controller specific */
```

The conceptual difference between the two routines is that **xxCtrlCreate( )** calloc's memory for the **xx_SCSI_CTRL** data structure and initializes information that is never expected to change (for example, clock rate). The remaining fields in the **xx_SCSI_CTRL** structure are initialized by **xxCtrlInit( )** and any necessary registers are written on the SCSI controller to effect the desired initialization. This routine can be called multiple times, although this is rarely required. For example, the bus ID of the SCSI controller can be changed without rebooting the VxWorks system.

CONFIGURING PHYSICAL SCSI DEVICES

Before a device can be used, it must be "created," that is, declared. This is done with **scsiPhysDevCreate( )** and can only be done after a **SCSI_CTRL** structure exists and has been properly initialized.

```
SCSI_PHYS_DEV *scsiPhysDevCreate
    (
    SCSI_CTRL * pScsiCtrl,/* ptr to SCSI controller info */
    int  devBusId,        /* device's SCSI bus ID */
    int  devLUN,          /* device's logical unit number */
    int  reqSenseLength,  /* length of REQUEST SENSE data dev returns */
    int  devType,         /* type of SCSI device */
    BOOL removable,       /* whether medium is removable */
    int  numBlocks,       /* number of blocks on device */
    int  blockSize        /* size of a block in bytes */
    )
```

Several of these parameters can be left unspecified, as follows:

reqSenseLength
If 0, issue a **REQUEST_SENSE** to determine a request sense length.

S

devType
: This parameter is ignored: an INQUIRY command is used to ascertain the device type. A value of NONE (-1) is the recommended placeholder.

numBlocks, *blockSize*
: If 0, issue a **READ_CAPACITY** to determine the number of blocks.

The above values are recommended, unless the device does not support the required commands, or other non-standard conditions prevail.

LOGICAL PARTITIONS ON DIRECT-ACCESS BLOCK DEVICES

It is possible to have more than one logical partition on a SCSI block device. This capability is currently not supported for removable media devices. A partition is an array of contiguously addressed blocks with a specified starting block address and specified number of blocks. The **scsiBlkDevCreate( )** routine is called once for each block device partition. Under normal usage, logical partitions should not overlap.

```
SCSI_BLK_DEV *scsiBlkDevCreate
    (
    SCSI_PHYS_DEV *  pScsiPhysDev,  /* ptr to SCSI physical device info */
    int              numBlocks,     /* number of blocks in block device */
    int              blockOffset    /* address of first block in volume */
    )
```

Note that if *numBlocks* is 0, the rest of the device is used.

ATTACHING DISK FILE SYSTEMS TO LOGICAL PARTITIONS

Files cannot be read or written to a disk partition until a file system (for example, dosFs, rt11Fs, or rawFs) has been initialized on the partition. For more information, see the relevant documentation in **dosFsLib**, **rt11FsLib**, or **rawFsLib**.

USING A SEQUENTIAL-ACCESS BLOCK DEVICE

The entire volume (tape) on a sequential-access block device is treated as a single raw file. This raw file is made available to higher-level layers like tapeFs by the **scsiSeqDevCreate( )** routine, described in **scsiSeqLib**. The **scsiSeqDevCreate( )** routine is called once for a given SCSI physical device.

```
SEQ_DEV *scsiSeqDevCreate
    (
    SCSI_PHYS_DEV *pScsiPhysDev * ptr to SCSI physical device info */
    )
```

TRANSMITTING ARBITRARY COMMANDS TO SCSI DEVICES

The **scsi2Lib**, **scsiCommonLib**, **scsiDirectLib**, and **scsiSeqLib** libraries collectively provide routines that implement all mandatory SCSI-2 direct-access and sequential-access commands. Still, there are situations that require commands not supported by these libraries (for example, writing software that needs to use an optional SCSI-2 command).

Arbitrary commands are handled with the **FIOSCSICOMMAND** option to **scsiIoctl( )**. The *arg* parameter for **FIOSCSICOMMAND** is a pointer to a valid **SCSI_TRANSACTION** structure. Typically, a call to **scsiIoctl( )** is written as a routine of the form:

```
STATUS myScsiCommand
    (
    SCSI_PHYS_DEV *  pScsiPhysDev,  /* ptr to SCSI physical device     */
    char *           buffer,        /* ptr to data buffer              */
    int              bufLength,     /* length of buffer in bytes       */
    int              someParam      /* param. specifiable in cmd block */
    )
    {
    SCSI_COMMAND myScsiCmdBlock;        /* SCSI command byte array */
    SCSI_TRANSACTION myScsiXaction;     /* info on a SCSI transaction */
    /* fill in fields of SCSI_COMMAND structure */
    myScsiCmdBlock [0] = MY_COMMAND_OPCODE;     /* the required opcode */
    .
    myScsiCmdBlock [X] = (UINT8) someParam;     /* for example */
    .
    myScsiCmdBlock [N-1] = MY_CONTROL_BYTE;     /* typically == 0 */
    /* fill in fields of SCSI_TRANSACTION structure */
    myScsiXaction.cmdAddress    = myScsiCmdBlock;
    myScsiXaction.cmdLength     = <# of valid bytes in myScsiCmdBlock>;
    myScsiXaction.dataAddress   = (UINT8 *) buffer;
    myScsiXaction.dataDirection = <O_RDONLY (0) or O_WRONLY (1)>;
    myScsiXaction.dataLength    = bufLength;
    myScsiXaction.addLengthByte = 0;            /* no longer used */
    myScsiXaction.cmdTimeout    = <timeout in usec>;
    myScsiXaction.tagType       = SCSI_TAG_{DEFAULT,UNTAGGED,
                                        SIMPLE,ORDERED,HEAD_OF_Q};
    myScsiXaction.priority      = [ 0 (highest) to 255 (lowest) ];
    if (scsiIoctl (pScsiPhysDev, FIOSCSICOMMAND, &myScsiXaction) == OK)
        return (OK);
    else
        /* optionally perform retry or other action based on value of
         *  myScsiXaction.statusByte
         */
        return (ERROR);
    }
```

INCLUDE FILES **scsiLib.h**, **scsi2Lib.h**

SEE ALSO **dosFsLib**, **rt11FsLib**, **rawFsLib**, **tapeFsLib**, **scsiLib**, **scsiCommonLib**, **scsiDirectLib**, **scsiSeqLib**, **scsiMgrLib**, **scsiCtrlLib**, *American National Standard for Information Systems - Small Computer System Interface (SCSI-2), ANSI X3T9, VxWorks Programmer's Guide: I/O System, Local File Systems*

S

scsiCommonLib

NAME

scsiCommonLib – SCSI library common commands for all devices (SCSI-2)

ROUTINES

No Callable Routines.

DESCRIPTION

This library contains commands common to all SCSI devices. The content of this library is separated from the other SCSI libraries in order to create an additional layer for better support of all SCSI devices.

Commands in this library include:

Command	Op Code
INQUIRY	(0x12)
REQUEST SENSE	(0x03)
TEST UNIT READY	(0x00)

INCLUDE FILES

scsiLib.h, **scsi2Lib.h**

SEE ALSO

dosFsLib, **rt11FsLib**, **rawFsLib**, **tapeFsLib**, **scsi2Lib**, *VxWorks Programmer's Guide: I/O System, Local File Systems*

scsiCtrlLib

NAME

scsiCtrlLib – SCSI thread-level controller library (SCSI-2)

ROUTINES

No Callable Routines.

DESCRIPTION

The purpose of the SCSI controller library is to support basic SCSI controller drivers that rely on a higher level of software in order to manage SCSI transactions. More advanced SCSI I/O processors do not require this protocol engine since software support for SCSI transactions is provided at the SCSI I/O processor level.

This library provides all the high-level routines that manage the state of the SCSI threads and guide the SCSI I/O transaction through its various stages:

– selecting a SCSI peripheral device;

– sending the identify message in order to establish the ITL nexus;

– cycling through information transfer, message and data, and status phases;

– handling bus-initiated reselects.

The various stages of the SCSI I/O transaction are reported to the SCSI manager as SCSI events. Event selection and management is handled by routines in this library.

INCLUDE FILES **scsiLib.h**, **scsi2Lib.h**

SEE ALSO **scsiLib**, **scsi2Lib**, **scsiCommonLib**, **scsiDirectLib**, **scsiSeqLib**, **scsiMgrLib**, *American National Standard for Information Systems - Small Computer System Interface (SCSI-2), ANSI X3T9, VxWorks Programmer's Guide: I/O System, Local File Systems*

---

scsiDirectLib

NAME **scsiDirectLib** – SCSI library for direct access devices (SCSI-2)

ROUTINES **scsiStartStopUnit( )** - issue a **START_STOP_UNIT** command to a SCSI device
scsiReserve( ) - issue a RESERVE command to a SCSI device
scsiRelease( ) - issue a RELEASE command to a SCSI device

DESCRIPTION This library contains commands common to all direct-access SCSI devices. These routines are separated from **scsi2Lib** in order to create an additional layer for better support of all SCSI direct-access devices.

Commands in this library include:

Command	Op Code
FORMAT UNIT	(0x04)
READ (6)	(0x08)
READ (10)	(0x28)
READ CAPACITY	(0x25)
RELEASE	(0x17)
RESERVE	(0x16)
MODE SELECT (6)	(0x15)
MODE SELECT (10)	(0x55)
MODE SENSE (6)	(0x1a)
MODE SENSE (10)	(0x5a)
START STOP UNIT	(0x1b)
WRITE (6)	(0x0a)
WRITE (10)	(0x2a)

INCLUDE FILES **scsiLib.h**, **scsi2Lib.h**

SEE ALSO **dosFsLib**, **rt11FsLib**, **rawFsLib**, **scsi2Lib**, *VxWorks Programmer's Guide: I/O System, Local File Systems*

S

scsiLib

NAME

scsiLib – Small Computer System Interface (SCSI) library

ROUTINES

scsiPhysDevDelete( ) - delete a SCSI physical-device structure
scsiPhysDevCreate( ) - create a SCSI physical device structure
scsiPhysDevIdGet( ) - return a pointer to a **SCSI_PHYS_DEV** structure
scsiAutoConfig( ) - configure all devices connected to a SCSI controller
scsiShow( ) - list the physical devices attached to a SCSI controller
scsiBlkDevCreate( ) - define a logical partition on a SCSI block device
scsiBlkDevInit( ) - initialize fields in a SCSI logical partition
scsiBlkDevShow( ) - show the **BLK_DEV** structures on a specified physical device
scsiBusReset( ) - pulse the reset signal on the SCSI bus
scsiIoctl( ) - perform a device-specific I/O control function
scsiFormatUnit( ) - issue a **FORMAT_UNIT** command to a SCSI device
scsiModeSelect( ) - issue a **MODE_SELECT** command to a SCSI device
scsiModeSense( ) - issue a **MODE_SENSE** command to a SCSI device
scsiReadCapacity( ) - issue a **READ_CAPACITY** command to a SCSI device
scsiRdSecs( ) - read sector(s) from a SCSI block device
scsiWrtSecs( ) - write sector(s) to a SCSI block device
scsiTestUnitRdy( ) - issue a **TEST_UNIT_READY** command to a SCSI device
scsiInquiry( ) - issue an **INQUIRY** command to a SCSI device
scsiReqSense( ) - issue a **REQUEST_SENSE** command to a SCSI device and read results

DESCRIPTION

The purpose of this library is to switch SCSI function calls (the common SCSI-1 and SCSI-2 calls listed above) to either **scsi1Lib** or **scsi2Lib**, depending upon the SCSI configuration in the Board Support Package (BSP). The normal usage is to configure SCSI-2. However, SCSI-1 is configured when device incompatibilities exist. VxWorks can be configured with either SCSI-1 or SCSI-2, but not both SCSI-1 and SCSI-2 simultaneously.

For more information about SCSI-1 functionality, refer to **scsi1Lib**. For more information about SCSI-2, refer to **scsi2Lib**.

INCLUDE FILES

scsiLib.h, **scsi1Lib.h**, **scsi2Lib.h**

SEE ALSO

dosFsLib, **rt11FsLib**, **rawFsLib**, **scsi1Lib**, **scsi2Lib**, *VxWorks Programmer's Guide: I/O System, Local File Systems*

scsiMgrLib

NAME

scsiMgrLib – SCSI manager library (SCSI-2)

ROUTINES

scsiMgrEventNotify( ) - notify the SCSI manager of a SCSI (controller) event
scsiMgrBusReset( ) - handle a controller-bus reset event
scsiMgrCtrlEvent( ) - send an event to the SCSI controller state machine
scsiMgrThreadEvent( ) - send an event to the thread state machine
scsiMgrShow( ) - show status information for the SCSI manager

DESCRIPTION

This SCSI-2 library implements the SCSI manager. The SCSI manager manages SCSI threads between requesting VxWorks tasks and the SCSI controller. The SCSI manager handles SCSI events and SCSI threads but allocation and de-allocation of SCSI threads is not the manager's responsibility. SCSI thread management includes despatching threads and scheduling multiple threads (which are performed by the SCSI manager, plus allocation and de-allocation of threads (which are performed by routines in **scsi2Lib**).

The SCSI manager is spawned as a task on initialization of the SCSI interface within VxWorks. The entry point of the SCSI manager task is **scsiMgr( )**. The task is usually spawned during initialization of the SCSI controller driver. The driver's **xxCtrlCreateScsi2( )** routine is typically responsible for such SCSI interface initializations.

Once the SCSI manager has been initialized, it is ready to handle SCSI requests from VxWorks tasks. The SCSI manager has the following responsibilities:

- It processes requests from client tasks.
- It activates a SCSI transaction thread by appending it to the target device's wait queue and allocating a specified time period to execute a transaction.
- It handles timeout events which cause threads to be aborted.
- It receives event notifications from the SCSI driver interrupt service routine (ISR) and processes the event.
- It responds to events generated by the controller hardware, such as disconnection and information transfer requests.
- It replies to clients when their requests have completed or aborted.

One SCSI manager task must be spawned per SCSI controller. Thus, if a particular hardware platform contains more than one SCSI controller then that number of SCSI manager tasks must be spawned by the controller-driver initialization routine.

INCLUDE FILES

scsiLib.h, **scsi2Lib.h**

SEE ALSO

scsiLib, **scsi2Lib**, **scsiCommonLib**, **scsiDirectLib**, **scsiSeqLib**, **scsiCtrlLib**, *American National Standard for Information Systems - Small Computer System Interface (SCSI-2), ANSI X3T9, VxWorks Programmer's Guide: I/O System, Local File Systems*

scsiSeqLib

NAME

scsiSeqLib – SCSI sequential access device library (SCSI-2)

ROUTINES

scsiSeqDevCreate( ) - create a SCSI sequential device
scsiErase( ) - issue an **ERASE** command to a SCSI device
scsiTapeModeSelect( ) - issue a **MODE_SELECT** command to a SCSI tape device
scsiTapeModeSense( ) - issue a **MODE_SENSE** command to a SCSI tape device
scsiSeqReadBlockLimits( ) - issue a **READ_BLOCK_LIMITS** command to a SCSI device
scsiRdTape( ) - read bytes or blocks from a SCSI tape device
scsiWrtTape( ) - write data to a SCSI tape device
scsiRewind( ) - issue a **REWIND** command to a SCSI device
scsiReserveUnit( ) - issue a **RESERVE UNIT** command to a SCSI device
scsiReleaseUnit( ) - issue a **RELEASE UNIT** command to a SCSI device
scsiLoadUnit( ) - issue a **LOAD/UNLOAD** command to a SCSI device
scsiWrtFileMarks( ) - write file marks to a SCSI sequential device
scsiSpace( ) - move the tape on a specified physical SCSI device
scsiSeqStatusCheck( ) - detect a change in media
scsiSeqIoctl( ) - perform an I/O control function for sequential access devices

DESCRIPTION

This library contains commands common to all sequential-access SCSI devices. Such devices are usually SCSI tape devices. These routines are separated from **scsi2Lib** in order to create an additional layer for better support of all SCSI sequential devices.

SCSI commands in this library include:

Command	Op Code
ERASE	(0x19)
MODE SELECT (6)	(0x15)
MODE_SENSE (6)	(0x1a)
READ (6)	(0x08)
READ BLOCK LIMITS	(0x05)
RELEASE UNIT	(0x17)
RESERVE UNIT	(0x16)
REWIND	(0x01)
SPACE	(0x11)
WRITE (6)	(0x0a)
WRITE FILEMARKS	(0x10)
LOAD/UNLOAD	(0x1b)

The SCSI routines implemented here operate mostly on a **SCSI_SEQ_DEV** structure. This structure acts as an interface between this library and a higher-level layer. The **SEQ_DEV** structure is analogous to the **BLK_DEV** structure for block devices.

The **scsiSeqDevCreate( )** routine creates a **SCSI_SEQ_DEV** structure whose first element is a **SEQ_DEV**, operated upon by higher layers. This routine publishes all functions to be invoked by higher layers and maintains some state information (for example, block size) for tracking SCSI-sequential-device information.

INCLUDE FILES **scsiLib.h**, **scsi2Lib.h**

SEE ALSO **tapeFsLib**, **scsi2Lib**, *VxWorks Programmer's Guide: I/O System, Local File Systems*

selectLib

NAME **selectLib** – UNIX BSD 4.3 select library

ROUTINES

selectInit( ) - initialize the select facility
select( ) - pend on a set of file descriptors
selWakeup( ) - wake up a task pended in **select( )**
selWakeupAll( ) - wake up all tasks in a **select( )** wake-up list
selNodeAdd( ) - add a wake-up node to a **select( )** wake-up list
selNodeDelete( ) - find and delete a node from a **select( )** wake-up list
selWakeupListInit( ) - initialize a **select( )** wake-up list
selWakeupListTerm( ) - terminate a **select( )** wake-up list
selWakeupListLen( ) - get the number of nodes in a **select( )** wake-up list
selWakeupType( ) - get the type of a **select( )** wake-up node

DESCRIPTION This library provides a BSD 4.3 compatible *select* facility to wait for activity on a set of file descriptors. **selectLib** provides a mechanism that gives a driver the ability to detect pended tasks that are awaiting activity on the driver's device. This allows a driver's interrupt service routine to wake up such tasks directly, eliminating the need for polling.

Applications can use **select( )** with pipes and serial devices, in addition to sockets. Also, **select( )** examines *write* file descriptors in addition to *read* file descriptors; however, exception file descriptors remain unsupported.

Typically, application developers need concern themselves only with the **select( )** call. However, driver developers should become familiar with the other routines that may be used with **select( )**, if they wish to support the **select( )** mechanism.

The select facility is included in a system when VxWorks is configured with the **INCLUDE_SELECT** component.

INCLUDE FILES **selectLib.h**

SEE ALSO *VxWorks Programmer's Guide: I/O System*

S

semBLib

NAME

semBLib – binary semaphore library

ROUTINES

semBCreate( ) - create and initialize a binary semaphore

DESCRIPTION

This library provides the interface to VxWorks binary semaphores. Binary semaphores are the most versatile, efficient, and conceptually simple type of semaphore. They can be used to: (1) control mutually exclusive access to shared devices or data structures, or (2) synchronize multiple tasks, or task-level and interrupt-level processes. Binary semaphores form the foundation of numerous VxWorks facilities.

A binary semaphore can be viewed as a cell in memory whose contents are in one of two states, full or empty. When a task takes a binary semaphore, using **semTake( )**, subsequent action depends on the state of the semaphore:

(1) If the semaphore is full, the semaphore is made empty, and the calling task continues executing.

(2) If the semaphore is empty, the task will be blocked, pending the availability of the semaphore. If a timeout is specified and the timeout expires, the pended task will be removed from the queue of pended tasks and enter the ready state with an **ERROR** status. A pended task is ineligible for CPU allocation. Any number of tasks may be pended simultaneously on the same binary semaphore.

When a task gives a binary semaphore, using **semGive( )**, the next available task in the pend queue is unblocked. If no task is pending on this semaphore, the semaphore becomes full. Note that if a semaphore is given, and a task is unblocked that is of higher priority than the task that called **semGive( )**, the unblocked task will preempt the calling task.

MUTUAL EXCLUSION

To use a binary semaphore as a means of mutual exclusion, first create it with an initial state of full. For example:

```
SEM_ID semMutex;
/* create a binary semaphore that is initially full */
semMutex = semBCreate (SEM_Q_PRIORITY, SEM_FULL);
```

Then guard a critical section or resource by taking the semaphore with **semTake( )**, and exit the section or release the resource by giving the semaphore with **semGive( )**. For example:

```
semTake (semMutex, WAIT_FOREVER);
    ...  /* critical region, accessible only by one task at a time */

semGive (semMutex);
```

While there is no restriction on the same semaphore being given, taken, or flushed by multiple tasks, it is important to ensure the proper functionality of the mutual-exclusion construct. While there is no danger in any number of processes taking a semaphore, the giving of a semaphore should be more carefully controlled. If a semaphore is given by a task that did not take it, mutual exclusion could be lost.

SYNCHRONIZATION

To use a binary semaphore as a means of synchronization, create it with an initial state of empty. A task blocks by taking a semaphore at a synchronization point, and it remains blocked until the semaphore is given by another task or interrupt service routine.

Synchronization with interrupt service routines is a particularly common need. Binary semaphores can be given, but not taken, from interrupt level. Thus, a task can block at a synchronization point with **semTake( )**, and an interrupt service routine can unblock that task with **semGive( )**.

In the following example, when **init( )** is called, the binary semaphore is created, an interrupt service routine is attached to an event, and a task is spawned to process the event. Task 1 will run until it calls **semTake( )**, at which point it will block until an event causes the interrupt service routine to call **semGive( )**. When the interrupt service routine completes, task 1 can execute to process the event.

```
SEM_ID semSync;    /* ID of sync semaphore */
init ()
    {
    intConnect (..., eventInterruptSvcRout, ...);
    semSync = semBCreate (SEM_Q_FIFO, SEM_EMPTY);
    taskSpawn (..., task1);
    }
task1 ()
    {
    ...
    semTake (semSync, WAIT_FOREVER);    /* wait for event */
    ...    /* process event */
    }
eventInterruptSvcRout ()
    {
    ...
    semGive (semSync);    /* let task 1 process event */
    ...
    }
```

A **semFlush( )** on a binary semaphore will atomically unblock all pended tasks in the semaphore queue, *i.e.*, all tasks will be unblocked at once, before any actually execute.

CAVEATS

There is no mechanism to give back or reclaim semaphores automatically when tasks are suspended or deleted. Such a mechanism, though desirable, is not currently feasible.

Without explicit knowledge of the state of the guarded resource or region, reckless automatic reclamation of a semaphore could leave the resource in a partial state. Thus, if a task ceases execution unexpectedly, as with a bus error, currently owned semaphores will not be given back, effectively leaving a resource permanently unavailable. The mutual-exclusion semaphores provided by **semMLib** offer protection from unexpected task deletion.

INCLUDE FILES **semLib.h**

SEE ALSO **semLib**, **semCLib**, **semMLib**, *VxWorks Programmer's Guide: Basic OS*

semCLib

NAME **semCLib** – counting semaphore library

ROUTINES **semCCreate( )** - create and initialize a counting semaphore

DESCRIPTION

This library provides the interface to VxWorks counting semaphores. Counting semaphores are useful for guarding multiple instances of a resource.

A counting semaphore may be viewed as a cell in memory whose contents keep track of a count. When a task takes a counting semaphore, using **semTake( )**, subsequent action depends on the state of the count:

(1) If the count is non-zero, it is decremented and the calling task continues executing.

(2) If the count is zero, the task will be blocked, pending the availability of the semaphore. If a timeout is specified and the timeout expires, the pended task will be removed from the queue of pended tasks and enter the ready state with an **ERROR** status. A pended task is ineligible for CPU allocation. Any number of tasks may be pended simultaneously on the same counting semaphore.

When a task gives a semaphore, using **semGive( )**, the next available task in the pend queue is unblocked. If no task is pending on this semaphore, the semaphore count is incremented. Note that if a semaphore is given, and a task is unblocked that is of higher priority than the task that called **semGive( )**, the unblocked task will preempt the calling task.

A **semFlush( )** on a counting semaphore will atomically unblock all pended tasks in the semaphore queue. So all tasks will be made ready before any task actually executes. The count of the semaphore will remain unchanged.

INTERRUPT USAGE

Counting semaphores may be given but not taken from interrupt level.

CAVEATS There is no mechanism to give back or reclaim semaphores automatically when tasks are suspended or deleted. Such a mechanism, though desirable, is not currently feasible. Without explicit knowledge of the state of the guarded resource or region, reckless automatic reclamation of a semaphore could leave the resource in a partial state. Thus, if a task ceases execution unexpectedly, as with a bus error, currently owned semaphores will not be given back, effectively leaving a resource permanently unavailable. The mutual-exclusion semaphores provided by **semMLib** offer protection from unexpected task deletion.

INCLUDE FILES **semLib.h**

SEE ALSO **semLib**, **semBLib**, **semMLib**, *VxWorks Programmer's Guide: Basic OS*

semEvLib

NAME **semEvLib** – VxWorks 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, the events specified are sent to that task. However, if a **semTake( )** is to be done afterwards, there is no guarantee that the semaphore will still be available.

INCLUDE FILES **semEvLib.h**

SEE ALSO **eventLib**, **semLib**, *VxWorks Programmer's Guide: Basic OS*

semLib

NAME **semLib** – general semaphore library

ROUTINES

semGive( ) - give a semaphore
semTake( ) - take a semaphore
semFlush( ) - unblock every task pended on a semaphore
semDelete( ) - delete a semaphore

DESCRIPTION

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.

Semaphore creation and initialization is handled by other libraries, depending on the type of semaphore used. These libraries contain full functional descriptions of the semaphore types:

semBLib - binary semaphores
semCLib - counting semaphores
semMLib - mutual exclusion semaphores
semSmLib - shared memory semaphores

Binary semaphores offer the greatest speed and the broadest applicability.

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

SEMAPHORE CONTROL

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

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

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

SEMAPHORE DELETION

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

SEMAPHORE INFORMATION

The **semInfo( )** call is a useful debugging aid, reporting all tasks blocked on a specified semaphore. It provides a snapshot of the queue at the time of the call, but because semaphores are dynamic, the information may be out of date by the time it is available. As with the current state of the semaphore, use of the queue of pended tasks should be restricted to debugging uses only.

VXWORKS EVENTS

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

INCLUDE FILES

semLib.h

SEE ALSO

taskLib, **semBLib**, **semCLib**, **semMLib**, **semSmLib**, **semEvLib**, **eventLib**, *VxWorks Programmer's Guide: Basic OS*

semMLib

NAME

semMLib – mutual-exclusion semaphore library

ROUTINES

semMCreate( ) - create and initialize a mutual-exclusion semaphore
semMGiveForce( ) - give a mutual-exclusion semaphore without restrictions

DESCRIPTION

This library provides the interface to VxWorks mutual-exclusion semaphores. Mutual-exclusion semaphores offer convenient options suited for situations requiring mutually exclusive access to resources. Typical applications include sharing devices and protecting data structures. Mutual-exclusion semaphores are used by many higher-level VxWorks facilities.

The mutual-exclusion semaphore is a specialized version of the binary semaphore, designed to address issues inherent in mutual exclusion, such as recursive access to resources, priority inversion, and deletion safety. The fundamental behavior of the mutual-exclusion semaphore is identical to the binary semaphore (see the manual entry for **semBLib**), except for the following restrictions:

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

These last two operations have no meaning in mutual-exclusion situations.

RECURSIVE RESOURCE ACCESS

A special feature of the mutual-exclusion semaphore is that it may be taken "recursively," *i.e.*, it can be taken more than once by the task that owns it before finally being released. Recursion is useful for a set of routines that need mutually exclusive access to a resource, but may need to call each other.

Recursion is possible because the system keeps track of which task currently owns a mutual-exclusion semaphore. Before being released, a mutual-exclusion semaphore taken recursively must be given the same number of times it has been taken; this is tracked by means of a count which is incremented with each **semTake( )** and decremented with each **semGive( )**.

The example below illustrates recursive use of a mutual-exclusion semaphore. Function A requires access to a resource which it acquires by taking **semM**; function A may also need to call function B, which also requires **semM**:

```
SEM_ID semM;
semM = semMCreate (...);
funcA ()
    {
    semTake (semM, WAIT_FOREVER);
```

```
    ...
    funcB ();
    ...
    semGive (semM);
    }
funcB ()
    {
    semTake (semM, WAIT_FOREVER);
    ...
    semGive (semM);
    }
```

PRIORITY-INVERSION SAFETY

If the option **SEM_INVERSION_SAFE** is selected, the library adopts a priority-inheritance protocol to resolve potential occurrences of "priority inversion," a problem stemming from the use semaphores for mutual exclusion. Priority inversion arises when a higher-priority task is forced to wait an indefinite period of time for the completion of a lower-priority task.

Consider the following scenario: T1, T2, and T3 are tasks of high, medium, and low priority, respectively. T3 has acquired some resource by taking its associated semaphore. When T1 preempts T3 and contends for the resource by taking the same semaphore, it becomes blocked. If we could be assured that T1 would be blocked no longer than the time it normally takes T3 to finish with the resource, the situation would not be problematic. However, the low-priority task is vulnerable to preemption by medium-priority tasks; a preempting task, T2, could inhibit T3 from relinquishing the resource. This condition could persist, blocking T1 for an indefinite period of time.

The priority-inheritance protocol solves the problem of priority inversion by elevating the priority of T3 to the priority of T1 during the time T1 is blocked on T3. This protects T3, and indirectly T1, from preemption by T2. Stated more generally, the priority-inheritance protocol assures that a task which owns a resource will execute at the priority of the highest priority task blocked on that resource. Once the task priority has been elevated, it remains at the higher level until all mutual-exclusion semaphores that the task owns are released; then the task returns to its normal, or standard, priority. Hence, the "inheriting" task is protected from preemption by any intermediate-priority tasks.

The priority-inheritance protocol also takes into consideration a task's ownership of more than one mutual-exclusion semaphore at a time. Such a task will execute at the priority of the highest priority task blocked on any of its owned resources. The task will return to its normal priority only after relinquishing all of its mutual-exclusion semaphores that have the inversion-safety option enabled.

SEMAPHORE DELETION

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

TASK-DELETION SAFETY

If the option **SEM_DELETE_SAFE** is selected, the task owning the semaphore will be protected from deletion as long as it owns the semaphore. This solves another problem endemic to mutual exclusion. Deleting a task executing in a critical region can be catastrophic. The resource could be left in a corrupted state and the semaphore guarding the resource would be unavailable, effectively shutting off all access to the resource.

As discussed in **taskLib**, the primitives **taskSafe( )** and **taskUnsafe( )** offer one solution, but as this type of protection goes hand in hand with mutual exclusion, the mutual-exclusion semaphore provides the option **SEM_DELETE_SAFE**, which enables an implicit **taskSafe( )** with each **semTake( )**, and a **taskUnsafe( )** with each **semGive( )**. This convenience is also more efficient, as the resulting code requires fewer entrances to the kernel.

CAVEATS

There is no mechanism to give back or reclaim semaphores automatically when tasks are suspended or deleted. Such a mechanism, though desirable, is not currently feasible. Without explicit knowledge of the state of the guarded resource or region, reckless automatic reclamation of a semaphore could leave the resource in a partial state. Thus if a task ceases execution unexpectedly, as with a bus error, currently owned semaphores will not be given back, effectively leaving a resource permanently unavailable. The **SEM_DELETE_SAFE** option partially protects an application, to the extent that unexpected deletions will be deferred until the resource is released.

Because the priority of a task which has been elevated by the taking of a mutual-exclusion semaphore remains at the higher priority until all mutexes held by that task are released, unbounded priority inversion situations can result when nested mutexes are involved. If nested mutexes are required, consider the following alternatives:

1. Avoid overlapping critical regions.
2. Adjust priorities of tasks so that there are no tasks at intermediate priority levels.
3. Adjust priorities of tasks so that priority inheritance protocol is not needed.
4. Manually implement a static priority ceiling protocol using a non-inversion-save mutex. This involves setting all blockers on a mutex to the ceiling priority, then taking the mutex. After **semGive( )**, set the priorities back to the base priority. Note that this implementation reduces the queue to a fifo queue.

INCLUDE FILES

semLib.h

SEE ALSO

semLib, **semBLib**, **semCLib**, *VxWorks Programmer's Guide: Basic OS*

semOLib

NAME

semOLib – release 4.x binary semaphore library

ROUTINES

semCreate( ) - create and initialize a release 4.x binary semaphore
semInit( ) - initialize a static binary semaphore
semClear( ) - take a release 4.x semaphore, if the semaphore is available

DESCRIPTION

This library is provided for backward compatibility with VxWorks 4.x semaphores. The semaphores are identical to 5.0 binary semaphores, except that timeouts -- missing or specified -- are ignored.

For backward compatibility, **semCreate( )** operates as before, allocating and initializing a 4.x-style semaphore. Likewise, **semClear( )** has been implemented as a **semTake( )**, with a timeout of **NO_WAIT**.

For more information on of the behavior of binary semaphores, see the manual entry for **semBLib**.

INCLUDE FILES

semLib.h

SEE ALSO

semLib, **semBLib**, *VxWorks Programmer's Guide: Basic OS*

semPxLib

NAME

semPxLib – semaphore synchronization library (POSIX)

ROUTINES

semPxLibInit( ) - initialize POSIX semaphore support
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)

DESCRIPTION

This library implements the POSIX 1003.1b semaphore interface. For alternative semaphore routines designed expressly for VxWorks, see the manual page for **semLib**

S

and other semaphore libraries mentioned there. POSIX semaphores are counting semaphores; as such they are most similar to the **semCLib** VxWorks-specific 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 manual entry, and the manual entries for subroutines in this library) uses the POSIX terminology, in order to make it easier to read in conjunction with other references on POSIX.

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 only close semaphores that the closing task has opened.)

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

INCLUDE FILES

semaphore.h

SEE ALSO

POSIX 1003.1b document, **semLib**, *VxWorks Programmer's Guide: Basic OS*

semPxShow

NAME **semPxShow** – POSIX semaphore show library

ROUTINES **semPxShowInit( )** - initialize the POSIX semaphore show facility

DESCRIPTION This library provides a show routine for POSIX semaphore objects.

semShow

NAME **semShow** – semaphore show routines

ROUTINES
semShowInit( ) - initialize the semaphore show facility
semInfo( ) - get a list of task IDs that are blocked on a semaphore
semShow( ) - show information about a semaphore

DESCRIPTION This library provides routines to show semaphore statistics, such as semaphore type, semaphore queuing method, tasks pended, *etc*.

The routine **semShowInit( )** links the semaphore show facility into the VxWorks system. It is called automatically when the semaphore show facility is configured into VxWorks using either of the following methods:

If you use the configuration header files, define **INCLUDE_SHOW_ROUTINES** in **config.h**.

If you use the Tornado project facility, select **INCLUDE_SEM_SHOW**.

INCLUDE FILES **semLib.h**

SEE ALSO **semLib**, *VxWorks Programmer's Guide: Basic OS*

S

semSmLib

NAME

semSmLib – shared memory semaphore library (VxMP Opt.)

ROUTINES

semBSmCreate( ) - create and initialize shared memory binary semaphore (VxMP Opt.)
semCSmCreate( ) - create and initialize a shared memory counting semaphore (VxMP Opt.)

DESCRIPTION

This library provides the interface to VxWorks shared memory binary and counting semaphores. Once a shared memory semaphore is created, the generic semaphore-handling routines provided in **semLib** are used to manipulate it. Shared memory binary semaphores are created using **semBSmCreate( )**. Shared memory counting semaphores are created using **semCSmCreate( )**.

Shared memory binary semaphores are used to: (1) control mutually exclusive access to multiprocessor-shared data structures, or (2) synchronize multiple tasks running in a multiprocessor system. For general information about binary semaphores, see the manual entry **semBLib**.

Shared memory counting semaphores are used for guarding multiple instances of a resource used by multiple CPUs. For general information about shared counting semaphores, see the manual entry for **semCLib**.

For information about the generic semaphore-handling routines, see the manual entry for **semLib**.

MEMORY REQUIREMENTS

The semaphore structure is allocated from a dedicated shared memory partition.

The shared semaphore dedicated shared memory partition is initialized by the shared memory objects master CPU. The size of this partition is defined by the maximum number of shared semaphores, set in the configuration parameter **SM_OBJ_MAX_SEM**.

This memory partition is common to shared binary and counting semaphores, thus **SM_OBJ_MAX_SEM** must be set to the sum total of binary and counting semaphores to be used in the system.

RESTRICTIONS

Shared memory semaphores differ from local semaphores in the following ways:

Interrupt Use:
Shared semaphores may not be given, taken, or flushed at interrupt level.

Deletion:
There is no way to delete a shared semaphore and free its associated shared memory. Attempts to delete a shared semaphore return **ERROR** and set **errno** to **S_smObjLib_NO_OBJECT_DESTROY**.

Queuing Style:
The shared semaphore queuing style specified when the semaphore is created must be FIFO.

INTERRUPT LATENCY
Internally, interrupts are locked while manipulating shared semaphore data structures, thus increasing local CPU interrupt latency.

CONFIGURATION Before routines in this library can be called, the shared memory object facility must be initialized by calling **usrSmObjInit( )**. This is done automatically during VxWorks initialization when the component **INCLUDE_SM_OBJ** is included.

AVAILABILITY This module is distributed as a component of the unbundled shared memory support option, VxMP.

INCLUDE FILES **semSmLib.h**

SEE ALSO **semLib**, **semBLib**, **semCLib**, **smObjLib**, **semShow**, **usrSmObjInit( )**, *VxWorks Programmer's Guide: Shared Memory Objects, VxWorks Programmer's Guide: Basic OS*

---

shellLib

NAME **shellLib** – shell execution routines

ROUTINES
shellInit( ) - start the shell
shell( ) - the shell entry point
shellScriptAbort( ) - signal the shell to stop processing a script
shellHistory( ) - display or set the size of shell history
shellPromptSet( ) - change the shell prompt
shellOrigStdSet( ) - set the shell's default input/output/error file descriptors
shellLock( ) - lock access to the shell

DESCRIPTION This library contains the execution support routines for the VxWorks shell. It provides the basic programmer's interface to VxWorks. It is a C-expression interpreter, containing no built-in commands.

The nature, use, and syntax of the shell are more fully described in the "Target Shell" chapter of the *VxWorks Programmer's Guide.*

INCLUDE FILES **shellLib.h**

SEE ALSO **ledLib**, *VxWorks Programmer's Guide: Target Shell*

sigLib

NAME

sigLib – software signal facility library

ROUTINES

sigInit( ) - initialize the signal facilities
sigqueueInit( ) - initialize the queued signal facilities
sigemptyset( ) - initialize a signal set with no signals included (POSIX)
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 (POSIX)
sigismember( ) - test to see if a signal is in a signal set (POSIX)
signal( ) - specify the handler associated with a signal
sigaction( ) - examine and/or specify the action associated with a signal (POSIX)
sigprocmask( ) - examine and/or change the signal mask (POSIX)
sigpending( ) - retrieve the set of pending signals blocked from delivery (POSIX)
sigsuspend( ) - suspend the task until delivery of a signal (POSIX)
pause( ) - suspend the task until delivery of a signal (POSIX)
sigtimedwait( ) - wait for a signal
sigwaitinfo( ) - wait for real-time signals
sigwait( ) - wait for a signal to be delivered (POSIX)
sigvec( ) - install a signal handler
sigsetmask( ) - set the signal mask
sigblock( ) - add to a set of blocked signals
raise( ) - send a signal to the caller's task
kill( ) - send a signal to a task (POSIX)
sigqueue( ) - send a queued signal to a task

DESCRIPTION

This library provides a signal interface for tasks. Signals are used to alter the flow control of tasks by communicating asynchronous events within or between task contexts. Any task or interrupt service can "raise" (or send) a signal to a particular task. The task being signaled will immediately suspend its current thread of execution and invoke a task-specified "signal handler" routine. The signal handler is a user-supplied routine that is bound to a specific signal and performs whatever actions are necessary whenever the signal is received. Signals are most appropriate for error and exception handling, rather than as a general purpose intertask communication mechanism.

This library has both a BSD 4.3 and POSIX signal interface. The POSIX interface provides a standardized interface which is more functional than the traditional BSD 4.3 interface. The chart below shows the correlation between BSD 4.3 and POSIX 1003.1 functions. An application should use only one form of interface and not intermix them.

BSD 4.3	POSIX 1003.1
sigmask( )	**sigemptyset( )**, **sigfillset( )**, **sigaddset( )**, **sigdelset( )**, **sigismember( )**

BSD 4.3	POSIX 1003.1
sigblock( )	**sigprocmask( )**
sigsetmask( )	**sigprocmask( )**
pause( )	**sigsuspend( )**
sigvec( )	**sigaction( )**
(none)	**sigpending( )**
signal( )	**signal( )**
kill( )	**kill( )**

POSIX 1003.1b (Real-Time Extensions) also specifies a queued-signal facility that involves four additional routines: **sigqueue( )**, **sigwaitinfo( )**, and **sigtimedwait( )**.

In many ways, signals are analogous to hardware interrupts. The signal facility provides a set of 31 distinct signals. A signal can be raised by calling **kill( )**, which is analogous to an interrupt or hardware exception. A signal handler is bound to a particular signal with **sigaction( )** in much the same way that an interrupt service routine is connected to an interrupt vector with **intConnect( )**. Signals are blocked for the duration of the signal handler, just as interrupts are locked out for the duration of the interrupt service routine. Tasks can block the occurrence of certain signals with **sigprocmask( )**, just as the interrupt level can be raised or lowered to block out levels of interrupts. If a signal is blocked when it is raised, its handler routine will be called when the signal becomes unblocked.

Several routines (**sigprocmask( )**, **sigpending( )**, and **sigsuspend( )**) take **sigset_t** data structures as parameters. These data structures are used to specify signal set masks. Several routines are provided for manipulating these data structures: **sigemptyset( )** clears all the bits in a **segset_t**, **sigfillset( )** sets all the bits in a **sigset_t**, **sigaddset( )** sets the bit in a **sigset_t** corresponding to a particular signal number, **sigdelset( )** resets the bit in a **sigset_t** corresponding to a particular signal number, and **sigismember( )** tests to see if the bit corresponding to a particular signal number is set.

FUNCTION RESTARTING

If a task is pended (for instance, by waiting for a semaphore to become available) and a signal is sent to the task for which the task has a handler installed, then the handler will run before the semaphore is taken. When the handler is done, the task will go back to being pended (waiting for the semaphore). If there was a timeout used for the pend, then the original value will be used again when the task returns from the signal handler and goes back to being pended.

Signal handlers are typically defined as:

```
void sigHandler
    (
    int sig,                        /* signal number                 */
    )
    {
        ...
    }
```

In VxWorks, the signal handler is passed additional arguments and can be defined as:

```
void sigHandler
    (
    int sig,                        /* signal number                 */
    int code,                       /* additional code               */
    struct sigcontext *pSigContext  /* context of task before signal */
    )
    {
        ...
    }
```

The parameter *code* is valid only for signals caused by hardware exceptions. In this case, it is used to distinguish signal variants. For example, both numeric overflow and zero divide raise **SIGFPE** (floating-point exception) but have different values for *code*. (Note that when the above VxWorks extensions are used, the compiler may issue warnings.)

SIGNAL HANDLER DEFINITION

Signal handling routines must follow one of two specific formats, so that they may be correctly called by the operating system when a signal occurs.

Traditional signal handlers receive the signal number as the sole input parameter. However, certain signals generated by routines which make up the POSIX Real-Time Extensions (P1003.1b) support the passing of an additional application-specific value to the handler routine. These include signals generated by the **sigqueue( )** call, by asynchronous I/O, by POSIX real-time timers, and by POSIX message queues.

If a signal handler routine is to receive these additional parameters, **SA_SIGINFO** must be set in the **sa_flags** field of the **sigaction** structure which is a parameter to the **sigaction( )** routine. Such routines must take the following form:

```
void sigHandler (int sigNum, siginfo_t * pInfo, void * pContext);
```

Traditional signal handling routines must not set **SA_SIGINFO** in the **sa_flags** field, and must take the form of:

```
void sigHandler (int sigNum);
```

EXCEPTION PROCESSING

Certain signals, defined below, are raised automatically when hardware exceptions are encountered. This mechanism allows user-defined exception handlers to be installed. This is useful for recovering from catastrophic events such as bus or arithmetic errors. Typically, **setjmp( )** is called to define the point in the program where control will be restored, and **longjmp( )** is called in the signal handler to restore that context. Note that **longjmp( )** restores the state of the task's signal mask. If a user-defined handler is not installed or the installed handler returns for a signal raised by a hardware exception, then the task is suspended and a message is logged to the console.

The following is a list of hardware exceptions caught by VxWorks and delivered to the offending task. The user may include the higher-level header file **sigCodes.h** in order to access the appropriate architecture-specific header file containing the code value.

Motorola 68K

Signal	Code	Exception
SIGSEGV	**NULL**	bus error
SIGBUS	**BUS_ADDERR**	address error
SIGILL	**ILL_ILLINSTR_FAULT**	illegal instruction
SIGFPE	**FPE_INTDIV_TRAP**	zero divide
SIGFPE	**FPE_CHKINST_TRAP**	chk trap
SIGFPE	**FPE_TRAPV_TRAP**	trapv trap
SIGILL	**ILL_PRIVVIO_FAULT**	privilege violation
SIGTRAP	**NULL**	trace exception
SIGEMT	**EMT_EMU1010**	line 1010 emulator
SIGEMT	**EMT_EMU1111**	line 1111 emulator
SIGILL	**ILL_ILLINSTR_FAULT**	coprocessor protocol violation
SIGFMT	**NULL**	format error
SIGFPE	**FPE_FLTBSUN_TRAP**	compare unordered
SIGFPE	**FPE_FLTINEX_TRAP**	inexact result
SIGFPE	**FPE_FLTDIV_TRAP**	divide by zero
SIGFPE	**FPE_FLTUND_TRAP**	underflow
SIGFPE	**FPE_FLTOPERR_TRAP**	operand error
SIGFPE	**FPE_FLTOVF_TRAP**	overflow
SIGFPE	**FPE_FLTNAN_TRAP**	signaling "Not A Number"

MIPS R3000/R4000

Signal	Code	Exception
SIGBUS	**BUS_TLBMOD**	TLB modified
SIGBUS	**BUS_TLBL**	TLB miss on a load instruction
SIGBUS	**BUS_TLBS**	TLB miss on a store instruction
SIGBUS	**BUS_ADEL**	address error (bad alignment) on load instr
SIGBUS	**BUS_ADES**	address error (bad alignment) on store instr
SIGSEGV	**SEGV_IBUS**	bus error (instruction)
SIGSEGV	**SEGV_DBUS**	bus error (data)
SIGTRAP	**TRAP_SYSCALL**	syscall instruction executed
SIGTRAP	**TRAP_BP**	break instruction executed
SIGILL	**ILL_ILLINSTR_FAULT**	reserved instruction
SIGILL	**ILL_COPROC_UNUSABLE**	coprocessor unusable
SIGFPE	**FPE_FPA_UIO, SIGFPE**	unimplemented FPA operation

Signal	Code	Exception
SIGFPE	**FPE_FLTNAN_TRAP**	invalid FPA operation
SIGFPE	**FPE_FLTDIV_TRAP**	FPA divide by zero
SIGFPE	**FPE_FLTOVF_TRAP**	FPA overflow exception
SIGFPE	**FPE_FLTUND_TRAP**	FPA underflow exception
SIGFPE	**FPE_FLTINEX_TRAP**	FPA inexact operation

Intel i386/i486

Signal	Code	Exception
SIGILL	**ILL_DIVIDE_ERROR**	divide error
SIGEMT	**EMT_DEBUG**	debugger call
SIGILL	**ILL_NON_MASKABLE**	NMI interrupt
SIGEMT	**EMT_BREAKPOINT**	breakpoint
SIGILL	**ILL_OVERFLOW**	INTO-detected overflow
SIGILL	**ILL_BOUND**	bound range exceeded
SIGILL	**ILL_INVALID_OPCODE**	invalid opcode
SIGFPE	**FPE_NO_DEVICE**	device not available
SIGILL	**ILL_DOUBLE_FAULT**	double fault
SIGFPE	**FPE_CP_OVERRUN**	coprocessor segment overrun
SIGILL	**ILL_INVALID_TSS**	invalid task state segment
SIGBUS	**BUS_NO_SEGMENT**	segment not present
SIGBUS	**BUS_STACK_FAULT**	stack exception
SIGILL	**ILL_PROTECTION_FAULT**	general protection
SIGBUS	**BUS_PAGE_FAULT**	page fault
SIGILL	**ILL_RESERVED**	(intel reserved)
SIGFPE	**FPE_CP_ERROR**	coprocessor error
SIGBUS	**BUS_ALIGNMENT**	alignment check

PowerPC

Signal	Code	Exception
SIGBUS	**_EXC_OFF_MACH**	machine check
SIGBUS	**_EXC_OFF_INST**	instruction access
SIGBUS	**_EXC_OFF_ALIGN**	alignment
SIGILL	**_EXC_OFF_PROG**	program
SIGBUS	**_EXC_OFF_DATA**	data access
SIGFPE	**_EXC_OFF_FPU**	floating point unavailable
SIGTRAP	**_EXC_OFF_DBG**	debug exception (PPC403)
SIGTRAP	**_EXC_OFF_INST_BRK**	inst. breakpoint (PPC603, PPCEC603, PPC604)
SIGTRAP	**_EXC_OFF_TRACE**	trace (PPC603, PPCEC603, PPC604, PPC860)
SIGBUS	**_EXC_OFF_CRTL**	critical interrupt (PPC403)

Signal	Code	Exception
SIGILL	**_EXC_OFF_SYSCALL**	system call

Hitachi SH770x

Signal	Code	Exception
SIGSEGV	**TLB_LOAD_MISS**	TLB miss/invalid (load)
SIGSEGV	**TLB_STORE_MISS**	TLB miss/invalid (store)
SIGSEGV	**TLB_INITITIAL_PAGE_WRITE**	Initial page write
SIGSEGV	**TLB_LOAD_PROTEC_VIOLATION**	TLB protection violation (load)
SIGSEGV	**TLB_STORE_PROTEC_VIOLATION**	TLB protection violation (store)
SIGBUS	**BUS_LOAD_ADDRESS_ERROR**	Address error (load)
SIGBUS	**BUS_STORE_ADDRESS_ERROR**	Address error (store)
SIGILL	**ILLEGAL_INSTR_GENERAL**	general illegal instruction
SIGILL	**ILLEGAL_SLOT_INSTR**	slot illegal instruction
SIGFPE	**FPE_INTDIV_TRAP**	integer zero divide

Hitachi SH7604/SH704x/SH703x/SH702x

Signal	Code	Exception
SIGILL	**ILL_ILLINSTR_GENERAL**	general illegal instruction
SIGILL	**ILL_ILLINSTR_SLOT**	slot illegal instruction
SIGBUS	**BUS_ADDERR_CPU**	CPU address error
SIGBUS	**BUS_ADDERR_DMA**	DMA address error
SIGFPE	**FPE_INTDIV_TRAP**	integer zero divide

Two signals are provided for application use: **SIGUSR1** and **SIGUSR2**. VxWorks will never use these signals; however, other signals may be used by VxWorks in the future.

INCLUDE FILES **signal.h**

SEE ALSO **intLib**, IEEE *POSIX 1003.1b, VxWorks Programmer's Guide: Basic OS*

smMemLib

NAME **smMemLib** – shared memory management library (VxMP Opt.)

ROUTINES
memPartSmCreate( ) - create a shared memory partition
smMemAddToPool( ) - add memory to shared memory system partition
smMemOptionsSet( ) - set debug options for shared memory system partition

smMemMalloc( ) - allocate block of memory from shared memory system partition
smMemCalloc( ) - allocate memory for array from shared memory system partition
smMemRealloc( ) - reallocate block of memory from shared memory system partition
smMemFree( ) - free a shared memory system partition block of memory
smMemFindMax( ) - find largest free block in shared memory system partition

DESCRIPTION

This library provides facilities for managing the allocation of blocks of shared memory from ranges of memory called shared memory partitions. The routine **memPartSmCreate( )** is used to create shared memory partitions in the shared memory pool. The created partition can be manipulated using the generic memory partition calls, **memPartAlloc( )**, **memPartFree( )**, *etc*. (for a complete list of these routines, see the manual entry for **memPartLib**). The maximum number of partitions that can be created is determined by the configuration parameter **SM_OBJ_MAX_MEM_PART**.

The **smMem...( )** routines provide an easy-to-use interface to the shared memory system partition. The shared memory system partition is created when the shared memory object facility is initialized.

Shared memory management information and statistics display routines are provided by **smMemShow**.

The allocation of memory, using **memPartAlloc( )** in the general case and **smMemMalloc( )** for the shared memory system partition, is done with a first-fit algorithm. Adjacent blocks of memory are coalesced when freed using **memPartFree( )** and **smMemFree( )**.

There is a 28-byte overhead per allocated block (architecture dependent), and allocated blocks are aligned on a 16-byte boundary.

All memory used by the shared memory facility must be in the same address space, that is, it must be reachable from all the CPUs with the same offset as the one used for the shared memory anchor.

CONFIGURATION

Before routines in this library can be called, the shared memory objects facility must be initialized by a call to **usrSmObjInit( )**, which is found in **target/config/comps/src/usrSmObj.c**. This is done automatically by VxWorks when the **INCLUDE_SM_OBJ** component is included.

ERROR OPTIONS

Various debug options can be selected for each partition using **memPartOptionsSet( )** and **smMemOptionsSet( )**. Two kinds of errors are detected: attempts to allocate more memory than is available, and bad blocks found when memory is freed. In both cases, options can be selected for system actions to take place when the error is detected: (1) return the error status, (2) log an error message and return the error status, or (3) log an error message and suspend the calling task.

One of the following options can be specified to determine the action to be taken when there is an attempt to allocate more memory than is available in the partition:

MEM_ALLOC_ERROR_RETURN
just return the error status to the calling task.

MEM_ALLOC_ERROR_LOG_MSG
log an error message and return the status to the calling task.

MEM_ALLOC_ERROR_LOG_AND_SUSPEND
log an error message and suspend the calling task.

The following option is specified by default to check every block freed to the partition. If this option is specified, **memPartFree( )** and **smMemFree( )** will make a consistency check of various pointers and values in the header of the block being freed.

MEM_BLOCK_CHECK
check each block freed.

One of the following options can be specified to determine the action to be taken when a bad block is detected when freed. These options apply only if the **MEM_BLOCK_CHECK** option is selected.

MEM_BLOCK_ERROR_RETURN
just return the status to the calling task.

MEM_BLOCK_ERROR_LOG_MSG
log an error message and return the status to the calling task.

MEM_BLOCK_ERROR_LOG_AND_SUSPEND
log an error message and suspend the calling task.

The default options when a shared partition is created are
MEM_ALLOC_ERROR_LOG_MSG, **MEM_BLOCK_CHECK**, **MEM_BLOCK_ERROR_RETURN**.

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

```
memPartOptionsSet (myPartId, MEM_ALLOC_ERROR_LOG_MSG |
                             MEM_BLOCK_CHECK |
                             MEM_BLOCK_ERROR_LOG_MSG);
```

AVAILABILITY
This module is distributed as a component of the unbundled shared memory objects support option, VxMP.

INCLUDE FILES
smMemLib.h

SEE ALSO
smMemShow, **memLib**, **memPartLib**, **smObjLib**, **usrSmObjInit( )**, *VxWorks Programmer's Guide: Shared Memory Objects*

smMemShow

NAME **smMemShow** – shared memory management show routines (VxMP Opt.)

ROUTINES **smMemShow( )** - show the shared memory system partition blocks and statistics (VxMP Opt.)

DESCRIPTION This library provides routines to show the statistics on a shared memory system partition.

General shared memory management routines are provided by **smMemLib**.

CONFIGURATION The routines in this library are included by default if the component **INCLUDE_SM_OBJ** is included.

AVAILABILITY This module is distributed as a component of the unbundled shared memory objects support option, VxMP.

INCLUDE FILES **smLib.h**, **smObjLib.h**, **smMemLib.h**

SEE ALSO **smMemLib**, *VxWorks Programmer's Guide: Shared Memory Objects*

smNameLib

NAME **smNameLib** – shared memory objects name database library (VxMP Opt.)

ROUTINES
smNameAdd( ) - add a name to the shared memory name database (VxMP Opt.)
smNameFind( ) - look up a shared memory object by name (VxMP Opt.)
smNameFindByValue( ) - look up a shared memory object by value (VxMP Opt.)
smNameRemove( ) - remove an object from the shared memory objects name database (VxMP Opt.)

DESCRIPTION This library provides facilities for managing the shared memory objects name database. The shared memory objects name database associates a name and object type with a value and makes that information available to all CPUs. A name is an arbitrary, null-terminated string. An object type is a small integer, and its value is a global (shared) ID or a global shared memory address.

Names are added to the shared memory name database with **smNameAdd( )**. They are removed by **smNameRemove( )**.

Objects in the database can be accessed by either name or value. The routine **smNameFind( )** searches the shared memory name database for an object of a specified name. The routine **smNameFindByValue( )** searches the shared memory name database for an object of a specified identifier or address.

Name database contents can be viewed using **smNameShow( )**.

The maximum number of names to be entered in the database is defined in the configuration parameter **SM_OBJ_MAX_NAME**. This value is used to determine the size of a dedicated shared memory partition from which name database fields are allocated.

The estimated memory size required for the name database can be calculated as follows:

```
name database pool size = SM_OBJ_MAX_NAME * 40 (bytes)
```

The display facility for the shared memory objects name database is provided by the **smNameShow** module.

EXAMPLE

The following code fragment allows a task on one CPU to enter the name, associated ID, and type of a created shared semaphore into the name database. Note that CPU numbers can belong to any CPU using the shared memory objects facility.

On CPU 1:

```
#include "vxWorks.h"
#include "semLib.h"
#include "smNameLib.h"
#include "semSmLib.h"
#include "stdio.h"
testSmSem1 (void)
    {
    SEM_ID smSemId;
    /* create a shared semaphore */
    if ((smSemId = semBSmCreate(SEM_Q_FIFO, SEM_EMPTY)) == NULL)
        {
        printf ("Shared semaphore creation error.");
        return (ERROR);
        }
    /*
     * make created semaphore Id available to all CPUs in
     * the system by entering its name in shared name database.
     */
    if (smNameAdd ("smSem", smSemId, T_SM_SEM_B) != OK )
        {
        printf ("Cannot add smSem into shared database.");
        return (ERROR);
        }
        ...
    /* now use the semaphore */
    semGive (smSemId);
        ...
```

S

```
        }
```

On CPU 2 :

```
    #include "vxWorks.h"
    #include "semLib.h"
    #include "smNameLib.h"
    #include "stdio.h"
    testSmSem2 (void)
        {
        SEM_ID smSemId;
        int    objType;  /* place holder for smNameFind() object type */
        /* get semaphore ID from name database */

        smNameFind ("smSem", (void **) &smSemId, &objType, WAIT_FOREVER);
            ...
        /* now that we have the shared semaphore ID, take it */

        semTake (smSemId, WAIT_FOREVER);
            ...
        }
```

CONFIGURATION Before routines in this library can be called, the shared memory object facility must be initialized by calling **usrSmObjInit( )**. This is done automatically during VxWorks initialization when the component **INCLUDE_SM_OBJ** is included.

AVAILABILITY This module is distributed as a component of the unbundled shared memory objects support option, VxMP.

INCLUDE FILES **smNameLib.h**

SEE ALSO **smNameShow**, **smObjLib**, **smObjShow**, **usrSmObjInit( )**, *VxWorks Programmer's Guide: Shared Memory Objects*

smNameShow

NAME **smNameShow** – shared memory objects name database show routines (VxMP Opt.)

ROUTINES **smNameShow( )** - show the contents of the shared memory objects name database (VxMP Opt.)

DESCRIPTION This library provides a routine to show the contents of the shared memory objects name database. The shared memory objects name database facility is provided by the **smNameLib** module.

CONFIGURATION The routines in this library are included by default if the component **INCLUDE_SM_OBJ** is included.

AVAILABILITY This module is distributed as a component of the unbundled shared memory objects support option, VxMP.

INCLUDE FILES **smNameLib.h**

SEE ALSO **smNameLib**, **smObjLib**, *VxWorks Programmer's Guide: Shared Memory Objects*

---

smNetLib

NAME **smNetLib** – VxWorks interface to shared memory network (backplane) driver

ROUTINES No Callable Routines.

DESCRIPTION This library implements the VxWorks-specific portions of the shared memory network interface driver. It provides the interface between VxWorks and the network driver modules (*e.g.*, how the OS initializes and attaches the driver, interrupt handling, *etc*.), as well as VxWorks-dependent system calls.

There are no user-callable routines.

The backplane master initializes the backplane shared memory and network structures by first calling **smNetInit( )**. Once the backplane has been initialized, all processors can be attached to the shared memory network via the **smNetAttach( )** routine. Both **smNetInit( )** and **smNetAttach( )** are called automatically during system initialization when backplane parameters are specified in the boot line.

For detailed information refer to *VxWorks Network Programmer's Guide: Data Link Layer Network Components.*

INCLUDE FILES **smNetLib.h**, **smPktLib.h**, **smUtilLib.h**

SEE ALSO **ifLib**, **if_sm**, *VxWorks Network Programmer's Guide*

smNetShow

NAME

smNetShow – shared memory network driver show routines

ROUTINES

smNetShow( ) - show information about a shared memory network

DESCRIPTION

This library provides show routines for the shared memory network interface driver.

The **smNetShow( )** routine is provided as a diagnostic aid to show current shared memory network status.

INCLUDE FILES

smNetLib.h, **smPktLib.h**

SEE ALSO

if_sm, **smNetLib**, **smPktLib**, *VxWorks Network Programmer's Guide*

smObjLib

NAME

smObjLib – shared memory objects library (VxMP Opt.)

ROUTINES

smObjLibInit( ) - install the shared memory objects facility
smObjSetup( ) - initialize the shared memory objects facility
smObjInit( ) - initialize a shared memory objects descriptor
smObjAttach( ) - attach the calling CPU to the shared memory objects facility
smObjLocalToGlobal( ) - convert a local address to a global address
smObjGlobalToLocal( ) - convert a global address to a local address
smObjTimeoutLogEnable( ) - control logging of failed attempts to take a spin-lock

DESCRIPTION

This library contains miscellaneous functions used by the shared memory objects facility (VxMP). Shared memory objects provide high-speed synchronization and communication among tasks running on separate CPUs that have access to a common shared memory. Shared memory objects are system objects (*e.g.*, semaphores and message queues) that can be used across processors.

The main uses of shared memory objects are inter-processor synchronization, mutual exclusion on multiprocessor shared data structures, and high-speed data exchange.

Routines for displaying shared memory objects statistics are provided by the **smObjShow** module.

SHARED MEMORY MASTER CPU

One CPU node acts as the shared memory objects master. This CPU initializes the shared memory area and sets up the shared memory anchor. These steps are performed by the master calling **smObjSetup( )**. This routine should be called only once by the master CPU. Usually **smObjSetup( )** is called from **usrSmObjInit( )**. (See *Configuration* below.)

Once **smObjSetup( )** has completed successfully, there is little functional difference between the master CPU and other CPUs using shared memory objects, except that the master is responsible for maintaining the heartbeat in the shared memory objects header.

ATTACHING TO SHARED MEMORY

Each CPU, master or non-master, that will use shared memory objects must attach itself to the shared memory objects facility, which must already be initialized.

Before it can attach to a shared memory region, each CPU must allocate and initialize a shared memory descriptor (**SM_DESC**), which describes the individual CPU's attachment to the shared memory objects facility. Since the shared memory descriptor is used only by the local CPU, it is not necessary for the descriptor itself to be located in shared memory. In fact, it is preferable for the descriptor to be allocated from the CPU's local memory, since local memory is usually more efficiently accessed.

The shared memory descriptor is initialized by calling **smObjInit( )**. This routine takes a number of parameters which specify the characteristics of the calling CPU and its access to shared memory.

Once the shared memory descriptor has been initialized, the CPU can attach itself to the shared memory region. This is done by calling **smObjAttach( )**.

When **smObjAttach( )** is called, it verifies that the shared memory anchor contains the value **SM_READY** and that the heartbeat located in the shared memory objects header is incrementing. If either of these conditions is not met, the routine will check periodically until either **SM_READY** or an incrementing heartbeat is recognized or a time limit is reached. The limit is expressed in seconds, and 600 seconds (10 minutes) is the default. If the time limit is reached before **SM_READY** or a heartbeat is found, **ERROR** is returned and **errno** is set to **S_smLib_DOWN**.

ADDRESS CONVERSION

This library also provides routines for converting between local and global shared memory addresses, **smObjLocalToGlobal( )** and **smObjGlobalToLocal( )**. A local shared memory address is the address required by the local CPU to reach a location in shared memory. A global shared memory address is a value common to all CPUs in the system used to reference a shared memory location. A global shared memory address is always an offset from the shared memory anchor.

SPIN-LOCK MECHANISM

The shared memory objects facilities use a spin-lock mechanism based on an indivisible read-modify-write (RMW) operation on a shared memory location which acts as a low-level mutual exclusion device. The spin-lock mechanism is called with a system-wide configuration parameter, **SM_OBJ_MAX_TRIES**, which specifies the maximum number of RMW tries on a spin-lock location.

Care must be taken that the number of RMW tries on a spin-lock on a particular CPU never reaches **SM_OBJ_MAX_TRIES**, otherwise system behavior becomes unpredictable. The default value should be sufficient for reliable operation.

The routine **smObjTimeoutLogEnable( )** can be used to enable or disable the printing of a message should a shared memory object call fail while trying to take a spin-lock.

RELATION TO BACKPLANE DRIVER

Shared memory objects and the shared memory network (backplane) driver use common underlying shared memory utilities. They also use the same anchor, the same shared memory header, and the same interrupt when they are used at the same time.

LIMITATIONS

A maximum of twenty CPUs can be used concurrently with shared memory objects. Each CPU in the system must have a hardware test-and-set (TAS) mechanism, which is called via the system-dependent routine **sysBusTas( )**.

The use of shared memory objects raises interrupt latency, because internal mechanisms lock interrupts while manipulating critical shared data structures. Interrupt latency does not depend on the number of objects or CPUs used.

GETTING STATUS INFORMATION

The routine **smObjShow( )** displays useful information regarding the current status of shared memory objects, including the number of tasks using shared objects, shared semaphores, and shared message queues, the number of names in the database, and also the maximum number of tries to get spin-lock access for the calling CPU.

CONFIGURATION

When the component **INCLUDE_SM_OBJ** is included, the init and setup routines in this library are called automatically during VxWorks initialization.

AVAILABILITY

This module is distributed as a component of the unbundled shared memory objects support option, VxMP.

INCLUDE FILES

smObjLib.h

SEE ALSO

smObjShow, **semSmLib**, **msgQSmLib**, **smMemLib**, **smNameLib**, **usrSmObjInit( )**, *VxWorks Programmer's Guide: Shared Memory Objects*

smObjShow

NAME **smObjShow** – shared memory objects show routines (VxMP Opt.)

ROUTINES **smObjShow( )** - display the current status of shared memory objects (VxMP Opt.)

DESCRIPTION This library provides routines to show shared memory object statistics, such as the current number of shared tasks, semaphores, message queues, *etc*.

CONFIGURATION The routines in this library are included by default if the component **INCLUDE_SM_OBJ** is included.

AVAILABILITY This module is distributed as a component of the unbundled shared memory objects support option, VxMP.

INCLUDE FILES **smObjLib.h**

SEE ALSO **smObjLib**, *VxWorks Programmer's Guide: Shared Memory Objects*

sntpcLib

NAME **sntpcLib** – Simple Network Time Protocol (SNTP) client library

ROUTINES **sntpcTimeGet( )** - retrieve the current time from a remote source

DESCRIPTION This library implements the client side of the Simple Network Time Protocol (SNTP), a protocol that allows a system to maintain the accuracy of its internal clock based on time values reported by one or more remote sources. The library is included in the VxWorks image if **INCLUDE_SNTPC** is defined at the time the image is built.

USER INTERFACE The **sntpcTimeGet( )** routine retrieves the time reported by a remote source and converts that value for POSIX-compliant clocks. The routine will either send a request and extract the time from the reply, or it will wait until a message is received from an SNTP/NTP server executing in broadcast mode.

INCLUDE FILES **sntpcLib.h**

SEE ALSO **clockLib**, RFC 1769

S

sntpsLib

NAME

sntpsLib – Simple Network Time Protocol (SNTP) server library

ROUTINES

sntpsClockSet( ) - assign a routine to access the reference clock
sntpsNsecToFraction( ) - convert portions of a second to NTP format
sntpsConfigSet( ) - change SNTP server broadcast settings

DESCRIPTION

This library implements the server side of the Simple Network Time Protocol (SNTP), a protocol that allows a system to maintain the accuracy of its internal clock based on time values reported by one or more remote sources. The library is included in the VxWorks image if **INCLUDE_SNTPS** is defined at the time the image is built.

USER INTERFACE

The routine **sntpsInit( )** is called automatically during system startup when the SNTP server library is included in the VxWorks image. Depending on the value of **SNTPS_MODE**, the server executes in either a passive or an active mode. When **SNTPS_MODE** is set to **SNTP_PASSIVE** (0x2), the server waits for requests from clients, and sends replies containing an NTP timestamp. When the mode is set to **SNTP_ACTIVE** (0x1), the server transmits NTP timestamp information at fixed intervals.

When executing in active mode, the SNTP server uses the **SNTPS_DSTADDR** and **SNTPS_INTERVAL** definitions to determine the target IP address and broadcast interval. By default, the server transmits the timestamp information to the local subnet broadcast address every 64 seconds. These settings can be changed with **sntpsConfigSet( )**. The SNTP server operating in active mode will still respond to client requests.

The **SNTP_PORT** definition in assigns the source and destination UDP port. The default port setting is 123 as specified by the relevant RFC. Finally, the SNTP server requires access to a reliable external time source. The **SNTPS_TIME_HOOK** constant specifies the name of a routine with the following interface:

```
STATUS sntpsTimeHook (int request, void *pBuffer);
```

This routine can be assigned directly by altering the value of **SNTPS_TIME_HOOK** or can be installed by a call to **sntpsClockSet( )**. The manual pages for **sntpsClockSet( )** describe the parameters and required operation of the timestamp retrieval routine. Until this routine is specified, the SNTP server will not provide timestamp information.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, the SNPT server can run in the kernel protection domain only. The **SNTPS_TIME_HOOK** MUST, if used, must reference a function in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

INCLUDE FILES

sntpsLib.h

SEE ALSO

sntpcLib, RFC 1769

sockLib

NAME

sockLib – generic socket library

ROUTINES

socket( ) - open a socket
bind( ) - bind a name to a socket
listen( ) - enable connections to a socket
accept( ) - accept a connection from a socket
connect( ) - initiate a connection to a socket
connectWithTimeout( ) - attempt socket connection within a specified duration
sendto( ) - send a message to a socket
send( ) - send data to a socket
sendmsg( ) - send a message to a socket
recvfrom( ) - receive a message from a socket
recv( ) - receive data from a socket
recvmsg( ) - receive a message from a socket
setsockopt( ) - set socket options
getsockopt( ) - get socket options
getsockname( ) - get a socket name
getpeername( ) - get the name of a connected peer
shutdown( ) - shut down a network connection

DESCRIPTION

This library provides UNIX BSD 4.4 compatible socket calls. Use these calls to open, close, read, and write sockets. These sockets can join processes on the same CPU or on different CPUs between which there is a network connection. The calling sequences of these routines are identical to their equivalents under UNIX BSD 4.4.

However, although the socket interface is compatible with VxWorks, the VxWorks environment does affect how you use sockets. Specifically, the globally accessible file descriptors available in the single address space world of VxWorks require that you take extra precautions when closing a file descriptor.

You must make sure that you do not close the file descriptor on which a task is pending during an **accept( )**. Although the **accept( )** on the closed file descriptor sometimes returns with an error, the **accept( )** can also fail to return at all. Thus, if you need to be able to close a socket connections file descriptor asynchronously, you may need to set up a semaphore-based locking mechanism that prevents the close while an **accept( )** is pending on the file descriptor.

ADDRESS FAMILY

VxWorks sockets support only the Internet Domain address family. Use **AF_INET** for the *domain* argument in subroutines that require it. There is no support for the UNIX Domain address family.

IOCTL FUNCTIONS

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

S

FIONBIO
Turns on/off non-blocking I/O.

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

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

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

SIOCATMARK
Reports whether there is out-of-band data to be read from the socket. On the return of **ioctl( )**, *atMark* is **TRUE** (1) if there is out-of-band data. Otherwise, it is **FALSE** (0).

```
status = ioctl (sFd, SIOCATMARK, &atMark);
```

To use this feature, include the following component: **INCLUDE_BSD_SOCKET**.

INCLUDE FILES **types.h**, **mbuf.h**, **socket.h**, **socketvar.h**

SEE ALSO **netLib**, *UNIX Network Programming*, *by W. Richard Stevens*

spyLib

NAME **spyLib** – spy CPU activity library

ROUTINES **spyLibInit( )** - initialize task CPU utilization tool package

DESCRIPTION This library provides a facility to monitor tasks' use of the CPU. The primary interface routine, **spy( )**, periodically calls **spyReport( )** to display the amount of CPU time utilized by each task, the amount of time spent at interrupt level, the amount of time spent in the kernel, and the amount of idle time. It also displays the total usage since the start of **spy( )** (or the last call to **spyClkStart( )**), and the change in usage since the last **spyReport( )**.

CPU usage can also be monitored manually by calling **spyClkStart( )** and **spyReport( )**, instead of **spy( )**. In this case, **spyReport( )** provides a one-time report of the same information provided by **spy( )**.

Data is gathered by an interrupt-level routine that is connected by **spyClkStart( )** to the auxiliary clock. Currently, this facility cannot be used with CPUs that have no auxiliary clock. Interrupts that are at a higher level than the auxiliary clock's interrupt level cannot be monitored.

All user interface routine except **spyLibInit( )** are available through **usrLib**.

EXAMPLE

The following call:

```
-> spy 10, 200
```

will generate a report in the following format every 10 seconds, gathering data at the rate of 200 times per second.

```
NAME          ENTRY        TID    PRI   total % (ticks)   delta % (ticks)
--------     --------     -----   ---   ---------------   ---------------
tExcTask     _excTask     fbb58     0     0% (       0)     0% (       0)
tLogTask     _logTask     fa6e0     0     0% (       0)     0% (       0)
tShell       _shell       e28a8     1     0% (       4)     0% (       0)
tRlogind     _rlogind     f08dc     2     0% (       0)     0% (       0)
tRlogOutTask _rlogOutTa   e93e0     2     2% (     173)     2% (      46)
tRlogInTask  _rlogInTas   e7f10     2     0% (       0)     0% (       0)
tSpyTask     _spyTask     ffe9c     5     1% (     116)     1% (      28)
tNetTask     _netTask     f3e2c    50     0% (       4)     0% (       1)
tPortmapd    _portmapd    ef240   100     0% (       0)     0% (       0)
KERNEL                                    1% (     105)     0% (      10)
INTERRUPT                                 0% (       0)     0% (       0)
IDLE                                     95% (    7990)    95% (    1998)
TOTAL                                    99% (    8337)    98% (    2083)
```

The "total" column reflects CPU activity since the initial call to **spy( )** or the last call to **spyClkStart( )**. The "delta" column reflects activity since the previous report. A call to **spyReport( )** will produce a single report; however, the initial auxiliary clock interrupts and data collection must first be started using **spyClkStart( )**.

Data collection/clock interrupts and periodic reporting are stopped by calling:

```
-> spyStop
```

INCLUDE FILES

spyLib.h

SEE ALSO

usrLib

S

symLib

NAME

symLib – symbol table subroutine library

ROUTINES

symLibInit( ) - initialize the symbol table library
symTblCreate( ) - create a symbol table
symTblDelete( ) - delete a symbol table
symAdd( ) - create and add a symbol to a symbol table, including a group number
symRemove( ) - remove a symbol from a symbol table

symFindByName( ) - look up a symbol by name
symFindByNameAndType( ) - look up a symbol by name and type
symByValueFind( ) - look up a symbol by value
symByValueAndTypeFind( ) - look up a symbol by value and type
symFindByValue( ) - look up a symbol by value
symFindByValueAndType( ) - look up a symbol by value and type
symEach( ) - call a routine to examine each entry in a symbol table

DESCRIPTION

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

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

If you use the **symLib** subroutines to manipulate the VxWorks system symbol table (whose ID is recorded in the global **sysSymTbl**), the values for **SYM_TYPE** are **SYM_UNDF**, **SYM_LOCAL**, **SYM_GLOBAL**, **SYM_ABS**, **SYM_TEXT**, **SYM_DATA**, **SYM_BSS**, and **SYM_COMM** (defined in **symbol.h**).

Tables are created with **symTblCreate( )**, which returns a symbol table ID. This ID serves as a handle for symbol table operations, including the adding to, removing from, and searching of tables. All operations on a symbol table are interlocked by means of a mutual-exclusion semaphore in the symbol table structure. 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, and a type. 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 next lower value than the specified value. The routines **symFindByNameAndType( )** and **symByValueAndTypeFind( )** allow the symbol type to be used as an additional criterion in the searches.

The routines **symFindByValue( )** and **symFindByValueAndType( )** are obsolete. They are replaced by the routines **symByValueFind( )** and **symByValueAndTypeFind( )**.

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 thus be much slower.

The routine **symEach( )** allows each 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 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( )** returns an error if there is an attempt to add a symbol with identical name and type. 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 by **symEach( )**.

The system symbol table (**sysSymTbl**) allows name clashes.

See the *VxWorks Programmmer's Guide* for more information about configuration, initialization, and use of the system symbol table.

INCLUDE FILES **symLib.h**

SEE ALSO **loadLib**

symSyncLib

NAME **symSyncLib** – host/target symbol table synchronization

ROUTINES
symSyncLibInit( ) - initialize host/target symbol table synchronization
symSyncTimeoutSet( ) - set WTX timeout

DESCRIPTION This module provides host/target symbol table synchronization. With synchronization, every module or symbol added to the run-time system from either the target or host side can be seen by facilities on both the target and the host. Symbol table synchronization makes it possible to use host tools to debug application modules loaded with the target loader or from a target file system. To enable synchronization, two actions must be performed:

1 The module is initialized by **symSyncLibInit( )**, which is called automatically when the configuration macro **INCLUDE_SYM_TBL_SYNC** is defined.

2 The target server is launched with the **-s** option.

If synchronization is enabled, **symSyncLib** spawns a synchronization task on the target, **tSymSync**. This task behaves as a WTX tool and attaches itself to the target server. When the task starts, it synchronizes target and host symbol tables so that every module loaded on the target before the target server was started can be seen by the host tools. This feature is particularly useful if VxWorks is started with a target-based startup script before the target server has been launched.

The **tSymSync** task synchronizes new symbols that are added by either the target or the host tools. The task waits for synchronization events on two channels: a WTX event from the host or a message queue addition from the target.

S

The **tSymSync** task, like all WTX tools, must be able to connect to the WTX registry. To make the WTX registry accessible from the target, do one of the following:

1 Boot the target from a host on the same subnet as the registry.

2 Start the registry on the same host the target boots from.

3 Add the needed routes with **routeAdd( )** calls, possibly in a startup script.

Neither the host tools nor the target loader wait for synchronization completion to return. To know when the synchronization is complete, you can wait for the corresponding event sent by the target server, or, if your target server was started with the **-V** option, it prints a message indicating synchronization has completed.

The event sent by the target server is of the following format:

```
SYNC_DONE syncType syncObj syncStatus
```

The following are examples of messages displayed by the target server indicating synchronization is complete:

```
Added target_modules         to target-server.....done
Added ttTest.o.68k           to target............done
```

If synchronization fails, the following message is displayed:

```
Added gopher.o               to target............failed
```

This error generally means that synchronization of the corresponding module or symbol is no longer possible because it no longer exists in the original symbol table. If so, it will be followed by:

```
Removed gopher.o             from target..........failed
```

Failure can also occur if a timeout is reached. Call **symSyncTimeoutSet( )** to modify the WTX timeout between the target synchronization task and the target server.

LIMITATIONS

Hardware: Because the synchronization task uses the WTX protocol to communicate with the target server, the target must include network facilities. Depending on how much synchronization is to be done (number of symbols to transfer), a reasonable throughput between the target server and target agent is required (the **wdbrpc** backend is recommended when large modules are to be loaded).

PERFORMANCE

The synchronization task requires some minor overhead in target routines **msgQSend( )**, **loadModule( )**, **symAdd( )**, and **symRemove( )**; however, if an application sends more than 15 synchronization events, it will fill the message queue and then need to wait for a synchronization event to be processed by **tSymSync**. Also, waiting for host synchronization events is done by polling; thus there may be some impact on performance if there are lower-priority tasks than **tSymSync**. If no more synchronization is needed, **tSymSync** can be suspended.

KNOWN PROBLEM Modules with undefined symbols that are loaded from the target are not synchronized; however, they are synchronized if they are loaded from the host.

SEE ALSO **tgtsvr**

sysLib

NAME **sysLib** – system-dependent library

ROUTINES

sysClkConnect( ) - connect a routine to the system clock interrupt
sysClkDisable( ) - turn off system clock interrupts
sysClkEnable( ) - turn on system clock interrupts
sysClkRateGet( ) - get the system clock rate
sysClkRateSet( ) - set the system clock rate
sysAuxClkConnect( ) - connect a routine to the auxiliary clock interrupt
sysAuxClkDisable( ) - turn off auxiliary clock interrupts
sysAuxClkEnable( ) - turn on auxiliary clock interrupts
sysAuxClkRateGet( ) - get the auxiliary clock rate
sysAuxClkRateSet( ) - set the auxiliary clock rate
sysIntDisable( ) - disable a bus interrupt level
sysIntEnable( ) - enable a bus interrupt level
sysBusIntAck( ) - acknowledge a bus interrupt
sysBusIntGen( ) - generate a bus interrupt
sysMailboxConnect( ) - connect a routine to the mailbox interrupt
sysMailboxEnable( ) - enable the mailbox interrupt
sysNvRamGet( ) - get the contents of non-volatile RAM
sysNvRamSet( ) - write to non-volatile RAM
sysModel( ) - return the model name of the CPU board
sysBspRev( ) - return the BSP version and revision number
sysHwInit( ) - initialize the system hardware
sysPhysMemTop( ) - get the address of the top of memory
sysMemTop( ) - get the address of the top of logical memory
sysToMonitor( ) - transfer control to the ROM monitor
sysProcNumGet( ) - get the processor number
sysProcNumSet( ) - set the processor number
sysBusTas( ) - test and set a location across the bus
sysScsiBusReset( ) - assert the RST line on the SCSI bus (Western Digital WD33C93 only)
sysScsiInit( ) - initialize an on-board SCSI port
sysScsiConfig( ) - system SCSI configuration
sysLocalToBusAdrs( ) - convert a local address to a bus address
sysBusToLocalAdrs( ) - convert a bus address to a local address
sysSerialHwInit( ) - initialize the BSP serial devices to a quiesent state

sysSerialHwInit2( ) - connect BSP serial device interrupts
sysSerialReset( ) - reset all SIO devices to a quiet state
sysSerialChanGet( ) - get the **SIO_CHAN** device associated with a serial channel
sysNanoDelay( ) - delay for specified number of nanoseconds

DESCRIPTION

This library provides board-specific routines.

NOTE: This is a generic reference entry for a BSP-specific library; this description contains general information only. For features and capabilities specific to the system library included in your BSP, see your BSP's reference entry for **sysLib**.

The file **sysLib.c** provides the board-level interface on which VxWorks and application code can be built in a hardware-independent manner. The functions addressed in this file include:

Initialization functions
- initialize the hardware to a known state
- identify the system
- initialize drivers, such as SCSI or custom drivers

Memory/address space functions
- get the on-board memory size
- make on-board memory accessible to external bus
- map local and bus address spaces
- enable/disable cache memory
- set/get nonvolatile RAM (NVRAM)
- define board's memory map (optional)
- virtual-to-physical memory map declarations for processors with MMUs

Bus interrupt functions
- enable/disable bus interrupt levels
- generate bus interrupts

Clock/timer functions
- enable/disable timer interrupts
- set the periodic rate of the timer

Mailbox/location monitor functions
- enable mailbox/location monitor interrupts for VME-based boards

The **sysLib** library does not support every feature of every board; a particular board may have various extensions to the capabilities described here. Conversely, some boards do not support every function provided by this library. Some boards provide some of the functions of this library by means of hardware switches, jumpers, or PALs, instead of software-controllable registers.

Typically, most functions in this library are not called by the user application directly. The configuration modules **usrConfig.c** and **bootConfig.c** are responsible for invoking the routines at the appropriate time. Device drivers may use some of the memory mapping routines and bus functions.

INCLUDE FILES **sysLib.h**

SEE ALSO *VxWorks Programmer's Guide: Configuration and Build*, BSP-specific reference entry for **sysLib**

tapeFsLib

NAME
tapeFsLib – tape sequential device file system library

ROUTINES
tapeFsDevInit( ) - associate a sequential device with tape volume functions
tapeFsInit( ) - initialize the tape volume library
tapeFsReadyChange( ) - notify **tapeFsLib** of a change in ready status
tapeFsVolUnmount( ) - disable a tape device volume

DESCRIPTION
This library provides basic services for tape devices that do not use a standard file or directory structure on tape. The tape volume is treated much like a large file. The tape may either be read or written. However, there is no high-level organization of the tape into files or directories, which must be provided by a higher-level layer.

USING THIS LIBRARY

The various routines provided by the VxWorks tape file system, or tapeFs, can be categorized into three broad groupings: general initialization, device initialization, and file system operation.

The **tapeFsInit( )** routine is the principal general initialization function; it needs to be called only once, regardless of how many tapeFs devices are used.

To initialize devices, **tapeFsDevInit( )** must be called for each tapeFs device.

Use of this library typically occurs through standard use of the I/O system routines **open( )**, **close( )**, **read( )**, **write( )** and **ioctl( )**. Besides these standard I/O system operations, several routines are provided to inform the file system of changes in the system environment. The **tapeFsVolUnmount( )** routine informs the file system that a particular device should be unmounted; any synchronization should be done prior to invocation of this routine, in preparation for a tape volume change. The **tapeFsReadyChange( )** routine is used to inform the file system that a tape may have been swapped and that the next tape operation should first remount the tape. Information about a ready-change is also obtained from the driver using the **SEQ_DEV** device structure. Note that **tapeFsVolUnmount( )** and **tapeFsReadyChange( )** should be called only after a file has been closed.

INITIALIZATION OF THE FILE SYSTEM

Before any other routines in **tapeFsLib** can be used, **tapeFsInit( )** must be called to initialize the library. This implementation of the tape file system assumes only one file descriptor per volume. However, this constraint can be changed in case a future implementation demands multiple file descriptors per volume.

During the **tapeFsInit( )** call, the tape device library is installed as a driver in the I/O system driver table. The driver number associated with it is then placed in a global variable, **tapeFsDrvNum**.

To enable this initialization, define **INCLUDE_TAPEFS** in the BSP, or simply start using the tape file system with a call to **tapeFsDevInit( )** and **tapeFsInit( )** will be called automatically if it has not been called before.

DEFINING A TAPE DEVICE

To use this library for a particular device, the device structure used by the device driver must contain, as the very first item, a sequential device description structure (**SEQ_DEV**). The **SEQ_DEV** must be initialized before calling **tapeFsDevInit( )**. The driver places in the **SEQ_DEV** structure the addresses of routines that it must supply: one that reads one or more blocks, one that writes one or more blocks, one that performs I/O control (**ioctl( )**) on the device, one that writes file marks on a tape, one that rewinds the tape volume, one that reserves a tape device for use, one that releases a tape device after use, one that mounts/unmounts a volume, one that spaces forward or backwards by blocks or file marks, one that erases the tape, one that resets the tape device, and one that checks the status of the device. The **SEQ_DEV** structure also contains fields that describe the physical configuration of the device. For more information about defining sequential devices, see the *VxWorks Programmer's Guide: I/O System.*

INITIALIZATION OF THE DEVICE

The **tapeFsDevInit( )** routine is used to associate a device with the **tapeFsLib** functions. The **volName** parameter expected by **tapeFsDevInit( )** is a pointer to a name string which identifies the device. This string serves as the pathname for I/O operations which operate on the device and appears in the I/O system device table, which can be displayed using **iosDevShow( )**.

The **pSeqDev** parameter expected by **tapeFsDevInit( )** is a pointer to the **SEQ_DEV** structure describing the device and containing the addresses of the required driver functions.

The **pTapeConfig** parameter is a pointer to a **TAPE_CONFIG** structure that contains information specifying how the tape device should be configured. The configuration items are fixed/variable block size, rewind/no-rewind device, and number of file marks to be written. For more information about the **TAPE_CONFIG** structure, look at the header file **tapeFsLib.h**.

The syntax of the **tapeFsDevInit( )** routine is as follows:

```
tapeFsDevInit
    (
    char *        volName,     /* name to be used for volume   */
    SEQ_DEV *     pSeqDev,     /* pointer to device descriptor */
    TAPE_CONFIG * pTapeConfig  /* pointer to tape config info  */
    )
```

When **tapeFsLib** receives a request from the I/O system, after **tapeFsDevInit( )** has been called, it calls the device driver routines (whose addresses were passed in the **SEQ_DEV** structure) to access the device.

OPENING AND CLOSING A FILE

A tape volume is opened by calling the I/O system routine **open( )**. A file can be opened only with the **O_RDONLY** or **O_WRONLY** flags. The **O_RDWR** mode is not used by this library. A call to **open( )** initializes the file descriptor buffer and state information, reserves the tape device, rewinds the tape device if it was configured as a rewind device, and mounts a volume. Once a tape volume has been opened, that tape device is reserved, disallowing any other system from accessing that device until the tape volume is closed. Also, the single file descriptor is marked "in use" until the file is closed, making sure that a file descriptor is not opened multiple times.

A tape device is closed by calling the I/O system routine **close( )**. Upon a **close( )** request, any unwritten buffers are flushed, the device is rewound (if it is a rewind device), and, finally, the device is released.

UNMOUNTING VOLUMES (CHANGING TAPES)

A tape volume should be unmounted before it is removed. When unmounting a volume, make sure that any open file is closed first. A tape may be unmounted by calling **tapeFsVolUnmount( )** directly.

If a file is open, it is not correct to change the medium and continue with the same file descriptor still open. Since tapeFs assumes only one file descriptor per device, to reuse that device, the file must be closed and opened later for the new tape volume.

Before **tapeFsVolUnmount( )** is called, the device should be synchronized by invoking the **ioctl( ) FIOSYNC** or **FIOFLUSH**. It is the responsibility of the higher-level layer to synchronize the tape file system before unmounting. Failure to synchronize the volume before unmounting may result in loss of data.

IOCTL FUNCTIONS

The VxWorks tape sequential device file system supports the following **ioctl( )** functions. The functions listed are defined in the header files **ioLib.h** and **tapeFsLib.h**.

FIOFLUSH
Writes all modified file descriptor buffers to the physical device.

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

FIOSYNC
Performs the same function as **FIOFLUSH**.

FIOBLKSIZEGET
Returns the value of the block size set on the physical device. This value is compared against the **sd_blkSize** value set in the **SEQ_DEV** device structure.

FIOBLKSIZESET
Sets a specified block size value on the physical device and also updates the value in the **SEQ_DEV** and **TAPE_VOL_DESC** structures, unless the supplied value is zero, in which case the device structures are updated but the device is not set to zero. This is because zero implies variable block operations, therefore the device block size is ignored.

MTIOCTOP

Allows use of the standard UNIX MTIO **ioctl** operations by means of the MTOP structure. The MTOP structure appears as follows:

```
typedef struct mtop
    {
    short       mt_op;                  /* operation */
    int         mt_count;               /* number of operations */
    } MTOP;
```

Use these **ioctl( )** operations as follows:

```
MTOP mtop;
mtop.mt_op    = MTWEOF;
mtop.mt_count = 1;
status = ioctl (fd, MTIOCTOP, (int) &mtop);
```

The permissible values for **mt_op** are:

MTWEOF

Writes an end-of-file record to tape. An end-of-file record is a file mark.

MTFSF

Forward space over a file mark and position the tape head in the gap between the file mark just skipped and the next data block. Any buffered data is flushed out to the tape if the tape is in write mode.

MTBSF

Backward space over a file mark and position the tape head in the gap preceding the file mark, that is, right before the file mark. Any buffered data is flushed out to the tape if the tape is in write mode.

MTFSR

Forward space over a data block and position the tape head in the gap between the block just skipped and the next block. Any buffered data is flushed out to the tape if the tape is in write mode.

MTBSR

Backward space over a data block and position the tape head right before the block just skipped. Any buffered data is flushed out to the tape if the tape is in write mode.

MTREW

Rewind the tape to the beginning of the medium. Any buffered data is flushed out to the tape if the tape is in write mode.

MTOFFL

Rewind and unload the tape. Any buffered data is flushed out to the tape if the tape is in write mode.

MTNOP

No operation, but check the status of the device, thus setting the appropriate

SEQ_DEV fields.

MTRETEN
Retention the tape. This command usually sets tape tension and can be used in either read or write mode. Any buffered data is flushed out to tape if the tape is in write mode.

MTERASE
Erase the entire tape and rewind it.

MTEOM
Position the tape at the end of the medium and unload the tape. Any buffered data is flushed out to the tape if the tape is in write mode.

INCLUDE FILES **tapeFsLib.h**

SEE ALSO **ioLib**, **iosLib**, *VxWorks Programmer's Guide: I/O System, Local File Systems*

tarLib

NAME **tarLib** – UNIX tar compatible library

ROUTINES
tarExtract( ) - extract all files from a tar formatted tape
tarArchive( ) - archive named file/dir onto tape in tar format
tarToc( ) - display all contents of a tar formatted tape

DESCRIPTION This library implements functions for archiving, extracting and listing of UNIX-compatible "tar" file archives. It can be used to archive and extract entire file hierarchies to/from archive files on local or remote disks, or directly to/from magnetic tapes.

SEE ALSO **dosFsLib**

CURRENT LIMITATIONS
This Tar utility does not handle MS-DOS file attributes, when used in conjunction with the MS-DOS file system. The maximum subdirectory depth supported by this library is 16, while the total maximum path name that can be handled by tar is limited at 100 characters.

taskArchLib

NAME **taskArchLib** – architecture-specific task management routines

ROUTINES
taskSRSet( ) - set the task status register (68K, MIPS, x86)
taskSRInit( ) - initialize the default task status register (MIPS)

DESCRIPTION This library provides architecture-specific task management routines that set and examine architecture-dependent registers. For information about architecture-independent task management facilities, see the manual entry for **taskLib**.

NOTE: There are no application-level routines in **taskArchLib** for SimSolaris, SimNT or SH.

INCLUDE FILES **regs.h**, **taskArchLib.h**

SEE ALSO **taskLib**

taskHookLib

NAME **taskHookLib** – task hook library

ROUTINES
taskHookInit( ) - initialize task hook facilities
taskCreateHookAdd( ) - add a routine to be called at every task create
taskCreateHookDelete( ) - delete a previously added task create routine
taskSwitchHookAdd( ) - add a routine to be called at every task switch
taskSwitchHookDelete( ) - delete a previously added task switch routine
taskDeleteHookAdd( ) - add a routine to be called at every task delete
taskDeleteHookDelete( ) - delete a previously added task delete routine

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

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

taskSwitchHookAdd( ) and **taskSwitchHookDelete( )**
Add and delete routines to be called when a task is switched.

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

This facility is used by **dbgLib** to provide task-specific breakpoints and single-stepping. It is used by **taskVarLib** for the "task variable" mechanism. It is also used by **fppLib** for floating-point coprocessor support.

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.

INCLUDE FILES **taskHookLib.h**

SEE ALSO **dbgLib**, **fppLib**, **taskLib**, **taskVarLib** *VxWorks Programmer's Guide: Basic OS*

taskHookShow

NAME **taskHookShow** – task hook show routines

ROUTINES
taskHookShowInit( ) - initialize the task hook show facility
taskCreateHookShow( ) - show the list of task create routines
taskSwitchHookShow( ) - show the list of task switch routines
taskDeleteHookShow( ) - show the list of task delete routines

DESCRIPTION This library provides routines which summarize the installed kernel hook routines. There is one routine dedicated to the display of each type of kernel hook: task operation, task switch, and task deletion.

The routine **taskHookShowInit( )** links the task hook show facility into the VxWorks system. It is called automatically when this show facility is configured into VxWorks using either of the following methods:

– If you use the configuration header files, define **INCLUDE_SHOW_ROUTINES** in **config.h**.

– If you use the Tornado project facility, select **INCLUDE_TASK_HOOK_SHOW**.

INCLUDE FILES **taskHookLib.h**

SEE ALSO **taskHookLib**, *VxWorks Programmer's Guide: Basic OS*

taskInfo

NAME **taskInfo** – task information library

ROUTINES
taskOptionsSet( ) - change task options
taskOptionsGet( ) - examine task options
taskRegsGet( ) - get a task's registers from the TCB
taskRegsSet( ) - set a task's registers
taskName( ) - get the name associated with a task ID
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
taskIdListGet( ) - get a list of active task IDs

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

Task information is crucial as a debugging aid and user-interface convenience during the development cycle of an application. The routines **taskOptionsGet( )**, **taskRegsGet( )**, **taskName( )**, **taskNameToId( )**, **taskIsReady( )**, **taskIsSuspended( )**, and **taskIdListGet( )** are used to obtain task information. Three routines -- **taskOptionsSet( )**, **taskRegsSet( )**, and **taskIdDefault( )** -- provide programmatic access to debugging features.

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

Task management and control routines are provided by **taskLib**. Higher-level task information display routines are provided by **taskShow**.

INCLUDE FILES **taskLib.h**

SEE ALSO **taskLib**, **taskShow**, **taskHookLib**, **taskVarLib**, **semLib**, **kernelLib**, *VxWorks Programmer's Guide: Basic OS*

taskLib

NAME

taskLib – task management library

ROUTINES

taskSpawn( ) - spawn a task
taskInit( ) - initialize a task with a stack at a specified address
taskActivate( ) - activate a task that has been initialized
exit( ) - exit a task (ANSI)
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
taskLock( ) - disable task rescheduling
taskUnlock( ) - enable task rescheduling
taskSafe( ) - make the calling task safe from deletion
taskUnsafe( ) - make the calling task unsafe from deletion
taskDelay( ) - delay a task from executing
taskIdSelf( ) - get the task ID of a running task
taskIdVerify( ) - verify the existence of a task
taskTcb( ) - get the task control block for a task ID

DESCRIPTION

This library provides the interface to the VxWorks task management facilities. Task control services are provided by the VxWorks kernel, which is comprised of **kernelLib**, **taskLib**, **semLib**, **tickLib**, **msgQLib**, and **wdLib**. Programmatic access to task information and debugging features is provided by **taskInfo**. Higher-level task information display routines are provided by **taskShow**.

TASK CREATION

Tasks are created with the general-purpose routine **taskSpawn( )**. Task creation consists of the following: allocation of memory for the stack and task control block (**WIND_TCB**), initialization of the **WIND_TCB**, and activation of the **WIND_TCB**. Special needs may require the use of the lower-level routines **taskInit( )** and **taskActivate( )**, which are the underlying primitives of **taskSpawn( )**.

Tasks in VxWorks execute in the most privileged state of the underlying architecture. In a shared address space, processor privilege offers no protection advantages and actually hinders performance.

There is no limit to the number of tasks created in VxWorks, as long as sufficient memory is available to satisfy allocation requirements.

The routine **sp( )** is provided in **usrLib** as a convenient abbreviation for spawning tasks. It calls **taskSpawn( )** with default parameters.

TASK DELETION

If a task exits its "main" routine, specified during task creation, the kernel implicitly calls **exit( )** to delete the task. Tasks can be explicitly deleted with the **taskDelete( )** or **exit( )** routine.

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

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

The **sigLib** facility may also be used to allow a task to execute clean-up code before actually expiring.

TASK CONTROL

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

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

TASK SCHEDULING

VxWorks schedules tasks on the basis of priority. Tasks may have priorities ranging from 0, the highest priority, to 255, the lowest priority. The priority of a task in VxWorks is dynamic, and an existing task's priority can be changed using **taskPrioritySet( )**.

INCLUDE FILES

taskLib.h

SEE ALSO

taskInfo, **taskShow**, **taskHookLib**, **taskVarLib**, **semLib**, **semMLib**, **kernelLib**, *VxWorks Programmer's Guide: Basic OS*

taskShow

NAME **taskShow** – task show routines

ROUTINES

taskShowInit( ) - initialize the task show routine facility
taskInfoGet( ) - get information about a task
taskShow( ) - display task information from TCBs
taskRegsShow( ) - display the contents of a task's registers
taskStatusString( ) - get a task's status as a string

DESCRIPTION

This library provides routines to show task-related information, such as register values, task status, *etc*.

The **taskShowInit( )** routine links the task show facility into the VxWorks system. It is called automatically when this show facility is configured into VxWorks using either of the following methods:

– If you use the configuration header files, define **INCLUDE_SHOW_ROUTINES** in **config.h**.

– If you use the Tornado project facility, select **INCLUDE_TASK_SHOW**.

Task information is crucial as a debugging aid and user-interface convenience during the development cycle of an application. The routines **taskInfoGet( )**, **taskShow( )**, **taskRegsShow( )**, and **taskStatusString( )** are used to display task information.

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**. Programmatic access to task information and debugging features is provided by **taskInfo**.

INCLUDE FILES **taskLib.h**

SEE ALSO **taskLib**, **taskInfo**, **taskHookLib**, **taskVarLib**, **semLib**, **kernelLib**, *VxWorks Programmer's Guide: Basic OS, Target Shell, Tornado User's Guide: Shell*

taskVarLib

NAME **taskVarLib** – task variables support library

ROUTINES
taskVarInit( ) - initialize the task variables facility
taskVarAdd( ) - add a task variable to a task
taskVarDelete( ) - remove a task variable from a task
taskVarGet( ) - get the value of a task variable
taskVarSet( ) - set the value of a task variable
taskVarInfo( ) - get a list of task variables of a task

DESCRIPTION
VxWorks provides a facility called "task variables," which allows 4-byte variables to be added to a task's context, and the variables' values to be switched each time a task switch occurs to or from the calling task. Typically, several tasks declare the same variable (4-byte memory location) as a task variable and treat that memory location as their own private variable. For example, this facility can be used when a routine must be spawned more than once as several simultaneous tasks.

The routines **taskVarAdd( )** and **taskVarDelete( )** are used to add or delete a task variable. The routines **taskVarGet( )** and **taskVarSet( )** are used to get or set the value of a task variable.

NOTE: If you are using task variables in a task delete hook (see **taskHookLib**), refer to the manual entry for **taskVarInit( )** for warnings on proper usage.

INCLUDE FILES **taskVarLib.h**

SEE ALSO **taskHookLib**, *VxWorks Programmer's Guide: Basic OS*

T

tcpShow

NAME **tcpShow** – TCP information display routines

ROUTINES
tcpShowInit( ) - initialize TCP show routines
tcpDebugShow( ) - display debugging information for the TCP protocol
tcpstatShow( ) - display all statistics for the TCP protocol

DESCRIPTION
This library provides routines to show TCP related statistics.

Interpreting these statistics requires detailed knowledge of Internet network protocols. Information on these protocols can be found in the following books:

TCP/IP Illustrated Volume II, The Implementation, by Richard Stevens

The Design and Implementation of the 4.4 BSD UNIX Operating System, by Leffler, McKusick, Karels and Quarterman

The **tcpShowInit( )** routine links the TCP show facility into the VxWorks system. This is performed automatically if **INCLUDE_TCP_SHOW** is defined.

SEE ALSO **netLib**, **netShow**

telnetdLib

NAME **telnetdLib** – server library

ROUTINES

telnetdInit( ) - initialize the **telnet** services
telnetdParserSet( ) - specify a command interpreter for **telnet** sessions
telnetdStart( ) - initialize the **telnet** services
telnetdExit( ) - close an active **telnet** session
telnetdStaticTaskInitializationGet( ) - report whether tasks were pre-started by **telnetd**

DESCRIPTION

The **telnet** protocol enables users on remote systems to login to VxWorks.

This library implements a **telnet** server which accepts remote **telnet** login requests and transfers input and output data between a command interpreter and the remote user. The default configuration redirects the input and output from the VxWorks shell if available. The **telnetdParserSet( )** routine allows the installation of an alternative command interpreter to handle the remote input and provide the output responses. If **INCLUDE_SHELL** is not defined, installing a command interpreter is required.

The **telnetdInit( )** routine initializes the **telnet** service when **INCLUDE_TELNET** is defined. If **INCLUDE_SHELL** is also defined, the **telnetdStart( )** routine automatically starts the server. Client sessions will connect to the shell, which only supports one client at a time.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, the **telnet** server runs within the kernel protection domain only. This restriction does not apply under non-AE versions of VxWorks.

INCLUDE FILES **telnetLib.h**

SEE ALSO **rlogLib**

tffsConfig

NAME

tffsConfig – TrueFFS configuration file for VxWorks

ROUTINES

tffsShowAll( ) - show device information on all socket interfaces
tffsShow( ) - show device information on a specific socket interface
tffsBootImagePut( ) - write to the boot-image region of the flash device

DESCRIPTION

This source file, with the help of **sysTffs.c**, configures TrueFFS for VxWorks. The functions defined here are generic to all BSPs. To include these functions in the BSP-specific module, the BSP's **sysTffs.c** file includes this file. Within the **sysTffs.c** file, define statements determine which functions from the **tffsConfig.c** file are ultimately included in TrueFFS.

The only externally callable routines defined in this file are **tffsShow( )**, **tffsShowAll( )**, and **tffsBootImagePut( )**. You can exclude the show utilities if you edit **config.h** and undefine **INCLUDE_SHOW_ROUTINES**. You can exclude **tffsBootImagePut( )** if you edit **sysTffs.c** and undefine **INCLUDE_TFFS_BOOT_IMAGE**. (If you find these utilities are missing and you want them included, edit **config.h** and define **INCLUDE_SHOW_ROUTINES** and **INCLUDE_TFFS_BOOT_IMAGE**.)

If you wish to include only the TrueFFS specific show routines you could define **INCLUDE_TFFS_SHOW** instead of **INCLUDE_SHOW_ROUTINES** in **config.h**.

However, for the most part, these externally callable routines are only a small part of the TrueFFS configuration needs handled by this file. The routines internal to this file make calls into the MTDs and translation layer modules of TrueFFS. At link time, resolving the symbols associated with these calls pulls MTD and translation layer modules into VxWorks.

However, each of these calls to the MTDs and the translation layer modules is only conditionally included. The constants that control the includes are defined in **sysTffs.c**. To exclude an MTD or translation layer module, you edit **sysTffs.c**, undefine the appropriate constant, and rebuild **sysTffs.o**. These constants are described in the reference entry for **sysTffs**.

INCLUDE FILES

stdcomp.h

T

tffsDrv

NAME

tffsDrv – TrueFFS interface for VxWorks

ROUTINES

tffsDrv( ) - initialize the TrueFFS system
tffsDevCreate( ) - create a TrueFFS block device suitable for use with dosFs
tffsDevOptionsSet( ) - set TrueFFS volume options
tffsDevFormat( ) - format a flash device for use with TrueFFS
tffsRawio( ) - low level I/O access to flash components

DESCRIPTION

This module defines the routines that VxWorks uses to create a TrueFFS block device. Using this block device, dosFs can access a board-resident flash memory array or a flash memory card (in the PCMCIA slot) just as if it was a standard disk drive. Also defined in this file are functions that you can use to format the flash medium, as well as well as functions that handle the low-level I/O to the device.

To include TrueFFS for Tornado in a VxWorks image, you must edit your BSP's **config.h** and define **INCLUDE_TFFS**, or, for some hardware, **INCLUDE_PCMCIA**. If you define **INCLUDE_TFFS**, this configures **usrRoot( )** to call **tffsDrv( )**. If you defined **INCLUDE_PCMCIA**, the call to **tffsDrv( )** is made from **pccardTffsEnabler( )**. The call to **tffsDrv( )** sets up the structures, global variables, and mutual exclusion semaphore needed to manage TrueFFS. This call to **tffsDrv( )** also registers socket component drivers for each flash device found attached to the target.

These socket component drivers are not quite block devices, but they are an essential layer within TrueFFS. Their function is to manage the hardware interface to the flash device, and they are intelligent enough to handle formatting and raw I/O requests to the flash device. The other two layers within TrueFFS are known as the translation layer and the MTD (the Memory Technology Driver). The translation layer of TrueFFS implements the error recover and wear-leveling features of TrueFFS. The MTD implements the low-level programming (map, read, write, and erase) of the flash medium.

To implement the socket layer, each BSP that supports TrueFFS includes a **sysTffs.c** file. This file contains the code that defines the socket component driver. This file also contains a set of defines that you can use to configure which translation layer modules and MTDs are included in TrueFFS. Which translation layer modules and MTDs you should include depends on which types of flash devices you need to support. Currently, there are three basic flash memory technologies, NAND-based, NOR-based, and SSFDC. Within **sysTffs.c**, define:

INCLUDE_TL_NFTL
To include the NAND-based translation layer module.

INCLUDE_TL_FTL
To include the NOR-based translation layer module.

INCLUDE_TL_SSFDC
To include the SSFDC-appropriate translation layer module.

To support these different technologies, TrueFFS ships with three different implementations of the translation layer. Optionally, TrueFFS can include all three modules. TrueFFS later binds the appropriate translation layer module to the flash device when it registers a socket component driver for the device.

Within these three basic flash device categories there are still other differences (largely manufacturer-specific). These differences have no impact on the translation layer. However, they do make a difference for the MTD. Thus, TrueFFS ships with eight different MTDs that can support a variety of flash devices from Intel, Sharp, Samsung, National, Toshiba, AMD, and Fujitsu. Within **sysTffs.c**, define:

INCLUDE_MTD_I28F016
For Intel 28f016 flash devices.

INCLUDE_MTD_I28F008
For Intel 28f008 flash devices.

INCLUDE_MTD_I28F008_BAJA
For Intel 28f008 flash devices on the Heurikon Baja 4000.

INCLUDE_MTD_AMD
For AMD, Fujitsu: 29F0{40,80,16} 8-bit flash devices.

INCLUDE_MTD_CDSN
For Toshiba, Samsung: NAND CDSN flash devices.

INCLUDE_MTD_DOC2
For Toshiba, Samsung: NAND DOC flash devices.

INCLUDE_MTD_CFISCS
For CFI/SCS flash devices.

INCLUDE_MTD_WAMD
For AMD, Fujitsu 29F0{40,80,16} 16-bit flash devices.

The socket component driver and the MTDs are provided in source form. If you need to write your own socket driver or MTD, use these working drivers as a model for your own.

T

EXTERNALLY CALLABLE ROUTINES

Most of the routines defined in this file are accessible through the I/O system only. However, four routines are callable externally. These are: **tffsDrv( )**, **tffsDevCreate( )**, **tffsDevFormat( )**, and **tffsRawio( )**.

The first routine called from this library must be **tffsDrv( )**. Call this routine exactly once. Normally, this is handled automatically for you from within **usrRoot( )**, if **INCLUDE_TFFS** is defined, or from within **pccardTffsEnabler( )**, if **INCLUDE_PCMCIA** is defined.

Internally, this call to **tffsDrv( )** registers socket component drivers for all the flash devices connected to your system. After registering a socket component driver for the device, TrueFFS can support calls to **tffsDevFormat( )** or **tffsRawio( )**. However, before you can mount dosFs on the flash device, you must call **tffsDevCreate( )**. This call creates a block device on top of the socket component driver, but does not mount dosFs on the device. Because mounting dosFs on the device is what you will want to do most of the time, the **sysTffs.c** file defines a helper function, **usrTffsConfig( )**. Internally, this function calls **tffsDevCreate( )** and then does everything necessary (such as calling the **dosFsDevInit( )** routine) to mount dosFs on the resulting block device.

LOW LEVEL I/O

Normally, you should handle your I/O to the flash device using dosFs. However, there are situations when that level of indirection is a problem. To handle such situations, this library defines **tffsRawio( )**. Using this function, you can bypass both dosFs and the TrueFFS translation services to program the flash medium directly.

However, you should not try to program the flash device directly unless you are intimately familiar with the physical limits of your flash device as well as with how TrueFFS formats the flash medium. Otherwise you risk not only corrupting the medium entirely but permanently damaging the flash device.

If all you need to do is write a boot image to the flash device, use the **tffsBootImagePut( )** utility instead of **tffsRawio( )**. This function provides safer access to the flash medium.

IOCTL

This driver responds to all ioctl codes by setting a global error flag. Do not attempt to format a flash drive using ioctl calls.

INCLUDE FILES

tffsDrv.h, **fatlite.h**

tftpdLib

NAME

tftpdLib – Trivial File Transfer Protocol server library

ROUTINES

tftpdInit( ) - initialize the TFTP server task
tftpdTask( ) - TFTP server daemon task
tftpdDirectoryAdd( ) - add a directory to the access list
tftpdDirectoryRemove( ) - delete a directory from the access list

DESCRIPTION

This library implements the VxWorks Trivial File Transfer Protocol (TFTP) server module. The server can respond to both read and write requests. It is started by a call to **tftpdInit( )**.

The server has access to a list of directories that can either be provided in the initial call to **tftpdInit( )** or changed dynamically using the **tftpdDirectoryAdd( )** and **tftpDirectoryRemove( )** calls. Requests for files not in the directory trees specified in the access list will be rejected, unless the list is empty, in which case all requests will be allowed. By default, the access list contains the directory given in the global variable **tftpdDirectory**. It is possible to remove the default by calling **tftpdDirectoryRemove( )**.

For specific information about the TFTP protocol, see RFC 783, "TFTP Protocol."

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can run the TFTP server in the kernel protection domain only. This restriction does not apply under non-AE versions of VxWorks.

INCLUDE FILES **tftpdLib.h**, **tftpLib.h**

SEE ALSO **tftpLib**, RFC 783 "TFTP Protocol"

tftpLib

NAME **tftpLib** – Trivial File Transfer Protocol (TFTP) client library

ROUTINES

tftpXfer( ) - transfer a file via TFTP using a stream interface
tftpCopy( ) - transfer a file via TFTP
tftpInit( ) - initialize a TFTP session
tftpModeSet( ) - set the TFTP transfer mode
tftpPeerSet( ) - set the TFTP server address
tftpPut( ) - put a file to a remote system
tftpGet( ) - get a file from a remote system
tftpInfoShow( ) - get TFTP status information
tftpQuit( ) - quit a TFTP session
tftpSend( ) - send a TFTP message to the remote system

DESCRIPTION

This library implements the VxWorks Trivial File Transfer Protocol (TFTP) client library. TFTP is a simple file transfer protocol (hence the name "trivial") implemented over UDP. TFTP was designed to be small and easy to implement. Therefore, it is limited in functionality in comparison with other file transfer protocols, such as FTP. TFTP provides only the read/write capability to and from a remote server.

TFTP provides no user authentication. Therefore, the remote files must have "loose" permissions before requests for file access will be granted by the remote TFTP server. This means that the files to be read must be publicly readable, and files to be written must exist and be publicly writable). Some TFTP servers offer a secure option (-s) that specifies a directory where the TFTP server is rooted. Refer to the host manuals for more information about a particular TFTP server.

HIGH-LEVEL INTERFACE

The **tftpLib** library has two levels of interface. The tasks **tftpXfer( )** and **tftpCopy( )**

operate at the highest level and are the main call interfaces. The **tftpXfer( )** routine provides a stream interface to TFTP. That is, it spawns a task to perform the TFTP transfer and provides a descriptor from which data can be transferred interactively. The **tftpXfer( )** interface is similar to **ftpXfer( )** in **ftpLib**. The **tftpCopy( )** routine transfers a remote file to or from a passed file (descriptor).

LOW-LEVEL INTERFACE

The lower-level interface is made up of various routines that act on a TFTP session. Each TFTP session is defined by a TFTP descriptor. These routines include:

tftpInit( ) to initialize a session;
tftpModeSet( ) to set the transfer mode;
tftpPeerSet( ) to set a peer/server address;
tftpPut( ) to put a file to the remote system;
tftpGet( ) to get file from remote system;
tftpInfoShow( ) to show status information; and
tftpQuit( ) to quit a TFTP session.

EXAMPLE

The following code provides an example of how to use the lower-level routines. It implements roughly the same function as **tftpCopy( )**.

```
char *          pHost;
int             port;
char *          pFilename;
char *          pCommand;
char *          pMode;
int             fd;
TFTP_DESC *     pTftpDesc;
int             status;
if ((pTftpDesc = tftpInit ()) == NULL)
    return (ERROR);
if ((tftpPeerSet (pTftpDesc, pHost, port) == ERROR) ||
    (tftpModeSet (pTftpDesc, pMode) == ERROR))
    {
    (void) tftpQuit (pTftpDesc);
    return (ERROR);
    }
if (strcmp (pCommand, "get") == 0)
    {
    status = tftpGet (pTftpDesc, pFilename, fd, TFTP_CLIENT);
    }
else if (strcmp (pCommand, "put") == 0)
    {
    status =  tftpPut (pTftpDesc, pFilename, fd, TFTP_CLIENT);
    }
else
```

```
        {
        errno = S_tftpLib_INVALID_COMMAND;
        status = ERROR;
        }
    (void) tftpQuit (pTftpDesc);
```

To use this feature, include the following component: **INCLUDE_TFTP_CLIENT**

INCLUDE FILES **tftpLib.h**

SEE ALSO **tftpdLib**

tickLib

NAME **tickLib** – clock tick support library

ROUTINES
tickAnnounce( ) - announce a clock tick to the kernel
tickSet( ) - set the value of the kernel's tick counter
tickGet( ) - get the value of the kernel's tick counter

DESCRIPTION This library is the interface to the VxWorks kernel routines that announce a clock tick to the kernel, get the current time in ticks, and set the current time in ticks.

Kernel facilities that rely on clock ticks include **taskDelay( )**, **wdStart( )**, **kernelTimeslice( )**, and semaphore timeouts. In each case, the specified timeout is relative to the current time, also referred to as "time to fire." Relative timeouts are not affected by calls to **tickSet( )**, which only changes absolute time. The routines **tickSet( )** and **tickGet( )** keep track of absolute time in isolation from the rest of the kernel.

Time-of-day clocks or other auxiliary time bases are preferable for lengthy timeouts of days or more. The accuracy of such time bases is greater, and some external time bases even calibrate themselves periodically.

INCLUDE FILES **tickLib.h**

SEE ALSO **kernelLib**, **taskLib**, **semLib**, **wdLib**, *VxWorks Programmer's Guide: Basic OS*

timerLib

NAME

timerLib – timer library (POSIX)

ROUTINES

timer_cancel( ) - cancel a timer
timer_connect( ) - connect a user routine to the timer signal
timer_create( ) - allocate a timer using the specified clock for a timing base (POSIX)
timer_delete( ) - remove a previously created timer (POSIX)
timer_gettime( ) - get the remaining time before expiration and the reload value (POSIX)
timer_getoverrun( ) - return the timer expiration overrun (POSIX)
timer_settime( ) - set the time until the next expiration and arm timer (POSIX)
nanosleep( ) - suspend the current task until the time interval elapses (POSIX)
sleep( ) - delay for a specified amount of time
alarm( ) - set an alarm clock for delivery of a signal

DESCRIPTION

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

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

The **clockLib** library should be installed and **clock_settime( )** set before the use of any timer routines.

ADDITIONS

Two non-POSIX functions are provided for user convenience:

– **timer_cancel( )** quickly disables a timer by calling **timer_settime( )**.
– **timer_connect( )** easily hooks up a user routine by calling **sigaction( )**.

CLARIFICATIONS

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

When a timer expires and the task has previously exited, **logMsg( )** indicates the expected task is not present. Similarly, **logMsg( )** indicates when a task arms a timer without installing a signal handler. Timers may be armed but not created or deleted at interrupt level.

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).

INCLUDE FILES

timers.h

SEE ALSO

clockLib, **sigaction( )**, POSIX 1003.1b documentation, *VxWorks Programmer's Guide: Basic OS*

timexLib

NAME

timexLib – execution timer facilities

ROUTINES

timexInit( ) - include the execution timer library
timexClear( ) - clear the list of function calls to be timed
timexFunc( ) - specify functions to be timed
timexHelp( ) - display synopsis of execution timer facilities
timex( ) - time a single execution of a function or functions
timexN( ) - time repeated executions of a function or group of functions
timexPost( ) - specify functions to be called after timing
timexPre( ) - specify functions to be called prior to timing
timexShow( ) - display the list of function calls to be timed

DESCRIPTION

This library contains routines for timing the execution of programs, individual functions, and groups of functions. The VxWorks system clock is used as a time base. Functions that have a short execution time relative to this time base can be called repeatedly to establish an average execution time with an acceptable percentage of error.

Up to four functions can be specified to be timed as a group. Additionally, sets of up to four functions can be specified as pre- or post-timing functions, to be executed before and after the timed functions. The routines **timexPre( )** and **timexPost( )** are used to specify the pre- and post-timing functions, while **timexFunc( )** specifies the functions to be timed.

The routine **timex( )** is used to time a single execution of a function or group of functions. If called with no arguments, **timex( )** uses the functions in the lists created by calls to **timexPre( )**, **timexPost( )**, and **timexFunc( )**. If called with arguments, **timex( )** times the function specified, instead of the previous list. The routine **timexN( )** works in the same manner as **timex( )** except that it iterates the function calls to be timed.

EXAMPLES

The routine **timex( )** can be used to obtain the execution time of a single routine:

```
-> timex myFunc, myArg1, myArg2, ...
```

The routine **timexN( )** calls a function repeatedly until a 2% or better tolerance is obtained:

```
-> timexN myFunc, myArg1, myArg2, ...
```

The routines **timexPre( )**, **timexPost( )**, and **timexFunc( )** are used to specify a list of functions to be executed as a group:

```
-> timexPre 0, myPreFunc1, preArg1, preArg2, ...
-> timexPre 1, myPreFunc2, preArg1, preArg2, ...
-> timexFunc 0, myFunc1, myArg1, myArg2, ...
-> timexFunc 1, myFunc2, myArg1, myArg2, ...
-> timexFunc 2, myFunc3, myArg1, myArg2, ...
-> timexPost 0, myPostFunc, postArg1, postArg2, ...
```

The list is executed by calling **timex( )** or **timexN( )** without arguments:

```
-> timex
```

or:

```
-> timexN
```

In this example, *myPreFunc1* and *myPreFunc2* are called with their respective arguments. *myFunc1*, *myFunc2*, and *myFunc3* are then called in sequence and timed. If **timexN( )** was used, the sequence is called repeatedly until a 2% or better error tolerance is achieved. Finally, *myPostFunc* is called with its arguments. The timing results are reported after all post-timing functions are called.

NOTE: The timings measure the execution time of the routine body, without the usual subroutine entry and exit code (usually LINK, UNLINK, and RTS instructions). Also, the time required to set up the arguments and call the routines is not included in the reported times. This is because these timing routines automatically calibrate themselves by timing the invocation of a null routine, and thereafter subtracting that constant overhead.

INCLUDE FILES **timexLib.h**

SEE ALSO **spyLib**

trgLib

NAME **trgLib** – trigger events control library

ROUTINES

trgLibInit( ) - initialize the triggering library
trgWorkQReset( ) - resets the trigger work queue task and queue
trgAdd( ) - add a new trigger to the trigger list
trgDelete( ) - delete a trigger from the trigger list
trgOn( ) - set triggering on
trgOff( ) - set triggering off
trgEnable( ) - enable a trigger
trgDisable( ) - turn a trigger off
trgChainSet( ) - chains two triggers
trgEvent( ) - trigger a user-defined event

DESCRIPTION This library provides the interface for triggering events. The routines provide tools for creating, deleting, and controlling triggers. However, in most cases it is preferable to use the GUI to create and manage triggers, since all order and dependency factors are automatically accounted for there.

The event types are defined as in WindView. Triggering and WindView share the same instrumentation points. Furthermore, one of the main uses of triggering is to start and stop WindView instrumentation. Triggering is started by the routine **trgOn( )**, which sets the shared variable **evtAction**. Once the variable is set, when an instrumented point is hit, **trgCheck( )** is called. The routine looks for triggers that apply to this event. The routine **trgOff( )** stops triggering. The routine **trgEnable( )** enables a specific trigger that was previously disabled with **trgDisable( )**. (At creation time all triggers are enabled by default.) This routine also checks the number of triggers currently enabled, and when this is zero, it turns triggering off.

NOTE: It is important to create a trigger before calling **trgOn( )**. **trgOn( )** checks the trigger list to see if there is at least one trigger there, and if not, it exits without setting **evtAction**.

INCLUDE FILES **trgLibP.h**

SEE ALSO *WindView User's Guide*

trgShow

NAME **trgShow** – trigger show routine

ROUTINES
trgShowInit( ) - initialize the trigger show facility
trgShow( ) - show trigger information

DESCRIPTION This library provides routines to show event triggering information, such as list of triggers, associated actions, trigger states, and so on.

The routine **trgShowInit( )** links the triggering show facility into the VxWorks system. It is called automatically when **INCLUDE_TRIGGER_SHOW** is defined.

SEE ALSO **trgLib**

ttyDrv

NAME

ttyDrv – provide terminal device access to serial channels

ROUTINES

ttyDrv( ) - initialize the *tty* driver
ttyDevCreate( ) - create a VxWorks device for a serial channel

DESCRIPTION

This library provides the OS-dependent functionality of a serial device, including canonical processing and the interface to the VxWorks I/O system.

The BSP provides "raw" serial channels which are accessed via an **SIO_CHAN** data structure. These raw devices provide only low level access to the devices to send and receive characters. This library builds on that functionality by allowing the serial channels to be accessed via the VxWorks I/O system using the standard read/write interface. It also provides the canonical processing support of **tyLib**.

The routines in this library are typically called by **usrRoot( )** in **usrConfig.c** to create VxWorks serial devices at system startup time.

INCLUDE FILES

ttyLib.h

SEE ALSO

tyLib, **sioLib.h**

tyLib

NAME

tyLib – *tty* driver support library

ROUTINES

tyDevInit( ) - initialize the *tty* device descriptor
tyDevRemove( ) - remove the *tty* device descriptor
tyAbortFuncSet( ) - set the abort function
tyAbortSet( ) - change the abort character
tyBackspaceSet( ) - change the backspace character
tyDeleteLineSet( ) - change the line-delete character
tyEOFSet( ) - change the end-of-file character
tyMonitorTrapSet( ) - change the trap-to-monitor character
tyIoctl( ) - handle device control requests
tyWrite( ) - do a task-level write for a *tty* device
tyRead( ) - do a task-level read for a *tty* device
tyITx( ) - interrupt-level output
tyIRd( ) - interrupt-level input

DESCRIPTION

This library provides routines used to implement drivers for serial devices. It provides all the necessary device-independent functions of a normal serial channel, including:

– ring buffering of input and output

– raw mode

– optional line mode with backspace and line-delete functions

– optional processing of X-on/X-off

– optional RETURN/LINEFEED conversion

– optional echoing of input characters

– optional stripping of the parity bit from 8-bit input

– optional special characters for shell abort and system restart

Most of the routines in this library are called only by device drivers. Functions that normally might be called by an application or interactive user are the routines to set special characters, **ty...Set( )**.

USE IN SERIAL DEVICE DRIVERS

Each device that uses **tyLib** is described by a data structure of type **TY_DEV**. This structure begins with an I/O system device header so that it can be added directly to the I/O system's device list. A driver calls **tyDevInit( )** to initialize a **TY_DEV** structure for a specific device and then calls **iosDevAdd( )** to add the device to the I/O system.

The call to **tyDevInit( )** takes three parameters: the pointer to the **TY_DEV** structure to initialize, the desired size of the read and write ring buffers, and the address of a transmitter start-up routine. This routine will be called when characters are added for output and the transmitter is idle. Thereafter, the driver can call the following routines to perform the usual device functions:

tyRead( )
user read request to get characters that have been input

tyWrite( )
user write request to put characters to be output

tyIoctl( )
user I/O control request

tyIRd( )
interrupt-level routine to get an input character

tyITx( )
interrupt-level routine to deliver the next output character

Thus, **tyRead( )**, **tyWrite( )**, and **tyIoctl( )** are called from the driver's read, write, and I/O control functions. The routines **tyIRd( )** and **tyITx( )** are called from the driver's interrupt handler in response to receive and transmit interrupts, respectively.

Examples of using **tyLib** in a driver can be found in the source file(s) included by **tyCoDrv**. Source files are located in **src/drv/serial**.

TTY OPTIONS

A full range of options affects the behavior of *tty* devices. These options are selected by setting bits in the device option word using the **FIOSETOPTIONS** function in the **ioctl( )** routine (see *I/O Control Functions* below for more information). The following is a list of available options. The options are defined in the header file **ioLib.h**.

OPT_LINE
Selects line mode. A *tty* device operates in one of two modes: raw mode (unbuffered) or line mode. Raw mode is the default. In raw mode, each byte of input from the device is immediately available to readers, and the input is not modified except as directed by other options below. In line mode, input from the device is not available to readers until a NEWLINE character is received, and the input may be modified by backspace, line-delete, and end-of-file special characters.

OPT_ECHO
Causes all input characters to be echoed to the output of the same channel. This is done simply by putting incoming characters in the output ring as well as the input ring. If the output ring is full, the echoing is lost without affecting the input.

OPT_CRMOD
C language conventions use the NEWLINE character as the line terminator on both input and output. Most terminals, however, supply a RETURN character when the return key is hit, and require both a RETURN and a LINEFEED character to advance the output line. This option enables the appropriate translation: NEWLINEs are substituted for input RETURN characters, and NEWLINEs in the output file are automatically turned into a RETURN-LINEFEED sequence.

OPT_TANDEM
Causes the driver to generate and respond to the special flow control characters CTRL-Q and CTRL-S in what is commonly known as X-on/X-off protocol. Receipt of a CTRL-S input character will suspend output to that channel. Subsequent receipt of a CTRL-Q will resume the output. Also, when the VxWorks input buffer is almost full, a CTRL-S will be output to signal the other side to suspend transmission. When the input buffer is almost empty, a CTRL-Q will be output to signal the other side to resume transmission.

OPT_7_BIT
Strips the most significant bit from all bytes input from the device.

OPT_MON_TRAP
Enables the special monitor trap character, by default CTRL-X. When this character is received and this option is enabled, VxWorks will trap to the ROM resident monitor program. Note that this is quite drastic. All normal VxWorks functioning is suspended, and the computer system is entirely controlled by the monitor. Depending on the particular monitor, it may or may not be possible to restart VxWorks from the point of interruption. The default monitor trap character can be changed by calling **tyMonitorTrapSet( )**.

OPT_ABORT
Enables the special shell abort character, by default CTRL-C. When this character is received and this option is enabled, the VxWorks shell is restarted. This is useful for freeing a shell stuck in an unfriendly routine, such as one caught in an infinite loop or one that has taken an unavailable semaphore. For more information, see the *VxWorks Programmer's Guide: Shell.*

OPT_TERMINAL
This is not a separate option bit. It is the value of the option word with all the above bits set.

OPT_RAW
This is not a separate option bit. It is the value of the option word with none of the above bits set.

I/O CONTROL FUNCTIONS

The *tty* devices respond to the following **ioctl( )** functions. The functions are defined in the header **ioLib.h**.

FIOGETNAME
Gets the file name of the file descriptor and copies it to the buffer referenced to by *nameBuf*:

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

This function is common to all file descriptors for all devices.

FIOSETOPTIONS, **FIOOPTIONS**
Sets the device option word to the specified argument. For example, the call:

```
status = ioctl (fd, FIOOPTIONS, OPT_TERMINAL);
status = ioctl (fd, FIOSETOPTIONS, OPT_TERMINAL);
```

enables all the *tty* options described above, putting the device in a "normal" terminal mode. If the line protocol (**OPT_LINE**) is changed, the input buffer is flushed. The various options are described in **ioLib.h**.

FIOGETOPTIONS
Returns the current device option word:

```
options = ioctl (fd, FIOGETOPTIONS, 0);
```

FIONREAD
Copies to *nBytesUnread* the number of bytes available to be read in the device's input buffer:

```
status = ioctl (fd, FIONREAD, &nBytesUnread);
```

In line mode (**OPT_LINE** set), the **FIONREAD** function actually returns the number of characters available plus the number of lines in the buffer. Thus, if five lines of just NEWLINEs were in the input buffer, it would return the value 10 (5 characters + 5 lines).

FIONWRITE
Copies to *nBytes* the number of bytes queued to be output in the device's output buffer:

```
status = ioctl (fd, FIONWRITE, &nBytes);
```

FIOFLUSH
Discards all the bytes currently in both the input and the output buffers:

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

FIOWFLUSH
Discards all the bytes currently in the output buffer:

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

FIORFLUSH
Discards all the bytes currently in the input buffers:

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

FIOCANCEL
Cancels a read or write. A task blocked on a read or write may be released by a second task using this **ioctl( )** call. For example, a task doing a read can set a watchdog timer before attempting the read; the auxiliary task would wait on a semaphore. The watchdog routine can give the semaphore to the auxiliary task, which would then use the following call on the appropriate file descriptor:

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

FIOBAUDRATE
Sets the baud rate of the device to the specified argument. For example, the call:

```
status = ioctl (fd, FIOBAUDRATE, 9600);
```

Sets the device to operate at 9600 baud. This request has no meaning on a pseudo terminal.

FIOISATTY
Returns **TRUE** for a *tty* device:

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

FIOPROTOHOOK
Adds a protocol hook function to be called for each input character. *pfunction* is a pointer to the protocol hook routine which takes two arguments of type *int* and returns values of type **STATUS** (**TRUE** or **FALSE**). The first argument passed is set by the user via the **FIOPROTOARG** function. The second argument is the input character. If no further processing of the character is required by the calling routine (the input routine of the driver), the protocol hook routine *pFunction* should return **TRUE**. Otherwise, it should return **FALSE**:

```
status = ioctl (fd, FIOPROTOHOOK, pFunction);
```

FIOPROTOARG
Sets the first argument to be passed to the protocol hook routine set by **FIOPROTOHOOK** function:

```
status = ioctl (fd, FIOPROTOARG, arg);
```

FIORBUFSET
Changes the size of the receive-side buffer to *size*:

```
status = ioctl (fd, FIORBUFSET, size);
```

FIOWBUFSET
Changes the size of the send-side buffer to *size*:

```
status = ioctl (fd, FIOWBUFSET, size);
```

Any other **ioctl( )** request will return an error and set the status to **S_ioLib_UNKNOWN_REQUEST**.

INCLUDE FILES **tyLib.h**, **ioLib.h**

SEE ALSO **ioLib**, **iosLib**, **tyCoDrv**, *VxWorks Programmer's Guide: I/O System*

udpShow

NAME

udpShow – UDP information display routines

ROUTINES

udpShowInit( ) - initialize UDP show routines
udpstatShow( ) - display statistics for the UDP protocol

DESCRIPTION

This library provides routines to show UDP related statistics.

Interpreting these statistics requires detailed knowledge of Internet network protocols. Information on these protocols can be found in the following books:

TCP/IP Illustrated Volume II, The Implementation, by Richard Stevens

The Design and Implementation of the 4.4 BSD UNIX Operating System, by Leffler, McKusick, Karels and Quarterman

The **udpShowInit( )** routine links the UDP show facility into the VxWorks system. This is performed automatically if **INCLUDE_NET_SHOW** and **INCLUDE_UDP** are defined.

SEE ALSO

netLib, **netShow**

unixDrv

NAME

unixDrv – UNIX-file disk driver (VxSim for Solaris and VxSim for HP)

ROUTINES

unixDrv( ) - install UNIX disk driver
unixDiskDevCreate( ) - create a UNIX disk device
unixDiskInit( ) - initialize a dosFs disk on top of UNIX

DESCRIPTION

This driver emulates a VxWorks disk driver, but actually uses the UNIX file system to store the data. The VxWorks disk appears under UNIX as a single file. The UNIX file name, and the size of the disk, may be specified during the **unixDiskDevCreate( )** call.

USER-CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. The routine **unixDrv( )** must be called to initialize the driver and the **unixDiskDevCreate( )** routine is used to create devices.

CREATING UNIX DISKS

Before a UNIX disk can be used, it must be created. This is done with the

unixDiskDevCreate( ) call. The format of this call is:

```
BLK_DEV *unixDiskDevCreate
    (
    char    *unixFile,      /* name of the UNIX file to use         */
    int     bytesPerBlk,    /* number of bytes per block            */
    int     blksPerTrack,   /* number of blocks per track           */
    int     nBlocks         /* number of blocks on this device      */
    )
```

The UNIX file must be pre-allocated separately. This can be done using the UNIX mkfile(8) command. Note that you have to create an appropriately sized file. For example, to create a UNIX file system that is used as a common floppy dosFs file system, you would issue the command:

```
mkfile 1440k /tmp/floppy.dos
```

This will create space for a 1.44 Meg DOS floppy (1474560 bytes, or 2880 512-byte blocks).

The *bytesPerBlk* parameter specifies the size of each logical block on the disk. If *bytesPerBlk* is zero, 512 is the default.

The *blksPerTrack* parameter specifies the number of blocks on each logical track of the UNIX disk. If *blksPerTrack* is zero, the count of blocks per track will be set to *nBlocks* (*i.e.*, the disk will be defined as having only one track). UNIX disk devices typically are specified with only one track.

The *nBlocks* parameter specifies the size of the disk, in blocks. If *nBlocks* is zero the size of the UNIX file specified, divided by the number of bytes per block, is used.

The formatting parameters (*bytesPerBlk*, *blksPerTrack*, and *nBlocks*) are critical only if the UNIX disk already contains the contents of a disk created elsewhere. In that case, the formatting parameters must be identical to those used when the image was created. Otherwise, they may be any convenient number.

Once the device has been created it still does not have a name or file system associated with it. This must be done by using the file system's device initialization routine (*e.g.*, **dosFsDevInit( )**). The dosFs and rt11Fs file systems also provide make-file-system routines (**dosFsMkfs( )** and **rt11FsMkfs( )**), which may be used to associate a name and file system with the block device and initialize that file system on the device using default configuration parameters.

The **unixDiskDevCreate( )** call returns a pointer to a block device structure (**BLK_DEV**). This structure contains fields that describe the physical properties of a disk device and specify the addresses of routines within the UNIX disk driver. The **BLK_DEV** structure address must be passed to the desired file system (dosFs, rt11Fs, or rawFs) during the file system's device initialization or make-file-system routine. Only then is a name and file system associated with the device, making it available for use.

As an example, to create a 200KB disk, 512-byte blocks, and only one track, the proper call would be:

```
BLK_DEV *pBlkDev;
pBlkDev = unixDiskDevCreate ("/tmp/filesys1",  512,  400,  400,  0);
```

This will attach the UNIX file **/tmp/filesys1** as a block device.

A convenience routine, **unixDiskInit( )**, is provided to do the **unixDiskDevCreate( )** followed by either a **dosFsMkFs( )** or **dosFsDevInit( )**, whichever is appropriate.

The format of this call is:

```
BLK_DEV *unixDiskInit
    (
    char * unixFile,  /* name of the UNIX file to use */
    char * volName,   /* name of the dosFs volume to use */
    int    nBytes     /* number of bytes in dosFs volume */
    )
```

This call will create the UNIX disk if required.

IOCTL

Only the **FIODISKFORMAT** request is supported; all other ioctl requests return an error, and set the task's **errno** to **S_ioLib_UNKNOWN_REQUEST**.

SEE ALSO

dosFsDevInit( ), **dosFsMkfs( )**, **rt11FsDevInit( )**, **rt11FsMkfs( )**, **rawFsDevInit( )**, *VxWorks Programmer's Guide: I/O System, Local File Systems*

unldLib

NAME

unldLib – object module unloading library

ROUTINES

unld( ) - unload an object module by specifying a file name or module ID
unldByModuleId( ) - unload an object module by specifying a module ID
unldByNameAndPath( ) - unload an object module by specifying a name and path
unldByGroup( ) - unload an object module by specifying a group number
reld( ) - reload an object module

DESCRIPTION

This library provides a facility for unloading object modules. Once an object module has been loaded into the system (using the facilities provided by **loadLib**), it can be removed from the system by calling one of the **unld...( )** routines in this library.

Unloading of an object module does the following:

(1) It frees the space allocated for text, data, and BSS segments, unless **loadModuleAt( )** was called with specific addresses, in which case the user is responsible for freeing the space.

(2) It removes all symbols associated with the object module from the system symbol table.

(3) It removes the module descriptor from the module list.

Once the module is unloaded, any calls to routines in that module from other modules will fail unpredictably. The user is responsible for ensuring that no modules are unloaded that are used by other modules. **unld( )** checks the hooks created by the following routines to ensure none of the unloaded code is in use by a hook:

taskCreateHookAdd( )
taskDeleteHookAdd( )
taskHookAdd( )
taskSwapHookAdd( )
taskSwitchHookAdd( )

However, **unld( )** *does not* check the hooks created by these routines:

etherInputHookAdd( )
etherOutputHookAdd( )
excHookAdd( )
rebootHookAdd( )
moduleCreateHookAdd( )

The routines **unld( )** and **reld( )** are **shell commands**. That is, they are designed to be used only in the shell, and not in code running on the target. In future releases, calling **unld( )** and **reld( )** directly from code may not be supported.

INCLUDE FILES **unldLib.h**, **moduleLib.h**

SEE ALSO **loadLib**, **moduleLib**, *Tornado User's Guide: Cross-Development*

usrAta

NAME **usrAta** – ATA/ATAPI initialization

ROUTINES **usrAtaConfig( )** - mount a DOS file system from an ATA hard disk or a CDROM
usrAtaInit( ) - initialize the hard disk driver

usrConfig

NAME **usrConfig** – user-defined system configuration library

ROUTINES **usrInit( )** - user-defined system initialization routine
usrRoot( ) - the root task
usrClock( ) - user-defined system clock interrupt routine

DESCRIPTION This library is the WRS-supplied configuration module for VxWorks. It contains the root task, the primary system initialization routine, the network initialization routine, and the clock interrupt routine.

The include file **config.h** includes a number of system-dependent parameters used in this file.

In an effort to simplify the presentation of the configuration of VxWorks, this file has been split into smaller files. These additional configuration source files are located in **../../src/config/usr***xxx***.c** and are **#included** into this file below. This file contains the bulk of the code a customer is likely to customize.

The module **usrDepend.c** contains checks that guard against unsupported configurations such as **INCLUDE_NFS** without **INCLUDE_RPC**. The module **usrKernel.c** contains the core initialization of the kernel which is rarely customized, but provided for information. The module **usrNetwork.c** now contains all network initialization code. Finally, the module **usrExtra.c** contains the conditional inclusion of the optional packages selected in **configAll.h**.

The source code necessary for the configuration selected is entirely included in this file during compilation as part of a standard build in the board support package. No other make is necessary.

INCLUDE FILES **config.h**

SEE ALSO *Tornado User's Guide: Getting Started, Cross-Development*

usrFd

NAME **usrFd** – floppy disk initialization

ROUTINES **usrFdConfig( )** - mount a DOS file system from a floppy disk

usrFdiskPartLib

NAME

usrFdiskPartLib – FDISK-style partition handler

ROUTINES

usrFdiskPartRead( ) - read an FDISK-style partition table
usrFdiskPartCreate( ) - create an FDISK-like partition table on a disk
usrFdiskPartShow( ) - parse and display partition data

DESCRIPTION

This module is provided is source code to accommodate various customizations of partition table handling, resulting from variations in the partition table format in a particular configuration. It is intended for use with **dpartCbio** partition manager.

This code supports both mounting MSDOS file systems and displaying partition tables written by MSDOS **FDISK.exe** or by any other MSDOS **FDISK.exe** compatible partitioning software.

The first partition table is contained within a hard drives Master Boot Record (MBR) sector, which is defined as sector one, cylinder zero, head zero or logical block address zero.

The mounting and displaying routines within this code will first parse the MBR partition tables entries (defined below) and also recursively parse any "extended" partition tables, which may reside within another sector further into the hard disk. MSDOS file systems within extended partitions are known to those familiar with the MSDOS **FDISK.exe** utility as "Logical drives within the extended partition".

Here is a picture showing the layout of a single disk containing multiple MSDOS file systems:

```
+---------------------------------------------------------+
|<---------------------The entire disk------------------->|
|M                                                        |
|B<---C:--->                                              |
|R           /---- First extended partition--------------\|
|           E<---D:---><-Rest of the ext part------------>|
|P          x                                             |
|A          t          E<---E:--->E<Rest of the ext part->|
|R                     x          x                       |
|T                     t          t<---------F:---------->|
+---------------------------------------------------------+
(Ext == extended partition sector)
C: is a primary partiion
D:, E:, and F: are logical drives within the extended partition.
```

A MS-DOS partition table resides within one sector on a hard disk. There is always one in the first sector of a hard disk partitioned with FDISK.exe. There first partition table may contain references to "extended" partition tables residing on other sectors if there are multiple partitions. The first sector of the disk is the starting point. Partition tables are of the format:

```
Offset from
the beginning
of the sector          Description
-------------          -------------------------
   0x1be               Partition 1 table entry  (16 bytes)
   0x1ce               Partition 2 table entry  (16 bytes)
   0x1de               Partition 3 table entry  (16 bytes)
   0x1ee               Partition 4 table entry  (16 bytes)
   0x1fe               Signature  (0x55aa, 2 bytes)
```

Individual MSDOS partition table entries are of the format:

```
Offset   Size      Description
------   ----      ------------------------------
 0x0     8 bits    boot type
 0x1     8 bits    beginning sector head value
 0x2     8 bits    beginning sector (2 high bits of cylinder#)
 0x3     8 bits    beginning cylinder# (low order bits of cylinder#)
 0x4     8 bits    system indicator
 0x5     8 bits    ending sector head value
 0x6     8 bits    ending sector (2 high bits of cylinder#)
 0x7     8 bits    ending cylinder# (low order bits of cylinder#)
 0x8    32 bits    number of sectors preceding the partition
 0xc    32 bits    number of sectors in the partition
```

The Cylinder, Head and Sector values herein are not used, instead the 32-bit partition offset and size (also known as LBA addresses) are used exclusively to determine partition geometry.

If a non-partitioned disk is detected, in which case the 0'th block is a DosFs boot block rather then an MBR, the entire disk will be configured as partition 0, so that disks formatted with VxWorks and disks formatted on MS-DOS or Windows can be accepted interchangeably.

The **usrFdiskPartCreate( )** will create a partition table with up to four partitions, which can be later used with **usrFdiskPartRead( )** and **dpartCbio** to manage a partitioned disk on VxWorks.

However, it can not be guaranteed that this partition table can be used on another system due to several BIOS specific parameters in the boot area. If interchangeability via removable disks is a requirement, partition tables should be created and volumes should be formatted on the other system with which the data is to be interchanged.

WARNING: The partition decode function is recursive, up to the maximum number of partitions expected, which is no more then 24.

Sufficient stack space needs to be provided via **taskSpawn( )** to accommodate the recursion level.

SEE ALSO **dpartCbio**

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
diskFormat( ) - format a disk
diskInit( ) - initialize a file system on a block device
ioHelp( ) - print a synopsis of I/O utility functions

U

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

USAGE FROM TORNADO

Some of the functions in this library have counterparts of the same names built into the Tornado 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.*: ce > @ls "/sd0" ce will list the directory of a disk named **/sd0** on the target, while

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

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

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

```
-> pwd
```

will display the Tornado 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.

SEE ALSO

ioLib, **dosFsLib**, **netDrv**, **nfsLib**, *VxWorks Programmer's Guide: Target Shell VxWorks Programmer's Guide: Tornado Users's Guide*

usrIde

NAME

usrIde – IDE initialization

ROUTINES

usrIdeConfig( ) - mount a DOS file system from an IDE hard disk

usrLib

NAME

usrLib – user interface subroutine library

ROUTINES

help( ) - print a synopsis of selected routines
netHelp( ) - print a synopsis of network routines
bootChange( ) - change the boot line
periodRun( ) - call a function periodically
period( ) - spawn a task to call a function periodically
repeatRun( ) - call a function repeatedly
repeat( ) - spawn a task to call a function repeatedly
sp( ) - spawn a task with default parameters
checkStack( ) - print a summary of each task's stack usage
i( ) - print a summary of each task's TCB
ti( ) - print complete information from a task's TCB
show( ) - print information on a specified object
ts( ) - suspend a task
tr( ) - resume a task
td( ) - delete a task
version( ) - print VxWorks version information
m( ) - modify memory
d( ) - display memory
ld( ) - load an object module into memory
devs( ) - list all system-known devices
lkup( ) - list symbols
lkAddr( ) - list symbols whose values are near a specified value
mRegs( ) - modify registers
pc( ) - return the contents of the program counter
printErrno( ) - print the definition of a specified error status value
printLogo( ) - print the VxWorks logo
logout( ) - log out of the VxWorks system
h( ) - display or set the size of shell history
spyReport( ) - display task activity data
spyTask( ) - run periodic task activity reports
spy( ) - begin periodic task activity reports
spyClkStart( ) - start collecting task activity data
spyClkStop( ) - stop collecting task activity data
spyStop( ) - stop spying and reporting
spyHelp( ) - display task monitoring help menu

DESCRIPTION

This library consists of routines meant to be executed from the VxWorks shell. It provides useful utilities for task monitoring and execution, system information, symbol table management, *etc*.

Many of the routines here are simply command-oriented interfaces to more general routines contained elsewhere in VxWorks. Users should feel free to modify or extend this library, and may find it preferable to customize capabilities by creating a new private library, using this one as a model, and appropriately linking the new one into the system.

Some routines here have optional parameters. If those parameters are zero, which is what the shell supplies if no argument is typed, default values are typically assumed.

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

INCLUDE FILES **usrLib.h**

SEE ALSO **usrFsLib**, **tarLib**, **spyLib**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

usrScsi

NAME **usrScsi** – SCSI initialization

ROUTINES **usrScsiConfig( )** - configure SCSI peripherals

vmBaseLib

NAME **vmBaseLib** – base virtual memory support library

ROUTINES **vmBaseLibInit( )** - initialize base virtual memory support
vmBaseGlobalMapInit( ) - initialize global mapping
vmBaseStateSet( ) - change the state of a block of virtual memory
vmBasePageSizeGet( ) - return the page size

DESCRIPTION This library provides the minimal MMU (Memory Management Unit) support needed in a system. Its primary purpose is to create cache-safe buffers for **cacheLib**. Buffers are provided to optimize I/O throughput.

A call to **vmBaseLibInit( )** initializes this library, thus permitting **vmBaseGlobalMapInit( )** to initialize the MMU and set up MMU translation tables. Additionally, **vmBaseStateSet( )** can be called to change the translation tables dynamically.

This library is a release-bundled complement to **vmLib** and **vmShow**, modules that offer full-featured MMU support and virtual memory information display routines. The **vmLib** and **vmShow** libraries are distributed as the unbundled virtual memory support option, VxVMI.

CONFIGURATION Bundled MMU support is included in VxWorks when the configuration macro **INCLUDE_MMU_BASIC** is defined. If the configuration macro **INCLUDE_MMU_FULL** is also defined, the default is full MMU support (unbundled).

INCLUDE FILES **sysLib.h**, **vmLib.h**

SEE ALSO **vmLib**, **vmShow**, *VxWorks Programmer's Guide: Virtual Memory*

vmLib

NAME **vmLib** – architecture-independent virtual memory support library (VxVMI Opt.)

ROUTINES **vmLibInit( )** - initialize the virtual memory support module (VxVMI Opt.)
vmGlobalMapInit( ) - initialize global mapping (VxVMI Opt.)
vmContextCreate( ) - create a new virtual memory context (VxVMI Opt.)
vmContextDelete( ) - delete a virtual memory context (VxVMI Opt.)
vmStateSet( ) - change the state of a block of virtual memory (VxVMI Opt.)
vmStateGet( ) - get the state of a page of virtual memory (VxVMI Opt.)

vmMap( ) - map physical space into virtual space (VxVMI Opt.)
vmGlobalMap( ) - map physical pages to virtual space in shared global virtual memory (VxVMI Opt.)
vmGlobalInfoGet( ) - get global virtual memory information (VxVMI Opt.)
vmPageBlockSizeGet( ) - get the architecture-dependent page block size (VxVMI Opt.)
vmTranslate( ) - translate a virtual address to a physical address (VxVMI Opt.)
vmPageSizeGet( ) - return the page size (VxVMI Opt.)
vmCurrentGet( ) - get the current virtual memory context (VxVMI Opt.)
vmCurrentSet( ) - set the current virtual memory context (VxVMI Opt.)
vmEnable( ) - enable or disable virtual memory (VxVMI Opt.)
vmTextProtect( ) - write-protect a text segment (VxVMI Opt.)

DESCRIPTION

This library provides an architecture-independent interface to the CPU's memory management unit (MMU). Although **vmLib** is implemented with architecture-specific libraries, application code need never reference directly the architecture-dependent code in these libraries.

A fundamental goal in the design of **vmLib** was to permit transparent backward compatibility with previous versions of VxWorks that did not use the MMU. System designers may opt to disable the MMU because of timing constraints, and some architectures do not support MMUs; therefore VxWorks functionality must not be dependent on the MMU. The resulting design permits a transparent configuration with no change in the programming environment (but the addition of several protection features, such as text segment protection) and the ability to disable virtual memory in systems that require it.

The **vmLib** library provides a mechanism for creating virtual memory contexts, **vmContextCreate( )**. These contexts are not automatically created for individual tasks, but may be created dynamically by tasks, and swapped in and out in an application specific manner.

All virtual memory contexts share a global transparent mapping of virtual to physical memory for all of local memory and the local hardware device space (defined in **sysLib.c** for each board port in the **sysPhysMemDesc** data structure). When the system is initialized, all of local physical memory is accessible at the same address in virtual memory (this is done with calls to **vmGlobalMap( )**.) Modifications made to this global mapping in one virtual memory context appear in all virtual memory contexts. For example, if the exception vector table (which resides at address 0 in physical memory) is made read only by calling **vmStateSet( )** on virtual address 0, the vector table will be read only in all virtual memory contexts.

Private virtual memory can also be created. When physical pages are mapped to virtual memory that is not in the global transparent region, this memory becomes accessible only in the context in which it was mapped. (The physical pages will also be accessible in the transparent translation at the physical address, unless the virtual pages in the global transparent translation region are explicitly invalidated.) State changes (writability, validity, *etc*.) to a section of private virtual memory in a virtual memory context do not appear in other contexts. To facilitate the allocation of regions of virtual space, **vmGlobalInfoGet( )** returns a pointer to an array of booleans describing which portions of the virtual address space are devoted to global memory. Each successive array element corresponds to contiguous regions of virtual memory the size of which is architecture-dependent and which may be obtained with a call to **vmPageBlockSizeGet( )**. If the boolean array element is true, the corresponding region of virtual memory, a "page block", is reserved for global virtual memory and should not be used for private virtual memory. (If **vmMap( )** is called to map virtual memory previously defined as global, the routine will return an error.)

All the state information for a block of virtual memory can be set in a single call to **vmStateSet( )**. It performs parameter checking and checks the validity of the specified virtual memory context. It may also be used to set architecture-dependent state information. See **vmLib.h** for additional architecture-dependent state information.

The routine **vmContextShow( )** in **vmShow** displays the virtual memory context for a specified context. For more information, see the manual entry for this routine.

CONFIGURATION

Full MMU support (**vmLib**, and optionally, **vmShow**) is included in VxWorks when the configuration macro **INCLUDE_MMU_FULL** is defined. If the configuration macro **INCLUDE_MMU_BASIC** is also defined, the default is full MMU support (unbundled).

The **sysLib.c** library contains a data structure called **sysPhysMemDesc**, which is an array of **PHYS_MEM_DESC** structures. Each element of the array describes a contiguous section of physical memory. The description of this memory includes its physical address, the virtual address where it should be mapped (typically, this is the same as the physical address, but not necessarily so), an initial state for the memory, and a mask defining which state bits in the state value are to be set. Default configurations are defined for each board support package (BSP), but these mappings may be changed to suit user-specific system configurations. For example, the user may need to map additional VME space where the backplane network interface data structures appear.

AVAILABILITY

This library and **vmShow** are distributed as the unbundled virtual memory support option, VxVMI. A scaled down version, **vmBaseLib**, is provided with VxWorks for systems that do not permit optional use of the MMU, or for architectures that require certain features of the MMU to perform optimally (in particular, architectures that rely heavily on caching, but do not support bus snooping, and thus require the ability to mark inter-processor communications buffers as non-cacheable.) Most routines in **vmBaseLib** are referenced internally by VxWorks; they are not callable by application code.

INCLUDE FILES

vmLib.h

SEE ALSO

sysLib, **vmShow**, *VxWorks Programmer's Guide: Virtual Memory*

vmShow

NAME **vmShow** – virtual memory show routines (VxVMI Opt.)

ROUTINES
vmShowInit( ) - include virtual memory show facility (VxVMI Opt.)
vmContextShow( ) - display the translation table for a context (VxVMI Opt.)

DESCRIPTION This library contains virtual memory information display routines.

The routine **vmShowInit( )** links this facility into the VxWorks system. It is called automatically when this facility is configured into VxWorks using either of the following methods:

If you use the configuration header files, define both **INCLUDE_MMU_FULL** and **INCLUDE_SHOW_ROUTINES** in **config.h**.

If you use the Tornado project facility, select **INCLUDE_MMU_FULL_SHOW**.

AVAILABILITY This module and **vmLib** are distributed as the unbundled virtual memory support option, VxVMI.

INCLUDE FILES **vmLib.h**

SEE ALSO **vmLib**, *VxWorks Programmer's Guide: Virtual Memory*

vxLib

NAME **vxLib** – miscellaneous support routines

ROUTINES
vxTas( ) - C-callable atomic test-and-set primitive
vxMemArchProbe( ) - architecture specific part of **vxMemProbe( )**
vxMemProbe( ) - probe an address for a bus error
vxSSEnable( ) - enable the superscalar dispatch (MC68060)
vxSSDisable( ) - disable the superscalar dispatch (MC68060)
vxPowerModeSet( ) - set the power management mode (PowerPC, SH, x86)
vxPowerModeGet( ) - get the power management mode (PowerPC, SH, x86)
vxPowerDown( ) - place the processor in reduced-power mode (PowerPC, SH)
vxCr0Get( ) - get a content of the Control Register 0 (x86)
vxCr0Set( ) - set a value to the Control Register 0 (x86)
vxCr2Get( ) - get a content of the Control Register 2 (x86)
vxCr2Set( ) - set a value to the Control Register 2 (x86)
vxCr3Get( ) - get a content of the Control Register 3 (x86)

vxCr3Set( ) - set a value to the Control Register 3 (x86)
vxCr4Get( ) - get a content of the Control Register 4 (x86)
vxCr4Set( ) - set a value to the Control Register 4 (x86)
vxEflagsGet( ) - get a content of the EFLAGS register (x86)
vxEflagsSet( ) - set a value to the EFLAGS register (x86)
vxDrGet( ) - get a content of the Debug Register 0 to 7 (x86)
vxDrSet( ) - set a value to the Debug Register 0 to 7 (x86)
vxTssGet( ) - get a content of the TASK register (x86)
vxTssSet( ) - set a value to the TASK register (x86)
vxGdtrGet( ) - get a content of the Global Descriptor Table Register (x86)
vxIdtrGet( ) - get a content of the Interrupt Descriptor Table Register (x86)
vxLdtrGet( ) - get a content of the Local Descriptor Table Register (x86)

DESCRIPTION This module contains miscellaneous VxWorks support routines.

INCLUDE FILES **vxLib.h**

wdbLib

NAME **wdbLib** – WDB agent context management library

ROUTINES **wdbSystemSuspend( )** - suspend the system.

DESCRIPTION This library provides a routine to transfer control from the run time system to the WDB agent running in external mode. This agent in external mode allows a system-wide control, including ISR debugging, from a host tool (*e.g.*: **Crosswind**, **WindSh** ...) through the target server and the WDB communication link.

INCLUDE FILES **wdb/wdbLib.h**

SEE ALSO *API Guide: WTX Protocol*, *Tornado User's Guide: Overview*

wdbUserEvtLib

NAME **wdbUserEvtLib** – WDB user event library

ROUTINES **wdbUserEvtLibInit( )** - include the WDB user event library
wdbUserEvtPost( ) - post a user event string to host tools.

DESCRIPTION This library contains routines for sending WDB User Events. The event is sent through the WDB agent, the WDB communication link and the target server to the host tools that have registered for it. The event received by host tools will be a WTX user event string.

INCLUDE FILES **wdb/wdbLib.h**

SEE ALSO *API Guide: WTX Protocol*

wdLib

NAME

wdLib – watchdog timer library

ROUTINES

wdCreate( ) - create a watchdog timer
wdDelete( ) - delete a watchdog timer
wdStart( ) - start a watchdog timer
wdCancel( ) - cancel a currently counting watchdog

DESCRIPTION

This library provides a general watchdog timer facility. Any task may create a watchdog timer and use it to run a specified routine in the context of the system-clock ISR, after a specified delay.

Once a timer has been created with **wdCreate( )**, it can be started with **wdStart( )**. The **wdStart( )** routine specifies what routine to run, a parameter for that routine, and the amount of time (in ticks) before the routine is to be called. (The timeout value is in ticks as determined by the system clock; see **sysClkRateSet( )** for more information.) After the specified delay ticks have elapsed (unless **wdCancel( )** is called first to cancel the timer) the timeout routine is invoked with the parameter specified in the **wdStart( )** call. The timeout routine is invoked whether the task which started the watchdog is running, suspended, or deleted.

The timeout routine executes only once per **wdStart( )** invocation; there is no need to cancel a timer with **wdCancel( )** after it has expired, or in the expiration callback itself.

Note that the timeout routine is invoked at interrupt level, rather than in the context of the task. Thus, there are restrictions on what the routine may do. Watchdog routines are constrained to the same rules as interrupt service routines. For example, they may not take semaphores, issue other calls that may block, or use I/O system routines like **printf( )**.

EXAMPLE

In the fragment below, if **maybeSlowRoutine( )** takes more than 60 ticks, **logMsg( )** will be called with the string as a parameter, causing the message to be printed on the console. Normally, of course, more significant corrective action would be taken.

```
WDOG_ID wid = wdCreate ();
wdStart (wid, 60, logMsg, "Help, I've timed out!");
maybeSlowRoutine ();        /* user-supplied routine */
wdCancel (wid);
```

INCLUDE FILES

wdLib.h

SEE ALSO

logLib, *VxWorks Programmer's Guide: Basic OS*

wdShow

NAME **wdShow** – watchdog show routines

ROUTINES **wdShowInit( )** - initialize the watchdog show facility
wdShow( ) - show information about a watchdog

DESCRIPTION This library provides routines to show watchdog statistics, such as watchdog activity, a watchdog routine, *etc*.

The routine **wdShowInit( )** links the watchdog show facility into the VxWorks system. It is called automatically when this show facility is configured into VxWorks using either of the following methods:

– If you use the configuration header files, define **INCLUDE_SHOW_ROUTINES** in **config.h**.

– If you use the Tornado project facility, select **INCLUDE_WATCHDOGS_SHOW**.

INCLUDE FILES **wdLib.h**

SEE ALSO **wdLib**, *VxWorks Programmer's Guide: Basic OS, Target Shell*, **windsh**, *Tornado User's Guide: Shell*

wvFileUploadPathLib

NAME **wvFileUploadPathLib** – file destination for event data

ROUTINES **fileUploadPathLibInit( )** - initialize the **wvFileUploadPathLib** library (Windview)
fileUploadPathCreate( ) - create a file for depositing event data (Windview)
fileUploadPathClose( ) - close the event-destination file (WindView)
fileUploadPathWrite( ) - write to the event-destination file (WindView)

DESCRIPTION This file contains routines that write events to a file rather than uploading them to the host using a type of socket connection. If the file indicated is a TSFS file, this routine has the same result as uploading to a host file using other methods, allowing it to replace **evtRecv( )**. The file can be created anywhere, however, and event data can be kept on the target if desired.

SEE ALSO **wvSockUploadPathLib**, **wvTsfsUploadPathLib**

wvLib

NAME **wvLib** – event logging control library (WindView)

ROUTINES
wvLibInit( ) - initialize **wvLib** - first step (WindView)
wvLibInit2( ) - initialize **wvLib** - final step (WindView)
wvEvtLogInit( ) - initialize an event log (WindView)
wvEvtLogStart( ) - start logging events to the buffer (WindView)
wvEvtLogStop( ) - stop logging events to the buffer (WindView)
wvEvtClassSet( ) - set the class of events to log (WindView)
wvEvtClassGet( ) - get the current set of classes being logged (WindView)
wvEvtClassClear( ) - clear the specified class of events from those being logged (WindView)
wvEvtClassClearAll( ) - clear all classes of events from those logged (WindView)
wvObjInstModeSet( ) - set object instrumentation on/off (WindView)
wvObjInst( ) - instrument objects (WindView)
wvSigInst( ) - instrument signals (WindView)
wvEventInst( ) - instrument VxWorks Events (WindView)
wvEvent( ) - log a user-defined event (WindView)
wvUploadStart( ) - start upload of events to the host (WindView)
wvUploadStop( ) - stop upload of events to host (WindView)
wvUploadTaskConfig( ) - set priority and stack size of **tWVUpload** task (WindView)
wvLogHeaderCreate( ) - create the event-log header (WindView)
wvLogHeaderUpload( ) - transfer the log header to the host (WindView)
wvEvtBufferGet( ) - return the ID of the WindView event buffer (WindView)
wvTaskNamesPreserve( ) - preserve an extra copy of task name events (WindView)
wvTaskNamesUpload( ) - upload preserved task name events (WindView)

DESCRIPTION This library contains routines that control event collection and upload of event data from the target to various destinations. The routines define the interface for the target component of WindView. When event data has been collected, the routines in this library are used to produce event logs that can be understood by the WindView host tools.

An event log is made up of a header, followed by the task names of each task present in the system when the log is started, followed by a string of events produced by the various event points throughout the kernel and associated libraries. In general, this information is gathered and stored temporarily on the target, and later uploaded to the host in the proper order to form an event log. The routines in this file can be used to create logs in various ways, depending on which routines are called, and in which order the routines are called.

There are three methods for uploading event logs. The first is to defer upload of event data until after logging has been stopped in order to eliminate events associated with upload activity from the event log. The second is to continuously upload event data as it is gathered. This allows the collection of very large event logs, that may contain more events than the target event buffer can store at one time. The third is to defer upload of the data until after a target reboot. This method allows event data to continuously overwrite earlier data in the event buffer, creating a log of the events leading to a target failure (a post-mortem event log).

Each of these three methods is explained in more detail in *CREATING AN EVENT LOG*.

EVENT BUFFERS AND UPLOAD PATHS

Many of the routines in **wvLib** require access to the buffer used to store event data (the event buffer) and to the communication paths from the target to the host (the upload paths). Both the buffer and the path are referenced with IDs that provide **wvLib** with the appropriate information for access.

The event buffering mechanism used by **wvLib** is provided by **rBuffLib**. The upload paths available for use with **wvLib** are provided by **wvFileUploadPathLib**, **wvTsfsUploadPathLib** and **wvSockUploadPathLib**.

The upload mechanism backs off and retries writing to the upload path if an error occurs during the write attempt with the **errno EAGAIN** or **EWOULDBLOCK**. Two global variables are used to set the amount of time to back off and the number of retries. The variables are:

```
int wvUploadMaxAttempts   /* number of attempts to try writing */
int wvUploadRetryBackoff  /* delay between tries (in ticks - 60/sec) */
```

INITIALIZATION

This library is initialized in two steps. The first step, done by calling **wvLibInit( )**, associates event logging routines to system objects. This is done when the kernel is initialized. The second step, done by calling **wvLibInit2( )**, associates all other event logging routines with the appropriate event points. Initialization is done automatically when **INCLUDE_WINDVIEW** is defined.

Before event logging can be started, and each time a new event buffer is used to store logged events, **wvEvtLogInit( )** must be called to bind the event logging routines to a specific buffer.

DETERMINING WHICH EVENTS ARE COLLECTED

There are three classes of events that can be collected. They are:

```
WV_CLASS_1              /* Events causing context switches */
WV_CLASS_2              /* Events causing task-state transitions */
WV_CLASS_3              /* Events from object and system libraries */
```

The second class includes all of the events contained within the first class, plus additional events causing task-state transitions but not causing context switches. The third class contains all of the second, and allows logging of events within system libraries. It can also be limited to specific objects or groups of objects:

– Using **wvObjInst( )** allows individual objects (for example, **sem1**) to be instrumented.

– Using **wvSigInst( )** allows signals to be instrumented.

– Using **wvObjInstModeSet( )** allows finer control over what type of objects are instrumented. **wvObjInstModeSet( )** allows types of system objects (for example, semaphores, watchdogs) to be instrumented as they are created.

Logging events in Class 3 generates the most data, which may be helpful during analysis of the log. It is also the most intrusive on the system, and may affect timing and performance. Class 2 is more intrusive than Class 1. In general, it is best to use the lowest class that still provides the required level of detail.

To manipulate the class of events being logged, the following routines can be used: **wvEvtClassSet( )**, **wvEvtClassGet( )**, **wvEvtClassClear( )**, and **wvEvtClassClearAll( )**. To log a user-defined event, **wvEvent( )** can be used. It is also possible to log an event from any point during execution using **e( )**, located in **dbgLib**.

CONTROLLING EVENT LOGGING

Once the class of events has been specified, event logging can be started with **wvEvtLogStart( )** and stopped with **wvEvtLogStop( )**.

CREATING AN EVENT LOG

An event log consists of a header, a section of task names, and a list of events logged after calling **wvEvtLogStart( )**. As discussed above, there are three common ways to upload an event log.

Deferred Upload

When creating an event log by uploading the event data after event logging has been stopped (deferred upload), the following series of calls can be used to start and stop the collection. In this example the memory allocated to store the log header is in the system partition. The event buffer should be allocated from the system memory partition as well. Error checking has been eliminated to simplify the example.

```
/* wvLib and rBuffLib initialized at system start up */
#include "vxWorks.h"
#include "wvLib.h"
#include "private/wvBufferP.h"
#include "private/wvUploadPathP.h"
#include "private/wvFileUploadPathLibP.h"
BUFFER_ID           bufId;
UPLOAD_ID           pathId;
WV_UPLOAD_TASK_ID   upTaskId;
WV_LOG_HEADER_ID    hdrId;
/*
 * To prepare the event log and start logging:
 */
/* Create event buffer in memSysPart, yielding bufId. */
wvEvtLogInit (bufId);
hdrId = wvLogHeaderCreate (memSysPartId);
wvEvtClassSet (WV_CLASS_1);         /* set to log class 1 events */
```

W

```
wvEvtLogStart ();
/*
 * To stop logging and complete the event log.
 */
wvEvtLogStop ();
/* Create an uplaod path using wvFileUploadPathLib, yielding pathId. */
wvLogHeaderUpload (hdrId, pathId);
upTaskId = wvUploadStart (bufId, pathId, TRUE);
wvUploadStop (upTaskId);
/* Close the upload path and destroy the event buffer */
```

Routines which can be used as they are, or modified to meet the users needs, are located in **usrWindview.c**. These routines, **wvOn( )** and **wvOff( )**, provide a way to produce useful event logs without using the host user interface of WindView.

Continuous Upload

When uploading event data as it is still being logged to the event buffer (continuous upload), simply rearrange the above calls:

```
/* Includes and declarations. */
/*
 * To prepare the event log and start logging:
 */
/* Create event buffer in memSysPart, yielding bufId. */
/* Create an uplaod path, yielding pathId. */
wvEvtLogInit (bufId);
upTaskId = wvUploadStart (bufId, pathId, TRUE);
hdrId = wvLogHeaderCreate (memSysPartId);
wvLogHeaderUpload (hdrId, pathId);
wvEvtClassSet (WV_CLASS_1);         /* set to log class 1 events */
wvEvtLogStart ();
/*
 * To stop logging and complete the event log:
 */
wvEvtLogStop ();
wvUploadStop (upTaskId);
/* Close the upload path and destroy the event buffer */
```

Post-Mortem Event Collection

This library also contains routines that preserve task name information throughout event logging in order to produce post-mortem event logs: **wvTaskNamesPreserve( )** and **wvTaskNamesUpload( )**.

Post-mortem event logs typically contain events leading up to a target failure. The memory containing the information to be stored in the log must not be zeroed when the system reboots. The event buffer is set up to allow event data to be logged to it continuously, overwriting the data collected earlier. When event logging is stopped, either by a system failure or at the request of the user, the event buffer may not contain the first events logged due to the overwriting. As tasks are created the **EVENT_TASKNAME** that is used by the WindView host tools to associate a task ID with a task name can be overwritten, while other events pertaining to that task ID may still be present in the event buffer. In order to assure that the WindView host tools can assign a task name to a context, a copy of all task name events can be preserved outside the event buffer and uploaded separately from the event buffer.

Note that several of the routines in **wvLib**, including **wvTaskNamesPreserve( )**, take a memory partition ID as an argument. This allows memory to be allocated from a user-specified partition. For post-mortem data collection, the memory partition should be within memory that is not zeroed upon system reboot. The event buffer, preserved task names, and log header should be stored in this partition.

Generating a post-mortem event log is similar to generating a deferred upload log. Typically event logging is stopped due to a system failure, but it may be stopped in any way. To retrieve the log header, task name buffer, and event buffer after a target reboot, these IDs must be remembered or stored along with the collected information in the non-zeroed memory. Also, the event buffer should be set to allow continuous logging by overwriting earlier event data. The following produces a post-mortem log. The non-zeroed memory partition has the ID *postMortemPartId*.

```
/* Includes, as in the examples above. */
BUFFER_ID           bufId;
UPLOAD_ID           pathId;
WV_UPLOAD_TASK_ID   upTaskId;
WV_LOG_HEADER_ID    hdrId;
WV_TASKBUF_ID       taskBufId;
/*
 * To prepare the event log and start logging:
 */
/*
 * Create event buffer in non-zeroed memory, allowing overwrite,
 @ yielding bufId.
 */
wvEvtLogInit (bufId);
taskBufId = wvTaskNamesPreserve (postMortemPartId, 32);
hdrId = wvLogHeaderCreate (postMortemPartId);
wvEvtClassSet (WV_CLASS_1);         /* set to log class 1 events */
wvEvtLogStart ();
/*
 * System fails and reboots.  Note that taskBufId, bufId and
 @ hdrId must be preserved through the reboot so they can be
 @ used to upload the data.
 */
/* Create an uplaod path, yielding pathId. */
wvLogHeaderUpload (hdrId, pathId);
```

```
upTaskId = wvUploadStart (bufId, pathId, TRUE);
wvUploadStop (upTaskId);
wvTaskNamesUpload (taskBufId, pathId);
/* Close the upload path and destroy the event buffer */
```

INCLUDE FILES **wvLib.h**, **eventP.h**

SEE ALSO **rBuffLib**, **wvFileUploadPathLib**, **wvSockUploadPathLib**, **wvTsfsUploadPathLib**, *WindView User's Guide*

wvNetLib

NAME **wvNetLib** – WindView for Networking Interface Library

ROUTINES

wvNetEnable( ) - begin reporting network events to WindView
wvNetDisable( ) - end reporting of network events to WindView
wvNetLevelAdd( ) - enable network events with specific priority level
wvNetLevelRemove( ) - disable network events with specific priority level
wvNetEventEnable( ) - activate specific network events
wvNetEventDisable( ) - deactivate specific network events
wvNetAddressFilterSet( ) - specify an address filter for events
wvNetAddressFilterClear( ) - remove the address filter for events
wvNetPortFilterSet( ) - specify an address filter for events
wvNetPortFilterClear( ) - remove the port number filter for events

DESCRIPTION This library provides the user interface to the network-related events for the WindView system visualization tool. These events are divided into two WindView classes. The **NET_CORE_EVENT** class indicates events directly related to data transfer. All other events (such as memory allocation and API routines) use the **NET_AUX_EVENT** class. Within each class, events are assigned one of eight priority levels. The four highest priority levels (**EMERGENCY**, **ALERT**, **CRITICAL**, and **ERROR**) indicate the occurrence of errors and the remaining four (**WARNING**, **NOTICE**, **INFO**, and **VERBOSE**) provide progressively more detailed information about the internal processing in the network stack.

USER INTERFACE If WindView support is included, the **wvNetStart( )** and **wvNetStop( )** routines will enable and disable event reporting for the network stack. The start routine takes a single parameter specifying the minimum priority level for all network components. That setting may be modified with the **wvNetLevelAdd( )** and **wvNetLevelRemove( )** routines. Individual events may be included or removed with the **wvNetEventEnable( )** and **wvNetDisable( )** routines.

The **wvNetAddressFilterSet( )** and **wvNetPortFilterSet( )** routines provide further screening for some events.

SEE ALSO *WindView for Tornado User's Guide*

wvSockUploadPathLib

NAME
wvSockUploadPathLib – socket upload path library

ROUTINES
sockUploadPathLibInit( ) - initialize **wvSockUploadPathLib** library (Windview)
sockUploadPathCreate( ) - establish an upload path to the host using a socket (Windview)
sockUploadPathClose( ) - close the socket upload path (Windview)
sockUploadPathWrite( ) - write to the socket upload path (Windview)

DESCRIPTION
This file contains routines that are used by **wvLib** to pass event data from the target buffers to the host. This particular event-upload path opens a normal network socket connected with the WindView host process to transfer the data.

SEE ALSO
wvTsfsUploadPathLib, **wvFileUploadPathLib**

wvTmrLib

NAME
wvTmrLib – timer library (WindView)

ROUTINES
wvTmrRegister( ) - register a timestamp timer (WindView)

DESCRIPTION
This library allows a WindView timestamp timer to be registered. When this timer is enabled, events are tagged with a timestamp as they are logged.

Seven routines are required: a timestamp routine, a timestamp routine that guarantees interrupt lockout, a routine that enables the timer driver, a routine that disables the timer driver, a routine that specifies the routine to run when the timer hits a rollover, a routine that returns the period of the timer, and a routine that returns the frequency of the timer.

SEE ALSO
wvLib, *WindView User's Guide*

wvTsfsUploadPathLib

NAME **wvTsfsUploadPathLib** – target host connection library using TSFS

ROUTINES
tsfsUploadPathLibInit( ) - initialize **wvTsfsUploadPathLib** library (Windview)
tsfsUploadPathCreate( ) - open an upload path to the host using a TSFS socket (Windview)
tsfsUploadPathClose( ) - close the TSFS-socket upload path (Windview)
tsfsUploadPathWrite( ) - write to the TSFS upload path (Windview)

DESCRIPTION This library contains routines that are used by **wvLib** to transfer event data from the target to the host. This transfer mechanism uses the socket functionality of the Target Server File System (TSFS), and can therefore be used without including any socket or network facilities within the target.

SEE ALSO **wvSockUploadPathLib**, **wvFileUploadPathLib**

zbufLib

NAME

zbufLib – zbuf interface library

ROUTINES

zbufCreate( ) - create an empty zbuf
zbufDelete( ) - delete a zbuf
zbufInsert( ) - insert a zbuf into another zbuf
zbufInsertBuf( ) - create a zbuf segment from a buffer and insert into a zbuf
zbufInsertCopy( ) - copy buffer data into a zbuf
zbufExtractCopy( ) - copy data from a zbuf to a buffer
zbufCut( ) - delete bytes from a zbuf
zbufSplit( ) - split a zbuf into two separate zbufs
zbufDup( ) - duplicate a zbuf
zbufLength( ) - determine the length in bytes of a zbuf
zbufSegFind( ) - find the zbuf segment containing a specified byte location
zbufSegNext( ) - get the next segment in a zbuf
zbufSegPrev( ) - get the previous segment in a zbuf
zbufSegData( ) - determine the location of data in a zbuf segment
zbufSegLength( ) - determine the length of a zbuf segment

DESCRIPTION

This library contains routines to create, build, manipulate, and delete zbufs. Zbufs, also known as "zero copy buffers," are a data abstraction designed to allow software modules to share buffers without unnecessarily copying data.

To support the data abstraction, the subroutines in this library hide the implementation details of zbufs. This also maintains the library's independence from any particular implementation mechanism, thus permitting the zbuf interface to be used with other buffering schemes.

Zbufs have three essential properties. First, a zbuf holds a sequence of bytes. Second, these bytes are organized into one or more segments of contiguous data, although the successive segments themselves are not usually contiguous. Third, the data within a segment may be shared with other segments; that is, the data may be in use by more than one zbuf at a time.

ZBUF TYPES

The following data types are used in managing zbufs:

ZBUF_ID
An arbitrary (but unique) integer that identifies a particular zbuf.

ZBUF_SEG
An arbitrary (but unique within a single zbuf) integer that identifies a segment within a zbuf.

ADDRESSING BYTES IN ZBUFS

The bytes in a zbuf are addressed by the combination *zbufSeg*, *offset*. The *offset* may be positive or negative, and is simply the number of bytes from the beginning of the segment *zbufSeg*.

A *zbufSeg* can be specified as **NULL**, to identify the segment at the beginning of a zbuf. If *zbufseg* is **NULL**, *offset* is the absolute offset to any byte in the zbuf. However, it is more efficient to identify a zbuf byte location relative to the *zbufSeg* that contains it; see **zbufSegFind( )** to convert any *zbufSeg*, *offset* pair to the most efficient equivalent.

Negative *offset* values always refer to bytes before the corresponding *zbufSeg*, and are not usually the most efficient address formulation (though using them may save your program other work in some cases).

The following special *offset* values, defined as constants, allow you to specify the very beginning or the very end of an entire zbuf, regardless of the *zbufSeg* value:

ZBUF_BEGIN
: The beginning of the entire zbuf.

ZBUF_END
: The end of the entire zbuf (useful for appending to a zbuf; see below).

INSERTION AND LIMITS ON OFFSETS

An *offset* is not valid if it points outside the zbuf. Thus, to address data currently within an N-byte zbuf, the valid offsets relative to the first segment are 0 through N-1.

Insertion routines are a special case: they obey the usual convention, but they use *offset* to specify where the new data begins after the insertion is complete. Therefore, the original zbuf data is always inserted just before the byte location addressed by the *offset* value. The value of this convention is that it permits inserting (or concatenating) data either before or after the existing data. To insert before all the data currently in a zbuf segment, use 0 as *offset*. To insert after all the data in an N-byte segment, use N as *offset*. An *offset* of N-1 inserts the data just before the last byte in an N-byte segment.

An *offset* of 0 is always a valid insertion point; for an empty zbuf, 0 is the only valid *offset* (and **NULL** the only valid *zbufSeg*).

SHARING DATA

The routines in this library avoid copying segment data whenever possible. Thus, by passing and manipulating **ZBUF_IDs** rather than copying data, multiple programs can communicate with greater efficiency. However, each program must be aware of data sharing: changes to the data in a zbuf segment are visible to all zbuf segments that reference the data.

To alter your own program's view of zbuf data without affecting other programs, first use **zbufDup( )** to make a new zbuf; then you can use an insertion or deletion routine, such as **zbufInsertBuf( )**, to add a segment that only your program sees (until you pass a zbuf containing it to another program). It is safest to do all direct data manipulation in a private buffer, before enrolling it in a zbuf: in principle, you should regard all zbuf segment data as shared.

Once a data buffer is enrolled in a zbuf segment, the zbuf library is responsible for noticing when the buffer is no longer in use by any program, and freeing it. To support this, **zbufInsertBuf( )** requires that you specify a callback to a free routine each time you build a zbuf segment around an existing buffer. You can use this callback to notify your application when a data buffer is no longer in use.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, this feature is restricted to the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

To use this feature, include the following component: **INCLUDE_ZBUF_SOCK**

SEE ALSO **zbufSockLib**

zbufSockLib

NAME **zbufSockLib** – zbuf socket interface library

ROUTINES

zbufSockLibInit( ) - initialize the zbuf socket interface library
zbufSockSend( ) - send zbuf data to a TCP socket
zbufSockSendto( ) - send a zbuf message to a UDP socket
zbufSockBufSend( ) - create a zbuf from user data and send it to a TCP socket
zbufSockBufSendto( ) - create a zbuf from a user message and send it to a UDP socket
zbufSockRecv( ) - receive data in a zbuf from a TCP socket
zbufSockRecvfrom( ) - receive a message in a zbuf from a UDP socket

DESCRIPTION

This library contains routines that communicate over BSD sockets using the *zbuf interface* described in the **zbufLib** manual page. These zbuf socket calls communicate over BSD sockets in a similar manner to the socket routines in **sockLib**, but they avoid copying data unnecessarily between application buffers and network buffers.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, this feature is accessible from the kernel protection domain only. This restriction does not apply under non-AE versions of VxWorks.

To use this feature, include the **INCLUDE_ZBUF_SOCK** component.

SEE ALSO **zbufLib**, **sockLib**

2

Routines

a0( )

NAME

a0( ) – return the contents of register **a0** (also **a1** - **a7**) (68K)

SYNOPSIS

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

DESCRIPTION

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

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

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

RETURNS

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

SEE ALSO

dbgArchLib, *VxWorks Programmer's Guide: Target Shell*

abort( )

NAME

abort( ) – cause abnormal program termination (ANSI)

SYNOPSIS

```
void abort (void)
```

DESCRIPTION

This routine causes abnormal program termination, unless the signal **SIGABRT** is being caught and the signal handler does not return. VxWorks does not flush output streams, close open streams, or remove temporary files. **abort( )** returns unsuccessful status termination to the host environment by calling:

```
raise (SIGABRT);
```

INCLUDE FILES

stdlib.h

RETURNS

This routine cannot return to the caller.

SEE ALSO

ansiStdlib

abs( )

NAME

abs( ) – compute the absolute value of an integer (ANSI)

SYNOPSIS

```
int abs
    (
    int i                     /* integer for which to return absolute value
*/
    )
```

DESCRIPTION

This routine computes the absolute value of a specified integer. If the result cannot be represented, the behavior is undefined.

INCLUDE FILES

stdlib.h

RETURNS

The absolute value of *i*.

SEE ALSO

ansiStdlib

accept( )

NAME

accept( ) – accept a connection from a socket

SYNOPSIS

```
int accept
    (
    int               s,      /* socket descriptor */
    struct sockaddr * addr,   /* peer address */
    int *             addrlen /* peer address length */
    )
```

DESCRIPTION

This routine accepts a connection on a socket, and returns a new socket created for the connection. The socket must be bound to an address with **bind( )**, and enabled for connections by a call to **listen( )**. The **accept( )** routine dequeues the first connection and creates a new socket with the same properties as *s*. It blocks the caller until a connection is present, unless the socket is marked as non-blocking.

The *addrlen* parameter should be initialized to the size of the available buffer pointed to by *addr*. Upon return, *addrlen* contains the size in bytes of the peer's address stored in *addr*.

WARNING: You must make sure that you do not close the file descriptor on which a task is pending during an **accept( )**. Although the **accept( )** on the closed file descriptor sometimes returns with an error, the **accept( )** can also fail to return at all. Thus, if you need to be able to close a socket connections file descriptor asynchronously, you may need to set up a semaphore-based locking mechanism that prevents the close while an **accept( )** is pending on the file descriptor.

RETURNS A socket descriptor, or **ERROR** if the call fails.

SEE ALSO **sockLib**

acos( )

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

SYNOPSIS

```
double acos
    (
    double x                  /* number between -1 and 1 */
    )
```

DESCRIPTION This routine returns principal value of the arc cosine of *x* in double precision (IEEE double, 53 bits). If *x* is the cosine of an angle *T*, this function returns *T*.

A domain error occurs for arguments not in the range [-1,+1].

INCLUDE FILES math.h

RETURNS The double-precision arc cosine of *x* in the range [0,pi] radians.

Special cases:
If *x* is NaN, **acos( )** returns *x*.
If |x|>1, it returns NaN.

SEE ALSO **ansiMath**, **mathALib**

acosf( )

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

SYNOPSIS

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

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

INCLUDE FILES
math.h

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

SEE ALSO
mathALib

aioPxLibInit( )

NAME
aioPxLibInit( ) – initialize the asynchronous I/O (AIO) library

SYNOPSIS

```
STATUS aioPxLibInit
    (
    int lioMax                /* max outstanding lio calls */
    )
```

DESCRIPTION
This routine initializes the AIO library. It should be called only once after the I/O system has been initialized. *lioMax* specifies the maximum number of outstanding **lio_listio( )** calls at one time. If *lioMax*is zero, the default value of **AIO_CLUST_MAX** is used.

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

ERRNO
S_aioPxLib_IOS_NOT_INITIALIZED

SEE ALSO
aioPxLib

A

aioShow( )

NAME
aioShow( ) – show AIO requests

SYNOPSIS

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

DESCRIPTION
This routine displays the outstanding AIO requests.

WARNING: The *drvNum* parameter is not currently used.

RETURNS
OK, always.

SEE ALSO
aioPxShow

aioSysInit( )

NAME
aioSysInit( ) – initialize the AIO system driver

SYNOPSIS

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

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

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

SEE ALSO
aioSysDrv

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

INCLUDE FILES

aio.h

SEE ALSO

aioPxLib

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**

INCLUDE FILES **aio.h**

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

---

aio_return( )

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

SYNOPSIS

```
size_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**

INCLUDE FILES **aio.h**

SEE ALSO **aioPxLib**

aio_suspend( )

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

SYNOPSIS

```
int aio_suspend
    (
    const struct aiocb *    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**

INCLUDE FILES **aio.h**

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**

INCLUDE FILES **aio.h**

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

---

alarm( )

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

SYNOPSIS

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

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

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

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

SEE ALSO **timerLib**

arpAdd( )

NAME

arpAdd( ) – create or modify an ARP table entry

SYNOPSIS

```
STATUS arpAdd
    (
    char * pHost,           /* host name or IP address */
    char * pEther,          /* Ethernet address */
    int    flags            /* ARP flags */
    )
```

DESCRIPTION

This routine assigns an Ethernet address to an IP address in the ARP table. The *pHost* parameter specifies the host by name or by Internet address using standard dotted decimal notation. The *pEther* parameter provides the Ethernet address as six hexadecimal bytes (between 0 and ff) separated by colons. A new entry is created for the specified host if necessary. Otherwise, the existing entry is changed to use the given Ethernet address.

The *flags* parameter combines any of the following options:

ATF_PERM (0x04)
Create a permanent ARP entry which will not time out.

ATF_PUBL (0x08)
Publish this entry. The host will respond to ARP requests even if the *pHost* parameter does not match a local IP address. This setting provides a limited form of proxy ARP.

ATF_PROXY (0x10)
Use a "wildcard" hardware address. The proxy server uses this setting to support multiple proxy networks. The entry always supplies the hardware address of the sending interface.

EXAMPLE

Create a permanent ARP table entry for "myHost" with Ethernet address 0:80:f9:1:2:3:

```
arpAdd ("myHost", "0:80:f9:1:2:3", 0x4);
```

Assuming "myHost" has the Internet address "90.0.0.3", the following call changes the Ethernet address to 0:80:f9:1:2:4. No additional flags are set for that entry.

```
arpAdd ("90.0.0.3", "0:80:f9:1:2:4", 0);
```

RETURNS

OK, or **ERROR** if unsuccessful.

ERRNO

S_arpLib_INVALID_ARGUMENT
S_arpLib_INVALID_HOST
S_arpLib_INVALID_ENET_ADDRESS
S_arpLib_INVALID_FLAG
or results of low-level ioctl call.

SEE ALSO **arpLib**

arpDelete( )

NAME **arpDelete( )** – remove an ARP table entry

SYNOPSIS

```
STATUS arpDelete
    (
    char * pHost              /* host name or IP address */
    )
```

DESCRIPTION This routine deletes an ARP table entry. The *pHost* parameter indicates the target entry using the host name or Internet address.

EXAMPLE

```
arpDelete ("91.0.0.3")
arpDelete ("myHost")
```

RETURNS **OK**, or **ERROR** if unsuccessful.

ERRNO **S_arpLib_INVALID_ARGUMENT**
S_arpLib_INVALID_HOST

SEE ALSO **arpLib**

arpFlush( )

NAME **arpFlush( )** – flush all entries in the system ARP table

SYNOPSIS

```
void arpFlush (void)
```

DESCRIPTION This routine flushes all non-permanent entries in the ARP cache.

RETURNS N/A

SEE ALSO **arpLib**

arpResolve( )

NAME **arpResolve( )** – resolve a hardware address for a specified Internet address

SYNOPSIS

```
STATUS arpResolve
    (
    char * targetAddr,        /* name or Internet address of target */
    char * pHwAddr,           /* where to return the H/W address */
    int    numTries,          /* number of times to try ARPing (-1 means */
                              /* try forever) */
    int    numTicks           /* number of ticks between ARPs */
    )
```

DESCRIPTION This routine uses the Address Resolution Protocol (ARP) and internal ARP cache to resolve the hardware address of a machine that owns the Internet address given in *targetAddr*.

The hardware address is copied to *pHwAddr* as network byte order, if the resolution of *targetAddr* is successful. *pHwAddr* must point to a buffer which is large enough to receive the address.

NOTE: RFC 1122 prohibits sending more than one arp request per second. Any numTicks value that would result in a shorter time than this is ignored.

RETURNS **OK** if the address is resolved successfully, or **ERROR** if *pHwAddr* is **NULL**, *targetAddr* is invalid, or address resolution is unsuccessful.

ERRNO **S_arpLib_INVALID_ARGUMENT**
S_arpLib_INVALID_HOST

SEE ALSO **arpLib**

A

arpShow( )

NAME **arpShow( )** – display entries in the system ARP table

SYNOPSIS `void arpShow (void)`

DESCRIPTION This routine displays the current Internet-to-Ethernet address mappings in the ARP table.

EXAMPLE

```
-> arpShow
LINK LEVEL ARP TABLE
Destination      LL Address        Flags  Refcnt Use        Interface
---------------------------------------------------------------------
90.0.0.63        08:00:3e:23:79:e7 0x405  0      82         lo0
---------------------------------------------------------------------
```

Some configuration is required when this routine is to be used remotely over the network, *e.g.*, through a telnet session or through the host shell using **WDB_COMM_NETWORK**. If more than 5 entries are expected in the table the parameter **RT_BUFFERED_DISPLAY** should be set to **TRUE** to prevent a possible deadlock. This requires a buffer whose size can be set with **RT_DISPLAY_MEMORY**. It will limit the number of entries that can be displayed (each entry requires approx. 70 bytes).

RETURNS N/A

SEE ALSO **netShow**

arptabShow( )

NAME **arptabShow( )** – display the known ARP entries

SYNOPSIS `void arptabShow (void)`

DESCRIPTION This routine displays current Internet-to-Ethernet address mappings in the ARP table.

RETURNS N/A

SEE ALSO **netShow**

asctime( )

NAME

asctime( ) – convert broken-down time into a string (ANSI)

SYNOPSIS

```
char * asctime
    (
    const struct tm * timeptr /* broken-down time */
    )
```

DESCRIPTION

This routine converts the broken-down time pointed to by *timeptr* into a string of the form:

```
SUN SEP 16 01:03:52 1973\n\0
```

This routine is not reentrant. For a reentrant version, see **asctime_r( )**.

INCLUDE FILES

time.h

RETURNS

A pointer to the created string.

SEE ALSO

ansiTime

asctime_r( )

NAME

asctime_r( ) – convert broken-down time into a string (POSIX)

SYNOPSIS

```
int asctime_r
    (
    const struct tm * timeptr,    /* broken-down time */
    char *            asctimeBuf, /* buffer to contain string */
    size_t *          buflen      /* size of buffer */
    )
```

DESCRIPTION

This routine converts the broken-down time pointed to by *timeptr* into a string of the form:

```
SUN SEP 16 01:03:52 1973\n\0
```

The string is copied to *asctimeBuf*. **asctimer( )** is the POSIX re-entrant version of **asctime( )**.

INCLUDE FILES

time.h

RETURNS

The size of the created string.

SEE ALSO

ansiTime

A

asin( )

NAME **asin( )** – compute an arc sine (ANSI)

SYNOPSIS

```
double asin
    (
    double x                  /* number between -1 and 1 */
    )
```

DESCRIPTION This routine returns the principal value of the arc sine of *x* in double precision (IEEE double, 53 bits). If *x* is the sine of an angle *T*, this function returns *T*.

A domain error occurs for arguments not in the range [-1,+1].

INCLUDE FILES **math.h**

RETURNS The double-precision arc sine of *x* in the range [-pi/2,pi/2] radians.

Special cases:
If *x* is NaN, **asin( )** returns *x*.
If |x>1, it returns NaN.

SEE ALSO **ansiMath**, **mathALib**

asinf( )

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

SYNOPSIS

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

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

INCLUDE FILES **math.h**

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

SEE ALSO **mathALib**

assert( )

NAME **assert( )** – put diagnostics into programs (ANSI)

SYNOPSIS

```
void assert
    (
    int a
    )
```

DESCRIPTION If an expression is false (that is, equal to zero), the **assert( )** macro writes information about the failed call to standard error in an implementation-defined format. It then calls **abort( )**. The diagnostic information includes:

- the text of the argument
- the name of the source file (value of preprocessor macro __**FILE**__)
- the source line number (value of preprocessor macro __**LINE**__)

INCLUDE **stdio.h**, **stdlib.h**, **assert.h**

RETURNS N/A

SEE ALSO **ansiAssert**

atan( )

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

SYNOPSIS

```
double atan
    (
    double x                  /* tangent of an angle */
    )
```

DESCRIPTION This routine returns the principal value of the arc tangent of *x* in double precision (IEEE double, 53 bits). If *x* is the tangent of an angle *T*, this function returns *T* (in radians).

INCLUDE FILES **math.h**

RETURNS The double-precision arc tangent of *x* in the range [-pi/2,pi/2] radians. Special case: if *x* is NaN, **atan( )** returns *x* itself.

SEE ALSO **ansiMath**, **mathALib**

A

atan2( )

NAME **atan2( )** – compute the arc tangent of y/x (ANSI)

SYNOPSIS

```
double atan2
    (
    double y,                   /* numerator */
    double x                    /* denominator */
    )
```

DESCRIPTION This routine returns the principal value of the arc tangent of y/x in double precision (IEEE double, 53 bits). This routine uses the signs of both arguments to determine the quadrant of the return value. A domain error may occur if both arguments are zero.

INCLUDE FILES **math.h**

RETURNS The double-precision arc tangent of y/x, in the range [-pi,pi] radians.

Special cases:
Notations: atan2(y,x) == ARG (x+iy) == ARG(x,y).

ARG(NAN, (anything))	is	NaN
ARG((anything), NaN)	is	NaN
ARG(+(anything but NaN), +-0)	is	+-0
ARG(-(anything but NaN), +-0)	is	+-PI
ARG(0, +-(anything but 0 and NaN))	is	+-PI/2
ARG(+INF, +-(anything but INF and NaN))	is	+-0
ARG(-INF, +-(anything but INF and NaN))	is	+-PI
ARG(+INF, +-INF)	is	+-PI/4
ARG(-INF, +-INF)	is	+-3PI/4
ARG((anything but 0, NaN, and INF),+-INF)	is	+-PI/2

SEE ALSO **ansiMath**, **mathALib**

atan2f( )

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

SYNOPSIS

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

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

INCLUDE FILES **math.h**

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

SEE ALSO **mathALib**

atanf( )

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

SYNOPSIS

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

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

INCLUDE FILES **math.h**

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

SEE ALSO **mathALib**

A

atexit( )

NAME **atexit( )** – call a function at program termination (Unimplemented) (ANSI)

SYNOPSIS

```
int atexit
    (
    void (* __func)(void)     /* pointer to a function */
    )
```

DESCRIPTION This routine is unimplemented. VxWorks task exit hooks provide this functionality.

INCLUDE FILES **stdlib.h**

RETURNS **ERROR**, always.

SEE ALSO **ansiStdlib**, **taskHookLib**

atof( )

NAME **atof( )** – convert a string to a **double** (ANSI)

SYNOPSIS

```
double atof
    (
    const char * s            /* pointer to string */
    )
```

DESCRIPTION This routine converts the initial portion of the string *s* to double-precision representation.

Its behavior is equivalent to:

```
strtod (s, (char **) NULL);
```

INCLUDE FILES **stdlib.h**

RETURNS The converted value in double-precision representation.

SEE ALSO **ansiStdlib**

atoi( )

NAME

atoi( ) – convert a string to an **int** (ANSI)

SYNOPSIS

```
int atoi
    (
    const char * s              /* pointer to string */
    )
```

DESCRIPTION

This routine converts the initial portion of the string *s* to **int** representation.

Its behavior is equivalent to:

```
    (int) strtol (s, (char **) NULL, 10);
```

INCLUDE FILES

stdlib.h

RETURNS

The converted value represented as an **int**.

SEE ALSO

ansiStdlib

atol( )

NAME

atol( ) – convert a string to a **long** (ANSI)

SYNOPSIS

```
long atol
    (
    const register char * s   /* pointer to string */
    )
```

DESCRIPTION

This routine converts the initial portion of the string *s* to long integer representation.

Its behavior is equivalent to:

```
    strtol (s, (char **) NULL, 10);
```

INCLUDE FILES

stdlib.h

RETURNS

The converted value represented as a **long**.

SEE ALSO

ansiStdlib

A

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.

SEE ALSO

usrFsLib

b( )

NAME

b( ) – set or display breakpoints

SYNOPSIS

```
STATUS b
    (
    INSTR * addr,              /* where to set breakpoint, or 0 = display */
                               /* all breakpoints */
    int     task,              /* task for which to set breakboint, 0 = */
                               /* set all tasks */
    int     count,             /* number of passes before hit */
    BOOL    quiet              /* TRUE = don't print debugging info, FALSE */
                               /* = print debugging info */
    )
```

DESCRIPTION

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

```
-> b
```

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

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

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

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

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

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

RETURNS

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

SEE ALSO

dbgLib, **bd( )**, **taskOptionsSet( )**, *VxWorks Programmer's Guide: Target Shell,* **windsh**, *Tornado User's Guide: Shell*

B

bcmp( )

NAME **bcmp( )** – compare one buffer to another

SYNOPSIS

```
int bcmp
    (
    char * buf1,            /* pointer to first buffer */
    char * buf2,            /* pointer to second buffer */
    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*.

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

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

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

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

SEE ALSO

bLib, **bcopy( )**

bd( )

NAME

bd( ) – delete a breakpoint

SYNOPSIS

```
STATUS bd
    (
    INSTR * addr,               /* address of breakpoint to delete */
    int     task                /* task for which to delete breakpoint, 0 = */
                                /* delete for all tasks */
    )
```

DESCRIPTION

This routine deletes a specified breakpoint.

To execute, enter:

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

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

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

SEE ALSO **dbgLib**, **b( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

bdall( )

NAME **bdall( )** – delete all breakpoints

SYNOPSIS

```
STATUS bdall
    (
    int task                  /* task for which to delete breakpoints, 0 */
                              /* = delete for all tasks */
    )
```

DESCRIPTION This routine removes all breakpoints.

To execute, enter:

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

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

RETURNS **OK**, always.

SEE ALSO **dbgLib**, **bd( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

bfill( )

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

SYNOPSIS

```
void bfill
    (
    char * buf,               /* pointer to buffer */
    int    nbytes,            /* number of bytes to fill */
    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

SEE ALSO
bLib, **bfillBytes( )**

bfillBytes( )

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

SYNOPSIS

```
void bfillBytes
    (
    char * buf,              /* pointer to buffer */
    int    nbytes,           /* number of bytes to fill */
    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

SEE ALSO
bLib, **bfill( )**

bh( )

NAME
bh( ) – set a hardware breakpoint

SYNOPSIS

```
STATUS bh
    (
    INSTR * addr,             /* where to set breakpoint, or 0 = display */
                              /* all breakpoints */
    int     access,           /* access type (arch dependant) */
    int     task,             /* task for which to set breakboint, 0 = */
```

```
                                /* set all tasks */
    int     count,              /* number of passes before hit */
    BOOL    quiet               /* TRUE = don't print debugging info, FALSE */
                                /* = print debugging info */
    )
```

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

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

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

SEE ALSO **dbgLib**, **b( )**, *VxWorks Programmer's Guide: Target Shell*

---

bind( )

NAME **bind( )** – bind a name to a socket

SYNOPSIS

```
STATUS bind
    (
    int               s,      /* socket descriptor */
    struct sockaddr * name,   /* name to be bound */
    int               namelen /* length of name */
    )
```

DESCRIPTION This routine associates a network address (also referred to as its "name") with a specified socket so that other processes can connect or send to it. When a socket is created with **socket( )**, it belongs to an address family but has no assigned name.

RETURNS **OK**, or **ERROR** if there is an invalid socket, the address is either unavailable or in use, or the socket is already bound.

SEE ALSO **sockLib**

B

bindresvport( )

NAME **bindresvport( )** – bind a socket to a privileged IP port

SYNOPSIS

```
STATUS bindresvport
    (
    int                   sd,  /* socket to be bound */
    struct sockaddr_in * sin   /* socket address -- value/result */
    )
```

DESCRIPTION This routine picks a port number between 600 and 1023 that is not being used by any other programs and binds the socket passed as *sd* to that port. Privileged IP ports (numbers between and including 0 and 1023) are reserved for privileged programs.

RETURNS **OK**, or **ERROR** if the address family specified in *sin* is not supported or the call fails.

SEE ALSO **remLib**

binvert( )

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

SYNOPSIS

```
void binvert
    (
    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

SEE ALSO **bLib**

bootBpAnchorExtract( )

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

SYNOPSIS

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

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

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

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

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

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

SEE ALSO **bootLib**

bootChange( )

NAME **bootChange( )** – change the boot line

SYNOPSIS

```
void bootChange (void)
```

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

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

RETURNS N/A

SEE ALSO **usrLib**, **windsh**, *Tornado User's Guide: Shell*

bootLeaseExtract( )

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

SYNOPSIS

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

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

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

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

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

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

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

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

RETURNS

2 if both lease values are specified correctly in *string*, or
-2 if one of the two values is specified incorrectly.

If only the lease duration is found, it returns:
1 if the lease duration in *string* is specified correctly,
0 if the lease duration is not specified in *string*, or
-1 if an invalid lease duration is specified in *string*.

SEE ALSO

bootLib

bootNetmaskExtract( )

NAME

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

SYNOPSIS

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

DESCRIPTION

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

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

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

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

RETURNS

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

SEE ALSO

bootLib

bootParamsPrompt( )

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

SYNOPSIS

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

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

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

RETURNS N/A

SEE ALSO **bootLib**

bootParamsShow( )

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

SYNOPSIS

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

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

RETURNS N/A

SEE ALSO **bootLib**

bootpLibInit( )

NAME **bootpLibInit( )** – BOOTP client library initialization

SYNOPSIS

```
STATUS bootpLibInit
    (
    int maxSize               /* largest link-level header, in bytes */
    )
```

DESCRIPTION This routine creates and initializes the global data structures used by the BOOTP client library to obtain configuration parameters. The *maxSize*parameter specifies the largest link level header for all supported devices. This value determines the maximum length of the outgoing IP packet containing a BOOTP message.

This routine must be called before using any other library routines. The routine is called automatically if **INCLUDE_BOOTP** is defined at the time the system is built and uses the **BOOTP_MAXSIZE** configuration setting for the *maxSize* parameter.

RETURNS **OK**, or **ERROR** if initialization fails.

ERRNO **S_bootpLib_MEM_ERROR**

SEE ALSO **bootpLib**

bootpMsgGet( )

NAME **bootpMsgGet( )** – send a BOOTP request message and retrieve reply

SYNOPSIS

```
STATUS bootpMsgGet
    (
    struct ifnet *   pIf,       /* network device for message exchange */
    struct in_addr * pIpDest,   /* destination IP address for request */
    USHORT           srcPort,   /* UDP source port for request */
    USHORT           dstPort,   /* UDP destination port for request */
    BOOTP_MSG *      pBootpMsg, /* request template and reply storage */
    u_int            maxSends   /* maximum number of transmit attempts */
    )
```

DESCRIPTION This routine sends a BOOTP request using the *pIf* network interface and waits for a reply. *pIpDest* specifies the destination IP address. It must be equal to either the broadcast address (255.255.255.255) or the IP address of a specific BOOTP server directly reachable using the network interface. The interface must support broadcasting in the first case.

The *srcPort* and *dstPort* arguments support sending and receiving BOOTP messages with arbitrary UDP ports. To receive replies, any BOOTP server must send those responses to the source port from the request. To comply with the RFC 1542 clarification, the request message must be sent to the reserved BOOTP server port (67) using the reserved BOOTP client port (68).

Except for the UDP port numbers, this routine only sets the **bp_xid** and **bp_secs** fields in the outgoing BOOTP message. All other fields in that message use the values from the *pBootpMsg* argument, which later holds the contents of any BOOTP reply received.

The *maxSends* parameter specifies the total number of requests to transmit if no reply is received. The retransmission interval starts at 4 seconds and doubles with each attempt up to a maximum of 64 seconds. Any subsequent retransmissions will occur at that maximum interval. To reduce the chances of network flooding, the timeout interval before each retransmission includes a randomized delay of plus or minus one second from the base value. After the final transmission, this routine will wait for the current interval to expire before returning a timeout error.

NOTE: The target must be able to respond to an ARP request for any IP address specified in the request template's **bp_ciaddr** field.

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

ERRNO
S_bootpLib_INVALID_ARGUMENT
S_bootpLib_NO_BROADCASTS
S_bootpLib_TIME_OUT

SEE ALSO **bootpLib**

bootpParamsGet( )

NAME **bootpParamsGet( )** – retrieve boot parameters using BOOTP

SYNOPSIS

```
STATUS bootpParamsGet
    (
    struct ifnet *       pIf,         /* network device used by client */
    u_int                maxSends,    /* maximum transmit attempts */
    struct in_addr *     pClientAddr, /* retrieved client address buffer */
    struct in_addr *     pServerAddr, /* buffer for server's IP address */
    char *               pHostName,   /* 64 byte (max) host name buffer */
    char *               pBootFile,   /* 128 byte (max) file name buffer */
    struct bootpParams * pBootpParams /* parameters descriptor */
    )
```

DESCRIPTION

This routine performs a BOOTP message exchange according to the process described in RFC 1542, so the server and client UDP ports are always equal to the defined values of 67 and 68.

The *pIf* argument indicates the network device which will be used to send and receive BOOTP messages. The BOOTP client only supports devices attached to the IP protocol with the MUX/END interface. The MTU size must be large enough to receive an IP packet of 328 bytes (corresponding to the BOOTP message length of 300 bytes). The specified device also must be capable of sending broadcast messages, unless this routine sends the request messages directly to the IP address of a specific server.

The *maxSends* parameter specifies the total number of requests before this routine stops waiting for a reply. After the final request, this routine will wait for the current interval before returning error. The timeout interval following each request follows RFC 1542, beginning at 4 seconds and doubling until a maximum limit of 64 seconds.

The *pClientAddr* parameter provides optional storage for the assigned IP address from the **yiaddr** field of a BOOTP reply. Since this routine can execute before the system is capable of accepting unicast datagrams or responding to ARP requests for a specific IP address, the corresponding **ciaddr** field in the BOOTP request message is equal to zero.

The *pServerAddr* parameter provides optional storage for the IP address of the responding server (from the **siaddr** field of a BOOTP reply). This routine broadcasts the BOOTP request message unless this buffer is available (*i.e.,* not **NULL**) and contains the explicit IP address of a BOOTP server as a non-zero value.

The *pHostName* parameter provides optional storage for the server's host name (from the **sname** field of a BOOTP reply). This routine also copies any initial string in that buffer into the **sname** field of the BOOTP request (which restricts booting to a specified host).

The *pBootFile* parameter provides optional storage for the boot file name (from the **file** field of a BOOTP reply). This routine also copies any initial string in that buffer into the **file** field of the BOOTP request message, which typically supplies a generic name to the server.

The remaining fields in the BOOTP request message use the values which RFC 1542 defines. In particular, the **giaddr** field is set to zero and the suggested "magic cookie" is always inserted in the (otherwise empty) **vend** field.

The *pBootpParams* argument provides access to any options defined in RFC 1533 using the following definition:

```
struct bootpParams
    {
    struct in_addr *            netmask;
    unsigned short *            timeOffset;
    struct in_addr_list *       routers;
    struct in_addr_list *       timeServers;
    struct in_addr_list *       nameServers;
    struct in_addr_list *       dnsServers;
```

```
struct in_addr_list *      logServers;
struct in_addr_list *      cookieServers;
struct in_addr_list *      lprServers;
struct in_addr_list *      impressServers;
struct in_addr_list *      rlpServers;
char *                     clientName;
unsigned short *           filesize;
char *                     dumpfile;
char *                     domainName;
struct in_addr *           swapServer;
char *                     rootPath;
char *                     extoptPath;
unsigned char *            ipForward;
unsigned char *            nonlocalSourceRoute;
struct in_addr_list *      policyFilter;
unsigned short *           maxDgramSize;
unsigned char *            ipTTL;
unsigned long *            mtuTimeout;
struct ushort_list *       mtuTable;
unsigned short *           intfaceMTU;
unsigned char *            allSubnetsLocal;
struct in_addr *           broadcastAddr;
unsigned char *            maskDiscover;
unsigned char *            maskSupplier;
unsigned char *            routerDiscover;
struct in_addr *           routerDiscAddr;
struct in_addr_list *      staticRoutes;
unsigned char *            arpTrailers;
unsigned long *            arpTimeout;
unsigned char *            etherPacketType;
unsigned char *            tcpTTL;
unsigned long *            tcpInterval;
unsigned char *            tcpGarbage;
char *                     nisDomain;
struct in_addr_list *      nisServers;
struct in_addr_list *      ntpServers;
char *                     vendString;
struct in_addr_list *      nbnServers;
struct in_addr_list *      nbddServers;
unsigned char *            nbNodeType;
char *                     nbScope;
struct in_addr_list *      xFontServers;
struct in_addr_list *      xDisplayManagers;
char *                     nispDomain;
struct in_addr_list *      nispServers;
struct in_addr_list *      ipAgents;
```

```
    struct in_addr_list *       smtpServers;
    struct in_addr_list *       pop3Servers;
    struct in_addr_list *       nntpServers;
    struct in_addr_list *       wwwServers;
    struct in_addr_list *       fingerServers;
    struct in_addr_list *       ircServers;
    struct in_addr_list *       stServers;
    struct in_addr_list *       stdaServers;
    };
```

This structure allows the retrieval of any BOOTP option specified in RFC 1533. The list of 2-byte (unsigned short) values is defined as:

```
struct ushort_list
    {
    unsigned char       num;
    unsigned short *    shortlist;
    };
```

The IP address lists use the following similar definition:

```
struct in_addr_list
    {
    unsigned char       num;
    struct in_addr *    addrlist;
    };
```

When these lists are present, the routine stores values retrieved from the BOOTP reply in the location indicated by the **shortlist** or **addrlist** members. The amount of space available is indicated by the **num** member. When the routine returns, the **num** member indicates the actual number of entries retrieved. In the case of **bootpParams.policyFilter.num** and **bootpParams.staticRoutes.num**, the **num** member value should be interpreted as the number of IP address pairs requested and received.

The contents of the BOOTP parameter descriptor implicitly selects options for retrieval from the BOOTP server. This routine attempts to retrieve the values for any options whose corresponding field pointers are non-**NULL** values. To obtain these parameters, the BOOTP server must support the vendor-specific options described in RFC 1048 (or its successors) and the corresponding parameters must be specified in the BOOTP server database. Where meaningful, the values are returned in host byte order.

The BOOTP request issued during system startup with this routine attempts to retrieve a subnet mask for the boot device, in addition to the host and client addresses and the boot file name.

RETURNS **OK**, or **ERROR** if unsuccessful.

SEE ALSO **bootpLib**, **bootLib**, RFC 1048, RFC 1533

B

bootStringToStruct( )

NAME

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

SYNOPSIS

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

DESCRIPTION

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

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

RETURNS

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

SEE ALSO

bootLib

bootStructToString( )

NAME

bootStructToString( ) – construct a boot line

SYNOPSIS

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

DESCRIPTION

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

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

RETURNS

OK.

SEE ALSO

bootLib

bpfDevCreate( )

NAME

bpfDevCreate( ) – create Berkeley Packet Filter device

SYNOPSIS

```
STATUS bpfDevCreate
    (
    char * pDevName,          /* I/O system device name */
    int    numUnits,          /* number of device units */
    int    bufSize            /* BPF device block size (0 for default) */
    )
```

DESCRIPTION

This routine creates a Berkeley Packet Filter device. Each of the *numUnits* units corresponds to a single available file descriptor for monitoring a network device. The *pDevName* parameter provides the name of the BPF device to the I/O system. The default name of "/bpf" (assigned if *pDevName* is **NULL**) produces units named "/bpf0", "/bpf1", etc., up to the *numUnits* limit.

RETURNS

OK, or **ERROR** if device creation failed.

ERRNO

S_ioLib_NO_DRIVER

SEE ALSO

bpfDrv

bpfDevDelete( )

NAME

bpfDevDelete( ) – destroy Berkeley Packet Filter device

SYNOPSIS

```
STATUS bpfDevDelete
    (
    char * pDevName           /* name of BPF device to remove */
    )
```

DESCRIPTION

This routine removes a Berkeley Packet Filter device and releases all allocated memory. It will close any open files using the device.

RETURNS

OK, or **ERROR** if device not found

ERRNO

S_ioLib_NO_DRIVER

SEE ALSO

bpfDrv

B

bpfDrv( )

NAME **bpfDrv( )** – initialize the BPF driver

SYNOPSIS

```
STATUS bpfDrv (void)
```

DESCRIPTION This routine installs the Berkeley Packet Filter driver for access through the I/O system. It is required before performing any I/O operations and is executed automatically if **INCLUDE_BPF** is defined at the time the system is built. Subsequent calls to the routine just count the number of users with BPF access.

RETURNS **OK**, or **ERROR** if initialization fails.

ERRNO N/A

SEE ALSO **bpfDrv**

bsearch( )

NAME **bsearch( )** – perform a binary search (ANSI)

SYNOPSIS

```
void * bsearch
    (
    const void * key,           /* element to match */
    const void * base0,         /* initial element in array */
    size_t       nmemb,         /* array to search */
    size_t       size,          /* size of array element */
    int (* compar) (const void * , const void * )
                                /* comparison function */
    )
```

DESCRIPTION This routine searches an array of *nmemb* objects, the initial element of which is pointed to by *base0*, for an element that matches the object pointed to by *key*. The *size* of each element of the array is specified by *size*.

The comparison function pointed to by *compar* is called with two arguments that point to the *key* object and to an array element, in that order. The function shall return an integer less than, equal to, or greater than zero if the *key* object is considered, respectively, to be less than, to match, or to be greater than the array element. The array shall consist of all the elements that compare greater than the *key* object, in that order.

INCLUDE FILES **stdlib.h**

RETURNS A pointer to a matching element of the array, or a **NULL** pointer if no match is found. If two elements compare as equal, which element is matched is unspecified.

SEE ALSO **ansiStdlib**

bswap( )

NAME **bswap( )** – swap buffers

SYNOPSIS

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

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

RETURNS 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

SEE ALSO **bLib**

c( )

NAME

c( ) – continue from a breakpoint

SYNOPSIS

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

DESCRIPTION

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

To execute, enter:

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

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

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

RETURNS

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

SEE ALSO

dbgLib, **tr( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

cache4kcLibInit( )

NAME

cache4kcLibInit( ) – initialize the 4kc cache library

SYNOPSIS

```
STATUS cache4kcLibInit
    (
    CACHE_MODE instMode,       /* instruction cache mode */
    CACHE_MODE dataMode,       /* data cache mode */
    UINT32     iCacheSize,
    UINT32     iCacheLineSize,
    UINT32     dCacheSize,
    UINT32     dCacheLineSize
    )
```

DESCRIPTION
This routine initializes the function pointers for the 4kc cache library. The board support package can select this cache library by assigning the function pointer *sysCacheLibInit* to **cache4kcLibInit( )**.

RETURNS
OK.

SEE ALSO
cache4kcLib

---

cacheArchClearEntry( )

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

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

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

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

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

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

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

SEE ALSO
cacheArchLib

C

cacheArchLibInit( )

NAME

cacheArchLibInit( ) – initialize the cache library

SYNOPSIS

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

DESCRIPTION

This routine initializes the cache library for the following processor cache families: Motorola 68K, Intel 960, Intel x86, PowerPC ARM, and the Solaris and Windows simulators. It initializes the function pointers and configures the caches to the specified cache modes.

68K PROCESSORS

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

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

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

x86 PROCESSORS

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

POWER PC PROCESSORS

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

ARM PROCESSORS

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

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

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

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

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

RETURNS

OK

SEE ALSO

cacheArchLib

C

cacheAuLibInit( )

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

SYNOPSIS

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

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

RETURNS OK.

SEE ALSO **cacheAuLib**

cacheClear( )

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

SYNOPSIS

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

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

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

SEE ALSO **cacheLib**

cacheCy604ClearLine( )

NAME **cacheCy604ClearLine( )** – clear a line from a CY7C604 cache

SYNOPSIS

```
STATUS cacheCy604ClearLine
    (
    CACHE_TYPE cache,         /* cache to clear */
    void *     address        /* virtual address */
    )
```

DESCRIPTION This routine flushes and invalidates a specified line from the specified CY7C604 cache.

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

SEE ALSO **cacheCy604Lib**

cacheCy604ClearPage( )

NAME **cacheCy604ClearPage( )** – clear a page from a CY7C604 cache

SYNOPSIS

```
STATUS cacheCy604ClearPage
    (
    CACHE_TYPE cache,         /* cache to clear */
    void *     address        /* virtual address */
    )
```

DESCRIPTION This routine flushes and invalidates the specified page from the specified CY7C604 cache.

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

SEE ALSO **cacheCy604Lib**

C

cacheCy604ClearRegion( )

NAME **cacheCy604ClearRegion( )** – clear a region from a CY7C604 cache

SYNOPSIS

```
STATUS cacheCy604ClearRegion
    (
    CACHE_TYPE cache,         /* cache to clear */
    void *     address        /* virtual address */
    )
```

DESCRIPTION This routine flushes and invalidates a specified region from the specified CY7C604 cache.

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

SEE ALSO **cacheCy604Lib**

cacheCy604ClearSegment( )

NAME **cacheCy604ClearSegment( )** – clear a segment from a CY7C604 cache

SYNOPSIS

```
STATUS cacheCy604ClearSegment
    (
    CACHE_TYPE cache,         /* cache to clear */
    void *     address        /* virtual address */
    )
```

DESCRIPTION This routine flushes and invalidates a specified segment from the specified CY7C604 cache.

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

SEE ALSO **cacheCy604Lib**

cacheCy604LibInit( )

NAME **cacheCy604LibInit( )** – initialize the Cypress CY7C604 cache library

SYNOPSIS

```
STATUS cacheCy604LibInit
    (
    CACHE_MODE instMode,      /* instruction cache mode */
    CACHE_MODE dataMode       /* data cache mode */
    )
```

DESCRIPTION This routine initializes the function pointers for the Cypress CY7C604 cache library. The board support package can select this cache library by assigning the function pointer **sysCacheLibInit** to **cacheCy604LibInit( )**.

The available cache modes are **CACHE_WRITETHROUGH** and **CACHE_COPYBACK**. Write-through uses "no-write allocate"; copyback uses "write allocate."

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

SEE ALSO **cacheCy604Lib**

cacheDisable( )

NAME **cacheDisable( )** – disable the specified cache

SYNOPSIS

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

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

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

SEE ALSO **cacheLib**

cacheDmaFree( )

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

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

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

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

SEE ALSO
cacheLib

cacheDmaMalloc( )

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

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

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

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

SEE ALSO
cacheLib

cacheDrvFlush( )

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

SYNOPSIS

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

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

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

SEE ALSO **cacheLib**

cacheDrvInvalidate( )

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

SYNOPSIS

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

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

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

SEE ALSO **cacheLib**

cacheDrvPhysToVirt( )

NAME **cacheDrvPhysToVirt( )** – translate a physical address for drivers

SYNOPSIS

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

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

RETURNS The virtual address that maps to the physical address argument.

SEE ALSO **cacheLib**

cacheDrvVirtToPhys( )

NAME **cacheDrvVirtToPhys( )** – translate a virtual address for drivers

SYNOPSIS

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

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

RETURNS The physical address translation of a virtual address argument.

SEE ALSO **cacheLib**

cacheEnable( )

NAME **cacheEnable( )** – enable the specified cache

SYNOPSIS

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

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

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

SEE ALSO **cacheLib**

cacheFlush( )

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

SYNOPSIS

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

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

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

SEE ALSO **cacheLib**

C

cacheInvalidate( )

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

SYNOPSIS

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

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

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

SEE ALSO **cacheLib**

cacheLibInit( )

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

SYNOPSIS

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

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

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

SEE ALSO **cacheLib**

cacheLock( )

NAME

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

SYNOPSIS

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

DESCRIPTION

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

RETURNS

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

SEE ALSO

cacheLib

cacheMb930ClearLine( )

NAME

cacheMb930ClearLine( ) – clear a line from an MB86930 cache

SYNOPSIS

```
STATUS cacheMb930ClearLine
    (
    CACHE_TYPE cache,         /* cache to clear entry */
    void *     address        /* virtual address */
    )
```

DESCRIPTION

This routine flushes and invalidates a specified line from the specified MB86930 cache.

RETURNS

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

SEE ALSO

cacheMb930Lib

C

cacheMb930LibInit( )

NAME **cacheMb930LibInit( )** – initialize the Fujitsu MB86930 cache library

SYNOPSIS

```
STATUS cacheMb930LibInit
    (
    CACHE_MODE instMode,      /* instruction cache mode */
    CACHE_MODE dataMode       /* data cache mode */
    )
```

DESCRIPTION This routine installs the function pointers for the Fujitsu MB86930 cache library and performs other necessary cache library initialization. The board support package selects this cache library by setting the function pointer **sysCacheLibInit** equal to **cacheMb930LibInit( )**. Note that **sysCacheLibInit** must be initialized on declaration, placing it in the ".data" section.

This routine invalidates the cache tags and leaves the cache disabled. It should only be called during initialization, before any cache locking has taken place.

The only available mode for the MB86930 is **CACHE_WRITETHROUGH**.

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

SEE ALSO **cacheMb930Lib**

cacheMb930LockAuto( )

NAME **cacheMb930LockAuto( )** – enable MB86930 automatic locking of kernel instructions/data

SYNOPSIS

```
void cacheMb930LockAuto (void)
```

DESCRIPTION This routine enables automatic cache locking of kernel instructions and data into MB86930 caches. Once entries are locked into the caches, they cannot be unlocked.

RETURNS N/A

SEE ALSO **cacheMb930Lib**

cachePipeFlush( )

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

SYNOPSIS

```
STATUS cachePipeFlush (void)
```

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

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

SEE ALSO **cacheLib**

cacheR3kLibInit( )

NAME **cacheR3kLibInit( )** – initialize the R3000 cache library

SYNOPSIS

```
STATUS cacheR3kLibInit
    (
    CACHE_MODE instMode,       /* instruction cache mode */
    CACHE_MODE dataMode        /* data cache mode */
    )
```

DESCRIPTION This routine initializes the function pointers for the R3000 cache library. The board support package can select this cache library by calling this routine.

RETURNS **OK**.

SEE ALSO **cacheR3kLib**

C

cacheR4kLibInit( )

NAME

cacheR4kLibInit( ) – initialize the R4000 cache library

SYNOPSIS

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

DESCRIPTION

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

RETURNS

OK.

SEE ALSO

cacheR4kLib

cacheR5kLibInit( )

NAME

cacheR5kLibInit( ) – initialize the R5000 cache library

SYNOPSIS

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

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

RETURNS OK.

SEE ALSO **cacheR5kLib**

cacheR7kLibInit( )

NAME **cacheR7kLibInit( )** – initialize the R7000 cache library

SYNOPSIS

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

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

RETURNS OK.

SEE ALSO **cacheR7kLib**

cacheR10kLibInit( )

NAME

cacheR10kLibInit( ) – initialize the R10000 cache library

SYNOPSIS

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

DESCRIPTION

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

RETURNS

OK.

SEE ALSO

cacheR10kLib

cacheR32kLibInit( )

NAME

cacheR32kLibInit( ) – initialize the RC32364 cache library

SYNOPSIS

```
STATUS cacheR32kLibInit
    (
    CACHE_MODE instMode,      /* instruction cache mode */
    CACHE_MODE dataMode       /* data cache mode */
    )
```

DESCRIPTION

This routine initializes the function pointers for the RC32364 cache library. The board support package can select this cache library by assigning the function pointer *sysCacheLibInit* to **cacheR32kLibInit( )**.

This routine determines the cache size and cache line size for the instruction and data cache automatically by reading the CP0 configuration register. This is different than most of the other cache library initialization calls, which take the cache and line sizes as parameters.

RETURNS
OK.

SEE ALSO
cacheR32kLib

cacheR32kMalloc( )

NAME
cacheR32kMalloc( ) – allocate a cache-safe buffer, if possible

SYNOPSIS
```
void * cacheR32kMalloc
    (
    size_t bytes
    )
```

DESCRIPTION
This routine will attempt to return a pointer to a section of memory that will not experience any cache coherency problems.

RETURNS
A pointer to the non-cached buffer, or **NULL**.

SEE ALSO
cacheR32kLib

cacheR33kLibInit( )

NAME
cacheR33kLibInit( ) – initialize the R33000 cache library

SYNOPSIS
```
STATUS cacheR33kLibInit
    (
    CACHE_MODE instMode,      /* instruction cache mode */
    CACHE_MODE dataMode       /* data cache mode */
    )
```

DESCRIPTION
This routine initializes the function pointers for the R33000 cache library. The board support package can select this cache library by calling this routine.

RETURNS
OK.

SEE ALSO
cacheR33kLib

C

cacheR333x0LibInit( )

NAME **cacheR333x0LibInit( )** – initialize the R333x0 cache library

SYNOPSIS

```
STATUS cacheR333x0LibInit
    (
    CACHE_MODE instMode,       /* instruction cache mode */
    CACHE_MODE dataMode        /* data cache mode */
    )
```

DESCRIPTION This routine initializes the function pointers for the R333x0 cache library. The board support package can select this cache library by calling this routine.

RETURNS **OK**.

SEE ALSO **cacheR333x0Lib**

cacheSh7040LibInit( )

NAME **cacheSh7040LibInit( )** – initialize the SH7040 cache library

SYNOPSIS

```
STATUS cacheSh7040LibInit
    (
    CACHE_MODE instMode,       /* instruction cache mode */
    CACHE_MODE dataMode        /* data cache mode (ignored) */
    )
```

DESCRIPTION This routine initializes the cache library for the Hitachi SH7040 processors. It initializes the function pointers and configures the caches to the specified cache modes. Modes should be set before caching is enabled. If two complementary flags are set (enable/disable), no action is taken for any of the input flags.

Next caching modes are available for the SH7040 processors:

SH7040:	**CACHE_WRITETHROUGH**	(cache for instruction)
	CACHE_SH7040_DRAM	(enable caching for DRAM space)
	CACHE_SH7040_CS3	(enable caching for CS3 space)
	CACHE_SH7040_CS2	(enable caching for CS2 space)
	CACHE_SH7040_CS1	(enable caching for CS1 space)
	CACHE_SH7040_CS0	(enable caching for CS0 space)

RETURNS **OK**, or **ERROR** if the specified caching modes were invalid.

SEE ALSO **cacheSh7040Lib**

cacheSh7604LibInit( )

NAME **cacheSh7604LibInit( )** – initialize the SH7604/SH7615 cache library

SYNOPSIS

```
STATUS cacheSh7604LibInit
    (
    CACHE_MODE instMode,      /* instruction cache mode (ignored) */
    CACHE_MODE dataMode       /* data cache mode */
    )
```

DESCRIPTION This routine initializes the cache library for the Hitachi SH7604/SH7615 processor. It initializes the function pointers and configures the caches to the specified cache modes. Modes should be set before caching is enabled.

The following caching modes are available for the SH7604/SH7615 processor:

SH7604:	**CACHE_WRITETHROUGH**	(cache for instruction and data)
	CACHE_2WAY_MODE	(2KB 2-way cache + 2KB RAM)

RETURNS **OK**, or **ERROR** if the specified caching modes were invalid.

SEE ALSO **cacheSh7604Lib**

cacheSh7622LibInit( )

NAME **cacheSh7622LibInit( )** – initialize the SH7622 cache library

SYNOPSIS

```
STATUS cacheSh7622LibInit
    (
    CACHE_MODE instMode,      /* instruction cache mode */
    CACHE_MODE dataMode       /* data cache mode */
    )
```

DESCRIPTION This routine initializes the cache library for the Hitachi SH7622 processor. It initializes the function pointers and configures the caches to the specified cache modes. Modes should be set before caching is enabled. If two complementary flags are set (enable/disable), no action is taken for any of the input flags. Data cache and instruction cache are mixed together in the SH7622.

Next caching modes are available for the SH7622 processor:

SH7622:	**CACHE_WRITETHROUGH**	(cache for instruction and data)
	CACHE_COPYBACK_P1	(write-back cache for P1)

RETURNS

OK, or **ERROR** if the specified caching modes were invalid.

SEE ALSO

cacheSh7622Lib

cacheSh7700LibInit( )

NAME

cacheSh7700LibInit( ) – initialize the SH7700 cache library

SYNOPSIS

```
STATUS cacheSh7700LibInit
    (
    CACHE_MODE instMode,      /* instruction cache mode (ignored) */
    CACHE_MODE dataMode       /* data cache mode */
    )
```

DESCRIPTION

This routine initializes the cache library for the Hitachi SH7700 processor. It initializes the function pointers and configures the caches to the specified cache modes. Modes should be set before caching is enabled. If two complementary flags are set (enable/disable), no action is taken for any of the input flags.

The following caching modes are available for the SH7700 processor:

SH7700:	**CACHE_WRITETHROUGH**	(cache for instruction and data)
	CACHE_COPYBACK	(cache for instruction and data)
	CACHE_COPYBACK_P1	(copy-back cache for P1, SH7709 only)
	CACHE_2WAY_MODE	(4KB 2-way cache + 4KB RAM)
	CACHE_1WAY_MODE	(2KB direct mapped cache, SH7702 only)
	CACHE_DMA_BYPASS_P0	(allocate DMA buffer to P2, free it to P0)
	CACHE_DMA_BYPASS_P1	(allocate DMA buffer to P2, free it to P1)
	CACHE_DMA_BYPASS_P3	(allocate DMA buffer to P2, free it to P3)

The **CACHE_DMA_BYPASS_Px** modes allow to allocate "cache-safe" buffers without MMU. If none of **CACHE_DMA_BYPASS_Px** modes is specified, **cacheDmaMalloc( )** returns a cache-safe buffer on logical space, which is created by the MMU. If **CACHE_DMA_BYPASS_P0** is selected, **cacheDmaMalloc( )** returns a cache-safe buffer on

P2 space, and **cacheDmaFree( )** releases the buffer to P0 space. Namely, if the system memory partition is located on P0, cache-safe buffers can be allocated and freed without MMU, by selecting **CACHE_DMA_BYPASS_P0**.

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

SEE ALSO **cacheSh7700Lib**

cacheSh7729LibInit( )

NAME **cacheSh7729LibInit( )** – initialize the SH7729 cache library

SYNOPSIS

```
STATUS cacheSh7729LibInit
    (
    CACHE_MODE instMode,      /* instruction cache mode (ignored) */
    CACHE_MODE dataMode       /* data cache mode */
    )
```

DESCRIPTION This routine initializes the cache library for the Hitachi SH7729 processor. It initializes the function pointers and configures the caches to the specified cache modes. Modes should be set before caching is enabled. If two complementary flags are set (enable/disable), no action is taken for any of the input flags.

The following caching modes are available for the SH7729 processor:

SH7729:	**CACHE_WRITETHROUGH**	(cache for instruction and data)
	CACHE_COPYBACK	(cache for instruction and data)
	CACHE_COPYBACK_P1	(copy-back cache for P1)
	CACHE_DMA_BYPASS_P0	(allocate DMA buffer to P2, free it to P0)
	CACHE_DMA_BYPASS_P1	(allocate DMA buffer to P2, free it to P1)
	CACHE_DMA_BYPASS_P3	(allocate DMA buffer to P2, free it to P3)

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

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

SEE ALSO **cacheSh7729Lib**

C

cacheSh7750LibInit( )

NAME

cacheSh7750LibInit( ) – initialize the SH7750 cache library

SYNOPSIS

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

DESCRIPTION

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

The following caching modes are available for the SH7750 processor:

SH7750:	**CACHE_WRITETHROUGH**	
	CACHE_COPYBACK	(copy-back cache for P0/P3, data cache only)
	CACHE_COPYBACK_P1	(copy-back cache for P1, data cache only)
	CACHE_RAM_MODE	(use half of cache as RAM, data cache only)
	CACHE_2WAY_MODE	(use RAM in 2way associ. mode, data cache only)
	CACHE_A25_INDEX	(use A25 as MSB of cache index)
	CACHE_DMA_BYPASS_P0	(allocate DMA buffer to P2, free it to P0)
	CACHE_DMA_BYPASS_P1	(allocate DMA buffer to P2, free it to P1)
	CACHE_DMA_BYPASS_P3	(allocate DMA buffer to P2, free it to P3)

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

RETURNS

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

SEE ALSO

cacheSh7750Lib

cacheStoreBufDisable( )

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

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

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

RETURNS
N/A

SEE ALSO
cacheArchLib

cacheStoreBufEnable( )

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

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

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

RETURNS
N/A

SEE ALSO
cacheArchLib

cacheSun4ClearContext( )

NAME
cacheSun4ClearContext( ) – clear a specific context from a Sun-4 cache

SYNOPSIS
```
STATUS cacheSun4ClearContext
    (
    CACHE_TYPE cache,         /* cache to clear */
    void *     address        /* virtual address */
    )
```

DESCRIPTION This routine flushes and invalidates a specified context from the specified Sun-4 cache.

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

SEE ALSO **cacheSun4Lib**

cacheSun4ClearLine( )

NAME **cacheSun4ClearLine( )** – clear a line from a Sun-4 cache

SYNOPSIS

```
STATUS cacheSun4ClearLine
    (
    CACHE_TYPE cache,         /* cache to clear */
    void *     address        /* virtual address */
    )
```

DESCRIPTION This routine flushes and invalidates a specified line from the specified Sun-4 cache.

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

SEE ALSO **cacheSun4Lib**

cacheSun4ClearPage( )

NAME **cacheSun4ClearPage( )** – clear a page from a Sun-4 cache

SYNOPSIS

```
STATUS cacheSun4ClearPage
    (
    CACHE_TYPE cache,         /* cache to clear */
    void *     address        /* virtual address */
    )
```

DESCRIPTION This routine flushes and invalidates a specified page from the specified Sun-4 cache.

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

SEE ALSO **cacheSun4Lib**

cacheSun4ClearSegment( )

NAME

cacheSun4ClearSegment( ) – clear a segment from a Sun-4 cache

SYNOPSIS

```
STATUS cacheSun4ClearSegment
    (
    CACHE_TYPE cache,         /* cache to clear */
    void *     address        /* virtual address */
    )
```

DESCRIPTION

This routine flushes and invalidates a specified segment from the specified Sun-4 cache.

RETURNS

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

SEE ALSO

cacheSun4Lib

cacheSun4LibInit( )

NAME

cacheSun4LibInit( ) – initialize the Sun-4 cache library

SYNOPSIS

```
STATUS cacheSun4LibInit
    (
    CACHE_MODE instMode,      /* instruction cache mode */
    CACHE_MODE dataMode       /* data cache mode */
    )
```

DESCRIPTION

This routine initializes the function pointers for the Sun Microsystems Sun-4 cache library. The board support package can select this cache library by assigning the function pointer **sysCacheLibInit** to **cacheSun4LibInit( )**.

The only available mode for the Sun-4 cache is **CACHE_WRITETHROUGH**.

RETURNS

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

SEE ALSO

cacheSun4Lib

cacheTextUpdate( )

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

SYNOPSIS

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

DESCRIPTION This routine flushes the data cache, then invalidates the instruction cache. This operation forces the instruction cache to fetch code that may have been created via the data path.

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

SEE ALSO **cacheLib**

cacheTiTms390LibInit( )

NAME **cacheTiTms390LibInit( )** – initialize the TI TMS390 cache library

SYNOPSIS

```
STATUS cacheTiTms390LibInit
    (
    CACHE_MODE instMode,      /* instruction cache mode */
    CACHE_MODE dataMode       /* data cache mode */
    )
```

DESCRIPTION This routine initializes the function pointers for the TI TMS390 cache library. The board support package can select this cache library by assigning the function pointer **sysCacheLibInit** to **cacheTiTms390LibInit( )**.

The only available cache mode is **CACHE_COPYBACK**.

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

SEE ALSO **cacheTiTms390Lib**

cacheTiTms390PhysToVirt( )

NAME

cacheTiTms390PhysToVirt( ) – translate a physical address for drivers

SYNOPSIS

```
void * cacheTiTms390PhysToVirt
    (
    void * address              /* physical address */
    )
```

DESCRIPTION

This routine performs a 32-bit physical to 32-bit virtual address translation in the current context.

It works for only DRAM addresses of the first EMC.

It guesses likely virtual addresses, and checks its guesses with **VM_TRANSLATE**. A likely virtual address is the same as the physical address, or some multiple of 16M less. If any match, it succeeds. If all guesses are wrong, it fails.

RETURNS

The virtual address that maps to the physical address bits [31:0] argument, or **NULL** if it fails.

SEE ALSO

cacheTiTms390Lib

cacheTiTms390VirtToPhys( )

NAME

cacheTiTms390VirtToPhys( ) – translate a virtual address for **cacheLib**

SYNOPSIS

```
void * cacheTiTms390VirtToPhys
    (
    void * address              /* virtual address */
    )
```

DESCRIPTION

This routine performs a 32-bit virtual to 32-bit physical address translation in the current context.

RETURNS

The physical address translation bits [31:0] of a virtual address argument, or **NULL** if the virtual address is not valid, or the physical address does not fit in 32 bits.

RETURNS

N/A

SEE ALSO

cacheTiTms390Lib

cacheTx49LibInit( )

NAME
cacheTx49LibInit( ) – initialize the Tx49 cache library

SYNOPSIS

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

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

RETURNS
OK.

SEE ALSO
cacheTx49Lib

cacheUnlock( )

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

SYNOPSIS

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

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

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

SEE ALSO
cacheLib

calloc( )

NAME

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

SYNOPSIS

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

DESCRIPTION

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

RETURNS

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

SEE ALSO

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

cbioBlkCopy( )

NAME

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

SYNOPSIS

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

DESCRIPTION

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

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

RETURNS

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

SEE ALSO

cbioLib

cbioBlkRW( )

NAME

cbioBlkRW( ) – transfer blocks to or from memory

SYNOPSIS

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

DESCRIPTION

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

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

RETURNS

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

SEE ALSO

cbioLib

cbioBytesRW( )

NAME

cbioBytesRW( ) – transfer bytes to or from memory

SYNOPSIS

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

DESCRIPTION

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

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

RETURNS

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

SEE ALSO

cbioLib

cbioDevCreate( )

NAME

cbioDevCreate( ) – initialize a CBIO device (Generic)

SYNOPSIS

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

DESCRIPTION

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

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

RETURNS

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

SEE ALSO

cbioLib

cbioDevVerify( )

NAME

cbioDevVerify( ) – verify **CBIO_DEV_ID**

SYNOPSIS

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

C

DESCRIPTION

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

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

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

RETURNS

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

SEE ALSO

cbioLib, **dosFsLib**, **dcacheCbio**, **dpartCbio**

cbioIoctl( )

NAME

cbioIoctl( ) – perform ioctl operation on device

SYNOPSIS

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

DESCRIPTION

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

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

– **CBIO_RESET** - reset the CBIO device. When the third argument to the ioctl call accompaning **CBIO_RESET** is **NULL**, the code verifies that the disk is inserted and is ready, after getting it to a known state. When the 3rd argument is a non-zero, it is assumed to be a **BLK_DEV** pointer and **CBIO_RESET** will install a new subordinate block device. This work is performed at the **BLK_DEV** to CBIO layer, and all layers shall account for it. A **CBIO_RESET** indicates a possible change in device geometry, and the **CBIO_PARAMS** members will be reinitialized after a **CBIO_RESET**.

– **CBIO_STATUS_CHK** - check device status of CBIO device and lower layer

– **CBIO_DEVICE_LOCK** - Prevent disk removal

– **CBIO_DEVICE_UNLOCK** - Allow disk removal

– **CBIO_DEVICE_EJECT** - Unmount and eject device

– **CBIO_CACHE_FLUSH** - Flush any dirty cached data

– **CBIO_CACHE_INVAL** - Flush & Invalidate all cached data

– **CBIO_CACHE_NEWBLK** - Allocate scratch block

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

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

SEE ALSO
cbioLib

cbioLibInit( )

NAME
cbioLibInit( ) – Initialize CBIO Library

SYNOPSIS

```
STATUS cbioLibInit (void)
```

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

RETURNS
OK or **ERROR**

SEE ALSO
cbioLib

cbioLock( )

NAME
cbioLock( ) – obtain CBIO device semaphore.

SYNOPSIS

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

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

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

SEE ALSO **cbioLib**

---

cbioModeGet( )

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

SYNOPSIS

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

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

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

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

SEE ALSO **cbioLib**

---

cbioModeSet( )

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

SYNOPSIS

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

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

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

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

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

SEE ALSO **cbioLib**

cbioParamsGet( )

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

SYNOPSIS

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

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

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

SEE ALSO **cbioLib**

cbioRdyChgdGet( )

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

SYNOPSIS

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

DESCRIPTION For example

C

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

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

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

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

SEE ALSO **cbioLib**

---

cbioRdyChgdSet( )

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

SYNOPSIS

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

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

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

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

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

SEE ALSO **cbioLib**

cbioShow( )

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

SYNOPSIS

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

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

It takes two arguments:

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

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

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

cbioUnlock( )

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

SYNOPSIS

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

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

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

SEE ALSO **cbioLib**

C

cbioWrapBlkDev( )

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

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

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

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

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

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

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

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

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

cbrt( )

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

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

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

INCLUDE FILES **math.h**

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

SEE ALSO **mathALib**

cbrtf( )

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

SYNOPSIS

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

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

INCLUDE FILES **math.h**

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

SEE ALSO **mathALib**

cd( )

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

SYNOPSIS

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

DESCRIPTION

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 Tornado Shell (**windsh**), which has a built-in command of the same name that operates on the Host's I/O system.

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

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**.

SEE ALSO

usrFsLib, **pwd( )**, *VxWorks Programmer's Guide: Target Shell*

cdromFsDevCreate( )

NAME **cdromFsDevCreate( )** – create a **cdromFsLib** device

SYNOPSIS

```
CDROM_VOL_DESC_ID cdromFsDevCreate
    (
    char *    devName,        /* device name */
    BLK_DEV * pBlkDev         /* ptr to block device */
    )
```

DESCRIPTION This routine creates an instance of a **cdromFsLib** device in the I/O system. As input, this function requires a pointer to a **BLK_DEV** structure for the CD-ROM drive on which you want to create a **cdromFsLib** device. Thus, you should already have called **scsiBlkDevCreate( )** prior to calling **cdfromFsDevCreate( )**.

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

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

cdromFsInit( )

NAME **cdromFsInit( )** – initialize **cdromFsLib**

SYNOPSIS

```
STATUS cdromFsInit (void)
```

DESCRIPTION This routine initializes **cdromFsLib**. It must be called exactly once before calling any other routine in **cdromFsLib**.

ERRNO **S_cdromFsLib_ALREADY_INIT**

RETURNS **OK** or **ERROR**, if **cdromFsLib** has already been initialized.

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

C

cdromFsVolConfigShow( )

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

SYNOPSIS

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

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

RETURNS N/A

SEE ALSO **cdromFsLib**

ceil( )

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

SYNOPSIS

```
double ceil
    (
    double v                  /* value to find the ceiling of */
    )
```

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

INCLUDE FILES **math.h**

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

SEE ALSO **ansiMath**, **mathALib**

ceilf( )

NAME

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

SYNOPSIS

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

DESCRIPTION

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

INCLUDE FILES

math.h

RETURNS

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

SEE ALSO

mathALib

cfree( )

NAME

cfree( ) – free a block of memory

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.

SEE ALSO

memLib

chdir( )

NAME **chdir( )** – set the current default path

SYNOPSIS

```
STATUS chdir
    (
    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.

SEE ALSO **ioLib**, **ioDefPathSet( )**, **ioDefPathGet( )**, **getcwd( )**

checkStack( )

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

SYNOPSIS

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

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

```
-> checkStack tShell
    NAME         ENTRY       TID     SIZE   CUR  HIGH  MARGIN
------------ ------------ -------- ----- ----- ----- ------
tShell       _shell       23e1c78   9208   832  3632   5576
```

The maximum stack usage is determined by scanning down from the top of the stack for the first byte whose value is not 0xee. In VxWorks, when a task is spawned, all bytes of a task's stack are initialized to 0xee.

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

RETURNS
N/A

SEE ALSO
usrLib, **taskSpawn( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

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 file system will be blocked from application code access, and will emit messages describing any inconsistencies found on the disk, as well as some statistics, depending on the value of the *verbose* argument. Depending 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.

SEE ALSO **usrFsLib**

cleanUpStoreBuffer( )

NAME **cleanUpStoreBuffer( )** – clean up store buffer after a data store error interrupt

SYNOPSIS

```
void cleanUpStoreBuffer
    (
    UINT mcntl,              /* Value of MMU Control Register */
    BOOL exception           /* TRUE if exception, FALSE if int */
    )
```

DESCRIPTION This routine cleans up the store buffer after a data store error interrupt. The first queued store is retried. It is logged as either a recoverable or unrecoverable error. Then the store buffer is re-enabled and other queued stores are processed by the store buffer.

RETURNS N/A

SEE ALSO **cacheTiTms390Lib**

clearerr( )

NAME **clearerr( )** – clear end-of-file and error flags for a stream (ANSI)

SYNOPSIS

```
void clearerr
    (
    FILE * fp                /* stream to clear EOF and ERROR flags for */
    )
```

DESCRIPTION This routine clears the end-of-file and error flags for a specified stream.

INCLUDE FILES **stdio.h**

RETURNS N/A

SEE ALSO **ansiStdio**, **feof( )**, **ferror( )**

clock( )

NAME
clock( ) – determine the processor time in use (ANSI)

SYNOPSIS

```
clock_t clock (void)
```

DESCRIPTION
This routine returns the implementation's best approximation of the processor time used by the program since the beginning of an implementation-defined era related only to the program invocation. To determine the time in seconds, the value returned by **clock( )** should be divided by the value of the macro **CLOCKS_PER_SEC**. If the processor time used is not available or its value cannot be represented, **clock( )** returns -1.

NOTE: This routine always returns -1 in VxWorks. VxWorks does not track per-task time or system idle time. There is no method of determining how long a task or the entire system has been doing work. **tickGet( )** can be used to query the number of system ticks since system start. **clock_gettime( )** can be used to get the current clock time.

INCLUDE FILES
time.h

RETURNS
-1

SEE ALSO
ansiTime, **tickGet( )**, **clock_gettime( )**

clock_getres( )

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

SYNOPSIS

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

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

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

ERRNO
EINVAL

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

clock_gettime( )

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

SYNOPSIS

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

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

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

ERRNO **EINVAL**, **EFAULT**

SEE ALSO **clockLib**

clock_setres( )

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

SYNOPSIS

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

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

NOTE: Non-POSIX.

RETURNS **OK** always.

ERRNO EINVAL

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

clock_settime( )

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

SYNOPSIS

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

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

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

ERRNO **EINVAL**

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

close( )

NAME **close( )** – close a file

SYNOPSIS

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

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

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

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**.

SEE ALSO

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

connect( )

NAME

connect( ) – initiate a connection to a socket

SYNOPSIS

```
STATUS connect
    (
    int               s,      /* socket descriptor */
    struct sockaddr * name,   /* addr of socket to connect */
    int               namelen /* length of name, in bytes */
    )
```

DESCRIPTION

If *s* is a socket of type **SOCK_STREAM**, this routine establishes a virtual circuit between *s* and another socket specified by *name*. If *s* is of type **SOCK_DGRAM**, it permanently specifies the peer to which messages are sent. If *s* is of type **SOCK_RAW**, it specifies the raw socket upon which data is to be sent and received. The *name* parameter specifies the address of the other socket.

NOTE: If a socket with type **SOCK_STREAM** is marked non-blocking, this routine will return **ERROR** with an error number of **EINPROGRESS** or **EALREADY** if a connection attempt is pending. A later call will return **ERROR** and set the error number to **EISCONN** once the connection is established. The connection attempt must be repeated until that result occurs or until this routine establishes a connection immediately and returns **OK**.

RETURNS

OK, or **ERROR** if the connection attempt does not complete.

SEE ALSO

sockLib

connectWithTimeout( )

NAME **connectWithTimeout( )** – attempt socket connection within a specified duration

SYNOPSIS

```
STATUS connectWithTimeout
    (
    int                sock,    /* socket descriptor */
    struct sockaddr * adrs,    /* addr of the socket to connect */
    int                adrsLen, /* length of the socket, in bytes */
    struct timeval *  timeVal  /* time-out value */
    )
```

DESCRIPTION Use this routine as an alternative to **connect( )** when your application requires a shorter time out on a connection attempt. By design, a TCP connection attempt times out after 75 seconds if unsuccessful. Thus, a blocking TCP socket **connect( )** call might not return for 75 seconds. A **connectWithTimeout( )** call lets you reduce this time out by scheduling an abort of the connection attempt if it is not successful before *timeVal*. However, **connectWithTimeout( )** does not actually change the TCP timeout value. Thus, you cannot use **connectWithTimeout( )** to lengthen the connection time out beyond the TCP default.

In all respects other than the time out value, a **connectWithTimeout( )** call behaves exactly like **connect( )**. Thus, if no application is listening for connections at the other end, **connectWithTimeout( )** returns immediately just like **connect( )**. If you specify a **NULL** pointer for *timeVal*, **connectWithTimeout( )** behaves exactly like a **connect( )** call.

RETURNS **OK**, or **ERROR** if a new connection is not established before timeout.

SEE ALSO **sockLib**, **connect( )**

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 CTRL+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*.

SEE ALSO
usrFsLib, **copyStreams( )**, **tyEOFSet( )**, **cp( )**, **xcopy( )**, *VxWorks Programmer's Guide: Target Shell*

---

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*.

SEE ALSO
usrFsLib, **copy( )**, *VxWorks Programmer's Guide: Target Shell*

cos( )

NAME **cos( )** – compute a cosine (ANSI)

SYNOPSIS

```
double cos
    (
    double x                  /* angle in radians */
    )
```

DESCRIPTION This routine computes the cosine of *x* in double precision. The angle *x* is expressed in radians.

INCLUDE FILES **math.h**

RETURNS The double-precision cosine of *x*.

SEE ALSO **ansiMath**, **mathALib**

cosf( )

NAME **cosf( )** – compute a cosine (ANSI)

SYNOPSIS

```
float cosf
    (
    float x   /* angle in radians */
    )
```

DESCRIPTION This routine returns the cosine of *x* in single precision. The angle *x* is expressed in radians.

INCLUDE FILES **math.h**

RETURNS The single-precision cosine of *x*.

SEE ALSO **mathALib**

cosh( )

NAME **cosh( )** – compute a hyperbolic cosine (ANSI)

SYNOPSIS

```
double cosh
    (
    double x                  /* value to compute the hyperbolic cosine of */
    )
```

DESCRIPTION This routine returns the hyperbolic cosine of *x* in double precision (IEEE double, 53 bits).

A range error occurs if *x* is too large.

INCLUDE FILES math.h

RETURNS The double-precision hyperbolic cosine of *x*.

Special cases:
If *x* is +INF, -INF, or NaN, **cosh( )** returns *x*.

SEE ALSO **ansiMath**, **mathALib**

coshf( )

NAME **coshf( )** – compute a hyperbolic cosine (ANSI)

SYNOPSIS

```
float coshf
    (
    float x   /* value to compute the hyperbolic cosine of */
    )
```

DESCRIPTION This routine returns the hyperbolic cosine of *x* in single precision.

INCLUDE FILES **math.h**

RETURNS The single-precision hyperbolic cosine of *x* if the parameter is greater than 1.0, or NaN if the parameter is less than 1.0.

Special cases:
If *x* is +INF, -INF, or NaN, **coshf( )** returns *x*.

SEE ALSO **mathALib**

cp( )

NAME
cp( ) – copy file into other file/directory.

SYNOPSIS

```
STATUS cp
    (
    const char * src,         /* source file or wildcard pattern */
    const char * dest         /* destination file name or directory */
    )
```

DESCRIPTION
This command copies from the input file to the output file. If destination name is directory, a source file is copied into this directory, using the last element of the source file name to be the name of the destination file.

This function is very similar to **copy( )**, except it is somewhat more similar to the UNIX "cp" program in its handling of the destination.

src may contain a wildcard pattern, in which case all files matching the pattern will be copied to the directory specified in *dest*. This function does not copy directories, and is not recursive. To copy entire subdirectories recursively, use **xcopy( )**.

EXAMPLES

```
-> cp( "/sd0/FILE1.DAT","/sd0/dir2/f001.dat")
-> cp( "/sd0/dir1/file88","/sd0/dir2")
-> cp( "/sd0/*.tmp","/sd0/junkdir")
```

RETURNS
OK or **ERROR** if destination is not a directory while *src* is a wildcard pattern, or if any of the files could not be copied.

SEE ALSO
xcopy( )

SEE ALSO
usrFsLib

cplusCallNewHandler( )

NAME
cplusCallNewHandler( ) – call the allocation failure handler (C++)

SYNOPSIS

```
extern void cplusCallNewHandler ()
```

DESCRIPTION
This function provides a procedural-interface to the new-handler. It can be used by user-defined new operators to call the current new-handler. This function is specific to VxWorks and may not be available in other C++ environments.

RETURNS N/A

SEE ALSO **cplusLib**

cplusCtors( )

NAME **cplusCtors( )** – call static constructors (C++)

SYNOPSIS

```
extern "C" void cplusCtors
    (
    const char * moduleName   /* name of loaded module */
    )
```

DESCRIPTION This function is used to call static constructors under the manual strategy (see **cplusXtorSet( )**). *moduleName* is the name of an object module that was "munched" before loading. If *moduleName* is 0, then all static constructors, in all modules loaded by the VxWorks module loader, are called.

EXAMPLES The following example shows how to initialize the static objects in modules called "applx.out" and "apply.out".

```
-> cplusCtors "applx.out"
value = 0 = 0x0
-> cplusCtors "apply.out"
value = 0 = 0x0
```

The following example shows how to initialize all the static objects that are currently loaded, with a single invocation of **cplusCtors( )**:

```
-> cplusCtors
value = 0 = 0x0
```

WARNING: **cplusCtors( )** should only be called once per module otherwise unpredictable behavior may result.

RETURNS N/A

SEE ALSO **cplusLib**, **cplusXtorSet( )**

cplusCtorsLink( )

NAME

cplusCtorsLink( ) – call all linked static constructors (C++)

SYNOPSIS

```
extern "C" void cplusCtorsLink ()
```

DESCRIPTION

This function calls constructors for all of the static objects linked with a VxWorks bootable image. When creating bootable applications, this function should be called from **usrRoot( )** to initialize all static objects. Correct operation depends on correctly munching the C++ modules that are linked with VxWorks.

RETURNS

N/A

SEE ALSO

cplusLib

cplusDemanglerSet( )

NAME

cplusDemanglerSet( ) – change C++ demangling mode (C++)

SYNOPSIS

```
extern "C" void cplusDemanglerSet
    (
    int mode
    )
```

DESCRIPTION

This command sets the C++ demangling mode to *mode*. The default mode is 2.

There are three demangling modes, *complete*, *terse*, and *off*. These modes are represented by numeric codes:

Mode	Code
off	0
terse	1
complete	2

In complete mode, when C++ function names are printed, the class name (if any) is prefixed and the function's parameter type list is appended.

In terse mode, only the function name is printed. The class name and parameter type list are omitted.

In off mode, the function name is not demangled.

EXAMPLES

The following example shows how one function name would be printed under each demangling mode:

Mode	Printed symbol
off	**_member__5classFPFl_PvPFPv_v**
terse	**_member**
complete	**foo::_member(void* (*)(long),void (*)(void*))**

RETURNS

N/A

SEE ALSO

cplusLib

cplusDemanglerStyleSet( )

NAME

cplusDemanglerStyleSet( ) – change C++ demangling style (C++)

SYNOPSIS

```
extern "C" void cplusDemanglerStyleSet
    (
    DEMANGLER_STYLE style
    )
```

DESCRIPTION

This command sets the C++ demangling mode to *style*. The available demangler styles are enumerated in **demangler.h**. The default demangling style depends on the toolchain used to build the kernel. For example if the Diab toolchain is used to build the kernel then the default demangler style is **DMGL_STYLE_DIAB**.

RETURNS

N/A

SEE ALSO

cplusLib

cplusDtors( )

NAME

cplusDtors( ) – call static destructors (C++)

SYNOPSIS

```
extern "C" void cplusDtors
    (
    const char * moduleName
    )
```

DESCRIPTION

This function is used to call static destructors under the manual strategy (see **cplusXtorSet( )**). *moduleName* is the name of an object module that was "munched" before loading. If *moduleName* is 0, then all static destructors, in all modules loaded by the VxWorks module loader, are called.

EXAMPLES

The following example shows how to destroy the static objects in modules called "applx.out" and "apply.out":

```
-> cplusDtors "applx.out"
value = 0 = 0x0
-> cplusDtors "apply.out"
value = 0 = 0x0
```

The following example shows how to destroy all the static objects that are currently loaded, with a single invocation of **cplusDtors( )**:

```
-> cplusDtors
value = 0 = 0x0
```

WARNING: **cplusDtors( )** should only be called once per module otherwise unpredictable behavior may result.

RETURNS

N/A

SEE ALSO

cplusLib, **cplusXtorSet( )**

cplusDtorsLink( )

NAME

cplusDtorsLink( ) – call all linked static destructors (C++)

SYNOPSIS

```
extern "C" void cplusDtorsLink ()
```

DESCRIPTION

This function calls destructors for all of the static objects linked with a VxWorks bootable image. When creating bootable applications, this function should be called during system shutdown to decommission all static objects. Correct operation depends on correctly munching the C++ modules that are linked with VxWorks.

RETURNS

N/A

SEE ALSO

cplusLib

cplusLibInit( )

NAME **cplusLibInit( )** – initialize the C++ library (C++)

SYNOPSIS

```
extern "C" STATUS cplusLibInit (void)
```

DESCRIPTION This routine initializes the C++ library and forces all C++ run-time support to be linked with the bootable VxWorks image. If the configuration macro **INCLUDE_CPLUS** is defined, **cplusLibInit( )** is called automatically from the root task, **usrRoot( )**, in **usrConfig.c**.

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

SEE ALSO **cplusLib**

cplusXtorSet( )

NAME **cplusXtorSet( )** – change C++ static constructor calling strategy (C++)

SYNOPSIS

```
extern "C" void cplusXtorSet
    (
    int strategy
    )
```

DESCRIPTION This command sets the C++ static constructor calling strategy to *strategy*. The default strategy is 1.

There are two static constructor calling strategies: *automatic* and *manual*. These modes are represented by numeric codes:

Strategy	Code
manual	0
automatic	1

Under the manual strategy, a module's static constructors and destructors are called by **cplusCtors( )** and **cplusDtors( )**, which are themselves invoked manually.

Under the automatic strategy, a module's static constructors are called as a side-effect of loading the module using the VxWorks module loader. A module's static destructors are called as a side-effect of unloading the module.

NOTE: The manual strategy is applicable only to modules that are loaded by the VxWorks module loader. Static constructors and destructors contained by modules linked with the VxWorks image are called using **cplusCtorsLink( )** and **cplusDtorsLink( )**.

RETURNS N/A

SEE ALSO **cplusLib**

cpsr( )

NAME **cpsr( )** – return the contents of the current processor status register (ARM)

SYNOPSIS

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

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

RETURNS The contents of the current processor status register.

SEE ALSO **dbgArchLib**, *VxWorks Programmer's Guide: Debugging*

creat( )

NAME **creat( )** – create a file

SYNOPSIS

```
int creat
    (
    const char * name,        /* name of the file to create */
    int          flag         /* O_RDONLY, O_WRONLY, or O_RDWR */
    )
```

DESCRIPTION This routine creates a file called *name* and opens it with a specified *flag*. This routine determines on which device to create the file; it then calls the create routine of the device driver to do most of the work. Therefore, much of what transpires is device/driver-dependent.

The parameter *flag* is set to **O_RDONLY** (0), **O_WRONLY** (1), or **O_RDWR** (2) for the duration of time the file is open. To create NFS files with a UNIX chmod-type file mode, call **open( )** with the file mode specified in the third argument.

NOTE: For more information about situations when there are no file descriptors available, see the manual 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**.

SEE ALSO

ioLib, **open( )**

cret( )

NAME

cret( ) – continue until the current subroutine returns

SYNOPSIS

```
STATUS cret
    (
    int task                  /* task to continue, 0 = default */
    )
```

DESCRIPTION

This routine places a breakpoint at the return address of the current subroutine of a specified task, then continues execution of that task.

To execute, enter:

```
-> cret [task]
```

If *task* is omitted or zero, the last task referenced is assumed.

When the breakpoint is hit, information about the task will be printed in the same format as in single-stepping. The breakpoint is automatically removed when hit, or if the task hits another breakpoint first.

RETURNS

OK, or **ERROR** if there is no such task or the breakpoint table is full.

SEE ALSO

dbgLib, **so( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

ctime( )

NAME

ctime( ) – convert time in seconds into a string (ANSI)

SYNOPSIS

```
char * ctime
    (
    const time_t * timer      /* calendar time in seconds */
    )
```

DESCRIPTION

This routine converts the calendar time pointed to by *timer* into local time in the form of a string. It is equivalent to:

```
asctime (localtime (timer));
```

This routine is not reentrant. For a reentrant version, see **ctime_r( )**.

INCLUDE FILES

time.h

RETURNS

The pointer returned by **asctime( )** with local broken-down time as the argument.

SEE ALSO

ansiTime, **asctime( )**, **localtime( )**

ctime_r( )

NAME

ctime_r( ) – convert time in seconds into a string (POSIX)

SYNOPSIS

```
char * ctime_r
    (
    const time_t * timer,      /* calendar time in seconds */
    char *         asctimeBuf, /* buffer to contain the string */
    size_t *       buflen      /* size of the buffer */
    )
```

DESCRIPTION

This routine converts the calendar time pointed to by *timer* into local time in the form of a string. It is equivalent to:

```
asctime (localtime (timer));
```

This routine is the POSIX re-entrant version of **ctime( )**.

INCLUDE FILES

time.h

RETURNS The pointer returned by **asctime( )** with local broken-down time as the argument.

SEE ALSO **ansiTime**, **asctime( )**, **localtime( )**

d( )

NAME

d( ) – display memory

SYNOPSIS

```
void d
    (
    void * adrs,     /* address to display (if 0, display next block */
    int    nunits,   /* number of units to print (if 0, use default) */
    int    width     /* width of displaying unit (1, 2, 4, 8)        */
    )
```

DESCRIPTION

This command displays the contents of memory, starting at *adrs*. If *adrs* is omitted or zero, **d( )** displays the next memory block, starting from where the last **d( )** command completed.

Memory is displayed in units specified by *width*. If *nunits* is omitted or zero, the number of units displayed defaults to last use. If *nunits* is non-zero, that number of units is displayed and that number then becomes the default. If *width* is omitted or zero, it defaults to the previous value. If *width* is an invalid number, it is set to 1. The valid values for *width* are 1, 2, 4, and 8. The number of units **d( )** displays is rounded up to the nearest number of full lines.

RETURNS

N/A

SEE ALSO

usrLib, **m( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

d0( )

NAME

d0( ) – return the contents of register **d0** (also **d1** - **d7**) (68K)

SYNOPSIS

```
int d0
    (
    int taskId                /* task ID, 0 means default task */
    )
```

DESCRIPTION

This command extracts the contents of register **d0** from the TCB of a specified task. If *taskId* is omitted or zero, the last task referenced is assumed.

Similar routines are provided for all data registers (**d0** - **d7**): **d0( )** - **d7( )**.

RETURNS

The contents of register **d0** (or the requested register).

SEE ALSO

dbgArchLib, *VxWorks Programmer's Guide: Target Shell*

D

dbgBpTypeBind( )

NAME **dbgBpTypeBind( )** – bind a breakpoint handler to a breakpoint type (MIPS)

SYNOPSIS

```
STATUS dbgBpTypeBind
    (
    int     bpType,          /* breakpoint type */
    FUNCPTR routine          /* function to bind */
    )
```

DESCRIPTION Dynamically bind a breakpoint handler to breakpoints of type 0 - 7. By default only breakpoints of type zero are handled with the vxWorks breakpoint handler (see **dbgLib**). Other types may be used for Ada stack overflow or other such functions. The installed handler must take the same parameters as **excExcHandle( )** (see **excLib**).

RETURNS **OK**, or **ERROR** if *bpType* is out of bounds.

SEE ALSO **dbgArchLib**, **dbgLib**, **excLib**

dbgHelp( )

NAME **dbgHelp( )** – display debugging help menu

SYNOPSIS

```
void dbgHelp (void)
```

DESCRIPTION This routine displays a summary of **dbgLib** utilities with a short description of each, similar to the following:

```
dbgHelp                           Print this list
dbgInit                           Install debug facilities
b                                 Display breakpoints
b         addr[,task[,count]]     Set breakpoint
e         addr[,eventNo[,task[,func[,arg]]]]] Set eventpoint (WindView)
bd        addr[,task]             Delete breakpoint
bdall     [task]                  Delete all breakpoints
c         [task[,addr[,addr1]]]   Continue from breakpoint
cret      [task]                  Continue to subroutine return
s         [task[,addr[,addr1]]]   Single step
so        [task]                  Single step/step over subroutine
l         [adr[,nInst]]           List disassembled memory
tt        [task]                  Do stack trace on task
```

```
    bh        addr[,access[,task[,count[,quiet]]]] set hardware breakpoint
                                             (if supported by the architecture)
```

RETURNS N/A

SEE ALSO **dbgLib**, *VxWorks Programmer's Guide: Target Shell*

dbgInit( )

NAME **dbgInit( )** – initialize the local debugging package

SYNOPSIS

```
STATUS dbgInit (void)
```

DESCRIPTION This routine initializes the local debugging package and enables the basic breakpoint and single-step functions.

This routine also enables the shell abort function, CTRL-C.

NOTE: The debugging package should be initialized before any debugging routines are used. If the configuration macro **INCLUDE_DEBUG** is defined, **dbgInit( )** is called by the root task, **usrRoot( )**, in **usrConfig.c**.

RETURNS **OK**, always.

SEE ALSO **dbgLib**, *VxWorks Programmer's Guide: Target Shell*

dcacheDevCreate( )

NAME **dcacheDevCreate( )** – create a disk cache

SYNOPSIS

```
CBIO_DEV_ID dcacheDevCreate
    (
    CBIO_DEV_ID subDev,      /* block device handle */
    char *      pRamAddr,    /* where it is in memory (NULL = KHEAP_ALLOC) */
    int         memSize,     /* amount of memory to use */
    char *      pDesc        /* device description string */
    )
```

DESCRIPTION This routine creates a CBIO layer disk data cache instance. The disk cache unit accesses the disk through the subordinate CBIO device driver, provided with the *subDev* argument.

A valid block device **BLK_DEV** handle may be provided instead of a CBIO handle, in which case it will be automatically converted into a CBIO device by using the wrapper functionality from **cbioLib**.

Memory which will be used for caching disk data may be provided by the caller with *pRamAddr*, or it will be allocated by **dcacheDevCreate( )** from the common system memory pool, if *memAddr* is passed as **NULL**. *memSize* is the amount of memory to use for disk caching, if 0 is passed, then a certain default value will be calculated, based on available memory. *pDesc* is a string describing the device, used later by **dcacheShow( )**, and is useful when there are many cached disk devices.

A maximum of 16 disk cache devices are supported at this time.

RETURNS disk cache device handle, or **NULL** if there is not enough memory to satisfy the request, or the *blkDev* handle is invalid.

SEE ALSO **dcacheCbio**

dcacheDevDisable( )

NAME **dcacheDevDisable( )** – disable the disk cache for this device

SYNOPSIS

```
STATUS dcacheDevDisable
    (
    CBIO_DEV_ID dev           /* CBIO device handle */
    )
```

DESCRIPTION This function disables the cache by setting the bypass count to zero and storing the old value, if there is already an old value then we won't repeat the process though.

RETURNS **OK** if cache is sucessfully disabled or **ERROR**.

SEE ALSO **dcacheCbio**

dcacheDevEnable( )

NAME **dcacheDevEnable( )** – re-enable the disk cache

SYNOPSIS

```
STATUS dcacheDevEnable
    (
    CBIO_DEV_ID dev            /* CBIO device handle */
    )
```

DESCRIPTION This function re-enables the cache if we disabled it. If we did not disable it, then we cannot re-enable it.

RETURNS **OK** if cache is sucessfully enabled or **ERROR**.

SEE ALSO **dcacheCbio**

dcacheDevMemResize( )

NAME **dcacheDevMemResize( )** – set a new size to a disk cache device

SYNOPSIS

```
STATUS dcacheDevMemResize
    (
    CBIO_DEV_ID dev,           /* device handle */
    size_t      newSize        /* new cache size in bytes */
    )
```

DESCRIPTION This routine is used to resize the dcache layer. This routine is also useful after a disk change event, for example a PCMCIA disk swap. The routine **pccardDosDevCreate( )** in **pccardLib.c** uses this routine for that function. This should be invoked each time a new disk is inserted on media where the device geometry could possibly change. This function will re-read all device geometry data from the block driver, carve out and initialize all cache descriptors and blocks.

RETURNS **OK** or **ERROR** if the device is invalid or if the device geometry is invalid (**EINVAL**) or if there is not enough memory to perform the operation.

SEE ALSO **dcacheCbio**

D

dcacheDevTune( )

NAME

dcacheDevTune( ) – modify tunable disk cache parameters

SYNOPSIS

```
STATUS dcacheDevTune
    (
    CBIO_DEV_ID dev,          /* device handle */
    int         dirtyMax,     /* max # of dirty cache blocks allowed */
    int         bypassCount,  /* request size for bypassing cache */
    int         readAhead,    /* how many blocks to read ahead */
    int         syncInterval  /* how many seconds between disk updates */
    )
```

DESCRIPTION

This function allows the user to tune some disk cache parameters to obtain better performance for a given application or workload pattern. These parameters are checked for sanity before being used, hence it is recommended to verify the actual parameters being set with **dcacheShow( )**.

Following is the description of each tunable parameter:

bypassCount
In order to achieve maximum performance, Disk Cache is bypassed for very large requests. This parameter sets the threshold number of blocks for bypassing the cache, resulting usually in the data being transferred by the low level driver directly to/from application data buffers (also known as cut-through DMA). Passing the value of 0 in this argument preserves the previous value of the associated parameter.

syncInterval
The Disk Cache provides a low priority task that will update all modified blocks onto the disk periodically. This parameters controls the time between these updates in seconds. The longer this period, the better throughput is likely to be achieved, while risking to loose more data in the event of a failure. For removable devices this interval is fixed at 1 second. Setting this parameter to 0 results in immediate writes to disk when requested, resulting in minimal data loss risk at the cost of somewhat degraded performance.

readAhead
In order to avoid accessing the disk in small units, the Disk Cache will read many contiguous blocks once a block which is absent from the cache is needed. Increasing this value increases read performance, but a value which is too large may cause blocks which are frequently used to be removed from the cache, resulting in a low Hit Ratio, and increasing the number of Seeks, slowing down performance dramatically. Passing the value of 0 in this argument preserves the pervious value of the associated parameter.

dirtyMax
Routinely the Disk Cache will keep modified blocks in memory until it is specifically instructed to update these blocks to the disk, or until the specified time interval between disk updates has elapsed, or until the number of modified blocks is large enough to justify an update. Because the disk is updated in an ordered manner, and the blocks are written in groups when adjacent blocks have been modified, a larger *dirtyMax* parameter will minimize the number of Seek operation, but a value which is too large may decrease the Hit Ratio, thus degrading performance. Passing the value of 0 in this argument preserves the pervious value of the associated parameter.

RETURNS **OK** or **ERROR** if device handle is invalid. Parameter value which is out of range will be silently corrected.

SEE ALSO **dcacheCbio**, **dcacheShow( )**

dcacheHashTest( )

NAME **dcacheHashTest( )** – test hash table integrity

SYNOPSIS

```
void dcacheHashTest
    (
    CBIO_DEV_ID dev
    )
```

DESCRIPTION

SEE ALSO **dcacheCbio**

dcacheShow( )

NAME **dcacheShow( )** – print information about disk cache

SYNOPSIS

```
void dcacheShow
    (
    CBIO_DEV_ID dev,          /* device handle */
    int         verbose       /* 1 - display state of each cache block */
    )
```

DESCRIPTION This routine displays various information regarding a disk cache, namely current disk parameters, cache size, tunable parameters and performance statistics. The information is displayed on the standard output.

The *dev* argument is the device handle, if it is **NULL**, all disk caches are displayed.

RETURNS N/A

SEE ALSO **dcacheCbio**

devs( )

NAME **devs( )** – list all system-known devices

SYNOPSIS

```
void devs (void)
```

DESCRIPTION This command displays a list of all devices known to the I/O system.

RETURNS N/A

SEE ALSO **usrLib**, **iosDevShow( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

dhcpcBind( )

NAME

dhcpcBind( ) – obtain a set of network configuration parameters with DHCP

SYNOPSIS

```
STATUS dhcpcBind
    (
    void * pCookie,           /* identifier returned by dhcpcInit() */
    BOOL   syncFlag           /* synchronous or asynchronous execution */
    )
```

DESCRIPTION

This routine initiates a DHCP negotiation according to the process described in RFC 1541. The *pCookie* argument contains the return value of an earlier **dhcpcInit( )** call and is used to identify a particular lease.

The *syncFlag* parameter specifies whether the DHCP negotiation started by this routine will execute synchronously or asynchronously. An asynchronous execution will return after starting the DHCP negotiation, but a synchronous execution will only return once the negotiation process completes.

When a new lease is established, any event hook provided for the lease will be called to process the configuration parameters. The hook is also called when the lease expires or the negotiation process fails. The results of an asynchronous DHCP negotiation are not available unless an event hook is installed.

If automatic configuration of the underlying network interface was specified during the lease initialization, this routine will prevent all higher-level protocols from accessing the underlying network interface used during the initial lease negotiation until that process is complete. In addition, any addressing information obtained will be applied to that network interface, which will remain disabled if the initial negotiation fails. Finally, the interface will be disabled if the lease expires.

NOTE: If the DHCP client is used to obtain the VxWorks boot parameters, this routine is called automatically during system startup using the automatic reconfiguration. Therefore, any calls to this routine which use the network boot device for message transfer when the DHCP client was used at boot time must not request automatic reconfiguration during initialization. Otherwise, the resulting lease settings will conflict with the configuration maintained by the lease established during system startup.

RETURNS

OK if routine completes, or **ERROR** otherwise.

ERRNO

S_dhcpcLib_BAD_COOKIE, **S_dhcpcLib_NOT_INITIALIZED**, **S_dhcpcLib_BAD_OPTION**, **S_dhcpcLib_BAD_DEVICE**

SEE ALSO

dhcpcLib

dhcpcBootBind( )

NAME **dhcpcBootBind( )** – initialize the network with DHCP at boot time

SYNOPSIS

```
STATUS dhcpcBootBind (void)
```

DESCRIPTION This routine performs the client side of a DHCP negotiation according to RFC 2131. The negotiation uses the network device specified with the initialization call. The addressing information retrieved is applied to that network device. Because the boot image is replaced by the downloaded target image, the resulting lease cannot be renewed. Therefore, the minimum lease length specified by **DHCPC_MIN_LEASE** must be set so that the target image has sufficient time to download and begin monitoring the lease. This routine is called automatically by the boot program when **INCLUDE_DHCPC** is defined and the automatic configuration option is set in the boot flags and no target address is present.

RETURNS **OK** if negotiation is successful, or **ERROR** otherwise.

ERRNO N/A

SEE ALSO **dhcpcBootLib**

dhcpcBootInformGet( )

NAME **dhcpcBootInformGet( )** – obtain additional configuration parameters with DHCP

SYNOPSIS

```
STATUS dhcpcBootInformGet
    (
    char * pAddrString        /* known address assigned to client */
    )
```

DESCRIPTION This routine uses DHCP to retrieve additional configuration parameters for a client with the externally configured network address given by the *pAddrString* parameter. It sends an INFORM message and waits for a reply following the process described in RFC 2131. The message exchange uses the network device specified with the initialization call. Any interface information retrieved is applied to that network device. Since this process does not establish a lease, the target address will not contain any timestamp information so that the runtime image will not attempt to verify the configuration parameters. This routine is called automatically by the boot program when **INCLUDE_DHCPC** is defined and the automatic configuration option is set in the boot flags if a target address is already present.

RETURNS **OK** if negotiation is successful, or **ERROR** otherwise.

ERRNO N/A

SEE ALSO **dhcpcBootLib**

dhcpcBootInit( )

NAME **dhcpcBootInit( )** – set up the DHCP client parameters and data structures

SYNOPSIS

```
void * dhcpcBootInit
    (
    struct ifnet * pIf,        /* network device used by client */
    int            serverPort,   /* port used by DHCP servers (default 67) */
    int            clientPort,   /* port used by DHCP clients (default 68) */
    int            maxSize,      /* largest DHCP message supported, in bytes */
    int            offerTimeout, /* interval to get additional DHCP offers */
    int            defaultLease, /* default value for requested lease length */
    int            minLease      /* minimum accepted lease length */
    )
```

DESCRIPTION This routine creates any necessary data structures and sets the client's option request list to retrieve a subnet mask and broadcast address for the network interface indicated by *pIf*. The routine is executed automatically by the boot program when **INCLUDE_DHCPC** is defined and the automatic configuration option is set in the boot flags. The network interface specified by *pIf* is used to transmit and receive all DHCP messages during the lease negotiation. The DHCP client supports interfaces attached to the IP protocol using the MUX/END interface and BSD Ethernet devices attached to that protocol. The interface must be capable of sending broadcast messages. The *maxSize* parameter specifies the maximum length supported for any DHCP message, including the UDP and IP headers and the link level header. The maximum length of the DHCP options field is based on this value or the MTU size for the given interface, whichever is less. The smallest valid value for the *maxSize* parameter is 576 bytes, corresponding to the minimum IP datagram a host must accept. The MTU size of the network interface must be large enough to handle those datagrams.

ERRNO N/A

RETURNS Lease handle for later use, or **NULL** if lease startup fails.

SEE ALSO **dhcpcBootLib**

D

dhcpcCacheHookAdd( )

NAME **dhcpcCacheHookAdd( )** – add a routine to store and retrieve lease data

SYNOPSIS

```
STATUS dhcpcCacheHookAdd
    (
    FUNCPTR pCacheHookRtn     /* routine to store/retrieve lease data */
    )
```

DESCRIPTION This routine adds a hook routine that is called at the bound state (to store the lease data) and during the **INIT_REBOOT** state (to re-use the parameters if the lease is still active). The calling sequence of the input hook routine is:

```
STATUS dhcpcCacheHookRtn
    (
    int command,                     /* requested cache operation */
    unsigned long *pTimeStamp,       /* lease timestamp data */
    int *pDataLen,                   /* length of data to access */
    char *pBuffer                    /* pointer to data buffer */
    )
```

The hook routine should return **OK** if the requested operation is completed successfully, or **ERROR** otherwise. All the supplied pointers reference memory locations that are reused upon return from the hook. The hook routine must copy the data elsewhere.

NOTE: The setting of the cache hook routine during a **dhcpcInit( )** call is recorded and used by the resulting lease throughout its lifetime. Since the hook routine is intended to store a single lease record, a separate hook routine should be specified before the **dhcpcInit( )** call for each lease which will re-use its parameters across reboots.

IMPLEMENTATION The *command* parameter specifies one of the following operations:

DHCP_CACHE_WRITE
Save the indicated data. The write operation must preserve the value referenced by *pTimeStamp* and the contents of *pBuffer*. The *pDataLen* parameter indicates the number of bytes in that buffer.

DHCP_CACHE_READ
Restore the saved data. The read operation must copy the data from the most recent write operation into the location indicated by *pBuffer*, set the contents of *pDataLen* to the amount of data provided, and store the corresponding timestamp value in *pTimeStamp*.

– The read operation has very specific requirements. On entry, the value referenced by *pDataLen* indicates the maximum buffer size available at *pBuffer*. If the amount of data stored by the previous write exceeds this value, the operation must return **ERROR**. A read must also return **ERROR** if the saved timestamp value is 0. Finally, the read operation must return **ERROR** if it is unable to retrieve all the data stored by the write operation or if the previous write was unsuccessful.

DHCP_CACHE_ERASE
Ignore all stored data. Following this operation, subsequent read operations must return **ERROR** until new data is written. All parameters except *command* are **NULL**.

RETURNS **OK**, always.

ERRNO N/A

SEE ALSO **dhcpcLib**

dhcpcCacheHookDelete( )

NAME **dhcpcCacheHookDelete( )** – delete a lease data storage routine

SYNOPSIS

```
STATUS dhcpcCacheHookDelete (void)
```

DESCRIPTION This routine deletes the hook used to store lease data, preventing re-use of the configuration parameters across system reboots for all subsequent lease attempts. Currently active leases will continue to use the routine specified before the lease initialization.

RETURNS **OK**, always.

ERRNO N/A

SEE ALSO **dhcpcLib**

D

dhcpcEventHookAdd( )

NAME **dhcpcEventHookAdd( )** – add a routine to handle configuration parameters

SYNOPSIS

```
STATUS dhcpcEventHookAdd
    (
    void *  pCookie,          /* identifier returned by dhcpcInit() */
    FUNCPTR pEventHook        /* routine to handle lease parameters */
    )
```

DESCRIPTION This routine installs a hook routine to handle changes in the configuration parameters provided for the lease indicated by *pCookie*. The hook provides an alternate configuration method for DHCP leases and uses the following interface:

```
void dhcpcEventHookRtn
    (
    int       leaseEvent,     /* new or expired parameters */
    void *    pCookie         /* lease identifier from dhcpcInit() */
    )
```

The routine is called with the *leaseEvent* parameter set to **DHCPC_LEASE_NEWwhenever** a lease is successfully established. The **DHCPC_LEASE_NEW** event does not occur when a lease is renewed by the same DHCP server, since the parameters do not change in that case. However, it does occur if the client rebinds to a different DHCP server. The **DHCPC_LEASE_INVALID** event indicates that the configuration parameters for the corresponding lease may no longer be used. That event occurs when a lease expires or a renewal or verification attempt fails, and coincides with re-entry into the initial state of the negotiation process.

If the lease initialization specified automatic configuration of the corresponding network interface, any installed hook routine will be invoked after the new address information is applied.

RETURNS **OK** if notification hook added, or **ERROR** otherwise.

ERRNO **S_dhcpcLib_BAD_COOKIE**, **S_dhcpcLib_NOT_INITIALIZED**

SEE ALSO **dhcpcLib**

dhcpcEventHookDelete( )

NAME **dhcpcEventHookDelete( )** – remove the configuration parameters handler

SYNOPSIS
```
STATUS dhcpcEventHookDelete
    (
    void * pCookie              /* identifier returned by dhcpcInit() */
    )
```

DESCRIPTION This routine removes the hook routine that handled changes in the configuration parameters for the lease indicated by *pCookie*. If the lease initialization specified automatic configuration of the corresponding network interface, the assigned address could change without warning after this routine is executed.

RETURNS **OK** if notification hook removed, or **ERROR** otherwise.

ERRNO **S_dhcpcLib_BAD_COOKIE**, **S_dhcpcLib_NOT_INITIALIZED**

SEE ALSO **dhcpcLib**

dhcpcInformGet( )

NAME **dhcpcInformGet( )** – obtain additional configuration parameters with DHCP

SYNOPSIS
```
STATUS dhcpcInformGet
    (
    void * pCookie,             /* identifier returned by dhcpcInit() */
    char * pAddrString,         /* known address assigned to client */
    BOOL   syncFlag             /* synchronous or asynchronous execution? */
    )
```

DESCRIPTION This routine uses DHCP to retrieve additional configuration parameters for a client with the externally configured network address given by the *pAddrString* parameter. It sends an INFORM message and waits for a reply following the process described in RFC 2131. The *pCookie* argument contains the return value of an earlier **dhcpcInit( )** call and is used to access the resulting configuration. Unlike the **dhcpcBind( )** call, this routine does not establish a lease with a server.

The *syncFlag* parameter specifies whether the message exchange started by this routine will execute synchronously or asynchronously. An asynchronous execution will return after sending the initial message, but a synchronous execution will only return once the process completes.

When a server responds with an acknowledgement message, any event hook provided will be called to process the configuration parameters. The hook is also called if the message exchange fails. The results of an asynchronous execution are not available unless an event hook is installed.

NOTE: This routine is designed as an alternative to the **dhcpcBind( )** routine. It should not be used for any **dhcpcInit( )** identifier corresponding to an active or pending lease.

RETURNS **OK** if routine completes, or **ERROR** otherwise.

ERRNO **S_dhcpcLib_BAD_COOKIE**, **S_dhcpcLib_NOT_INITIALIZED**, **S_dhcpcLib_BAD_OPTION**

SEE ALSO **dhcpcLib**

dhcpcInit( )

NAME **dhcpcInit( )** – assign network interface and setup lease request

SYNOPSIS

```
void * dhcpcInit
    (
    struct ifnet * pIf,       /* network device used by client */
    BOOL           autoConfig /* reconfigure network device? */
    )
```

DESCRIPTION This routine creates the data structures used to obtain a set of parameters with DHCP and must be called before each attempt at establishing a DHCP lease, but after the **dhcpcLibInit( )** routine has initialized the global data structures.

The *pIf* argument indicates the network device which will be used for transmission and reception of DHCP messages during the lifetime of the lease. The DHCP client supports devices attached to the IP protocol with the MUX/END interface. The specified device must be capable of sending broadcast messages. It also supports BSD Ethernet devices attached to the IP protocol. The MTU size of any interface must be large enough to receive a minimum IP datagram of 576 bytes. If the interface MTU size is less than the maximum message size set in the library initialization it also determines the maximum length of the DHCP options field.

If the *autoConfig* parameter is set to **TRUE**, any address information obtained will automatically be applied to the specified interface. The *autoConfig* parameter also selects the default option request list for a lease. If set to **FALSE**, no specific lease options are requested since any configuration parameters obtained are not intended for the underlying network device. In that case, any specific options required may be added to the request list at any time before the corresponding **dhcpcBind( )** call. If *autoConfig* is **TRUE**, this routine sets the configuration parameters to request the minimal address information (subnet mask and broadcast address) necessary for reconfiguring the network device specified by *pIf*.

The internal lease identifier returned by this routine must be used in subsequent calls to the DHCP client library.

NOTE: This routine is called automatically during system startup if the DHCP client was used to obtain the VxWorks boot parameters. The resulting lease will always reconfigure the network boot device. Therefore, any further calls to this routine which specify the network boot device for use in obtaining additional DHCP leases must set *autoConfig* to **FALSE**. Otherwise, that device will be unable to maintain a stable configuration. The global variable **pDhcpcBootCookie** provides access to the configuration parameters for any DHCP lease created during system startup.

RETURNS

Lease handle for later use, or **NULL** if lease setup fails.

ERRNO

S_dhcpcLib_NOT_INITIALIZED, **S_dhcpcLib_BAD_DEVICE**, **S_dhcpcLib_BAD_OPTION**, **S_dhcpcLib_MAX_LEASES_REACHED**, **S_dhcpcLib_MEM_ERROR**

SEE ALSO

dhcpcLib, **dhcpcOptionSet( )**, **dhcpcEventHookAdd( )**

dhcpcLibInit( )

NAME

dhcpcLibInit( ) – DHCP client library initialization

SYNOPSIS

```
STATUS dhcpcLibInit
    (
    int serverPort,          /* port used by DHCP servers (default 67) */
    int clientPort,          /* port used by DHCP clients (default 68) */
    int maxLeases,           /* max number of simultaneous leases allowed */
    int maxSize,             /* largest DHCP message supported, in bytes */
    int offerTimeout,        /* interval to get additional DHCP offers */
    int defaultLease,        /* default value for requested lease length */
    int minLease             /* minimum accepted lease length */
    )
```

DESCRIPTION

This routine creates and initializes the global data structures used by the DHCP client library to maintain multiple leases, up to the limit specified by the *maxLeases* parameter.

Every subsequent lease attempt will collect additional DHCP offers until the interval specified by *offerTimeout*expires and will request the lease duration indicated by *defaultLease*. The *maxSize* parameter specifies the maximum length supported for any DHCP message, including the UDP and IP headers and the largest link level header for all supported devices. The maximum length of the DHCP options field is based on this value or the MTU size for a lease's underlying interface, whichever is less. The smallest valid value for the *maxSize* parameter is 576 bytes, corresponding to the minimum IP datagram a host must accept. Larger values will allow the client to handle longer DHCP messages.

This routine must be called before calling any other library routines. The routine is called automatically if **INCLUDE_DHCPC** is defined at the time the system is built and assigns the global lease settings to the values specified by **DHCPC_SPORT**, **DHCPC_CPORT**, **DHCPC_MAX_LEASES**, **DHCPC_MAX_MSGSIZE**, **DHCPC_DEFAULT_LEASE**, and **DHCPC_OFFER_TIMEOUT**.

RETURNS **OK**, or **ERROR** if initialization fails.

ERRNO **S_dhcpcLib_MEM_ERROR**

SEE ALSO **dhcpcLib**

---

dhcpcOptionAdd( )

NAME **dhcpcOptionAdd( )** – add an option to the client messages

SYNOPSIS

```
STATUS dhcpcOptionAdd
    (
    void *  pCookie,          /* identifier returned by dhcpcInit() */
    UCHAR   option,           /* RFC 2132 tag of desired option */
    int     length,           /* length of option data */
    UCHAR * pData             /* option data */
    )
```

DESCRIPTION This routine inserts option tags and associated values into the body of all outgoing messages for the lease indicated by the *pCookie* parameter. Each lease can accept option data up to the MTU size of the underlying interface, minus the link-level header size and the additional 283 bytes required for a minimum DHCP message (including mandatory options).

The *option* parameter specifies an option tag defined in RFC 2132. See the **dhcp/dhcp.h** include file for a listing of defined aliases for the available option tags. This routine will not accept the following *option* values, which are used for control purposes and cannot be included arbitrarily:

_DHCP_PAD_TAG
_DHCP_OPT_OVERLOAD_TAG
_DHCP_MSGTYPE_TAG
_DHCP_SERVER_ID_TAG
_DHCP_MAXMSGSIZE_TAG
_DHCP_END_TAG

This routine also will not accept *option* values 62 or 63, which are not currently defined.

The *length* parameter indicates the number of bytes in the option body provided by the *pData* parameter.

The maximum length of the option field in a DHCP message depends on the MTU size of the associated interface and the maximum DHCP message size set during the DHCP library initialization. These option settings share that field with any option request list created through the **dhcpcOptionSet( )** routine. Options which exceed the limit will not be stored.

Each call to this routine with the same *option* value usually replaces the value of the existing option, if any. However, the routine will append the new data for the *option* values which contain variable length lists, corresponding to tags 3-11, 21, 25, 33, 41-45, 48-49, 55, 65, and 68-76.

WARNING: The **_DHCP_REQ_LIST_TAG** *option* value (55) will replace any existing list created with the **dhcpcOptionSet( )** routine.

RETURNS

OK if the option was inserted successfully, or **ERROR** if the option is invalid or storage failed.

ERRNO

S_dhcpcLib_BAD_OPTION, **S_dhcpcLib_OPTION_NOT_STORED**

SEE ALSO

dhcpcCommonLib

dhcpcOptionGet( )

NAME

dhcpcOptionGet( ) – retrieve an option provided to a client and store in a buffer

SYNOPSIS

```
STATUS dhcpcOptionGet
    (
    void * pCookie,           /* identifier returned by dhcpcInit() */
    int    option,            /* RFC 2132 option tag */
    int *  pLength,           /* size of provided buffer and data returned */
    char * pBuf               /* location for option data */
    )
```

DESCRIPTION

This routine retrieves the data for a specified option from a lease indicated by the *pCookie* parameter. The *option* parameter specifies an option tag as defined in RFC 2132. See the **dhcp/dhcp.h** include file for a listing of defined aliases for the available option tags. This routine will not accept the following *option* values, which are either used by the server for control purposes or only supplied by the client:

_DHCP_PAD_TAG
_DHCP_REQUEST_IPADDR_TAG
_DHCP_OPT_OVERLOAD_TAG
_DHCP_MSGTYPE_TAG
_DHCP_REQ_LIST_TAG
_DHCP_MAXMSGSIZE_TAG
_DHCP_CLASS_ID_TAG
_DHCP_CLIENT_ID_TAG
_DHCP_END_TAG

If the option is found, the data is stored in the provided buffer, up to the limit specified in the *pLength* parameter. The option is not available if the DHCP client is not in the bound state or if the server did not provide it. After returning, the *pLength* parameter indicates the amount of data actually retrieved. The provided buffer may contain IP addresses stored in network byte order. All other numeric values are stored in host byte order. See RFC 2132 for specific details on the data retrieved.

RETURNS

OK if option available, or **ERROR** otherwise.

ERRNO

S_dhcpcLib_BAD_COOKIE, **S_dhcpcLib_NOT_INITIALIZED**, **S_dhcpcLib_NOT_BOUND**, **S_dhcpcLib_OPTION_NOT_PRESENT**

SEE ALSO

dhcpcLib, **dhcpcOptionSet( )**

dhcpcOptionSet( )

NAME

dhcpcOptionSet( ) – add an option to the option request list

SYNOPSIS

```
STATUS dhcpcOptionSet
    (
    void * pCookie,           /* identifier returned by dhcpcInit() */
    int    option             /* RFC 2132 tag of desired option */
    )
```

DESCRIPTION

This routine specifies which options the lease indicated by the *pCookie* parameter will request from a server. The *option* parameter specifies an option tag as defined in RFC 2132. See the **dhcp/dhcp.h** include file for a listing of defined aliases for the available option tags. This routine will not accept the following *option* values, which are either used by the server for control purposes or only supplied by the client:

_DHCP_PAD_TAG
_DHCP_REQUEST_IPADDR_TAG
_DHCP_LEASE_TIME_TAG
_DHCP_OPT_OVERLOAD_TAG
_DHCP_MSGTYPE_TAG
_DHCP_SERVER_ID_TAG
_DHCP_REQ_LIST_TAG
_DHCP_ERRMSG_TAG
_DHCP_MAXMSGSIZE_TAG
_DHCP_CLASS_ID_TAG
_DHCP_CLIENT_ID_TAG
_DHCP_END_TAG

This routine also will not accept *option* values 62 or 63, which are not currently defined.

The maximum length of the option field in a DHCP message depends on the MTU size of the associated interface and the maximum DHCP message size set during the DHCP library initialization. Both the option request list and the options sent by the client through the **dhcpcOptionAdd( )** routine share that field. Options which exceed the limit will not be stored.

NOTE: The boot program automatically requests all options necessary for default target configuration. This routine is only necessary to support special circumstances in which additional options are required. Any options requested in that case may be retrieved after the runtime image has started.

NOTE: The DHCP specification forbids changing the option request list after a lease has been established. Therefore, this routine must not be used after the **dhcpcBind( )** call (in a runtime image) or the **dhcpcBootBind( )** call (for a boot image). Changing the request list at that point could have unpredictable results.

NOTE: Options are added directly to outgoing DHCP messages, and numeric options (*e.g.*, lease duration time) are expected to be provided in network byte order. Care must be taken on little-endian hosts to insure that numeric arguments are properly byte-swapped before being passed to this routine.

RETURNS
OK if the option was set successfully, or **ERROR** if the option is invalid or storage failed.

ERRNO
S_dhcpcLib_BAD_OPTION, S_dhcpcLib_OPTION_NOT_STORED

SEE ALSO
dhcpcCommonLib

D

dhcpcParamsGet( )

NAME

dhcpcParamsGet( ) – retrieve current configuration parameters

SYNOPSIS

```
STATUS dhcpcParamsGet
    (
    void *              pCookie,    /* identifier returned by dhcpcInit() */
    struct dhcp_param * pParamList  /* requested parameters */
    )
```

DESCRIPTION

This routine copies the current configuration parameters for the lease specified by the *pCookie* argument to the user-supplied and allocated **dhcp_param** structure referenced in *pParamList*. Within this structure, defined in **h/dhcp/dhcpc.h**, you should supply buffer pointers for the parameters that interest you. Set all other structure members to zero. When **dhcpcParamsGet( )** returns, the buffers you specified in the submitted **dhcpc_param** structure will contain the information you requested. This assumes that the specified lease is in the bound state and that DHCP knows that the lease parameters are good.

NOTE: The **temp_sname** and **temp_file** members of the **dhcp_param** structure are for internal use only. They reference temporary buffers for options that are passed using the **sname** and **file** members. Do not request either **temp_sname** or **temp_file**. Instead, request either **sname** or **file** if you want those parameters.

Many of the parameters within the user-supplied structure use one of the following secondary data types: **struct in_addrs**, **struct u_shorts**, and **struct vendor_list**. Each of those structures accepts a length designation and a data pointer. For the first two data types, the **num** member indicates the size of the buffer in terms of the number of underlying elements. For example, the **STATIC_ROUTE** option returns one or more IP address pairs. Thus, setting the **num** member to 2 in the **static_route** entry would indicate that the corresponding buffer contained 16 bytes. By contrast, the **len** member in the struct **vendor_list** data type consists of the buffer size, in bytes. See RFC 1533 for specific details on the types of data for each option.

On return, each of the length designators are set to indicate the amount of data returned. For instance, the **num** member in the **static_route** entry could be set to 1 to indicate that only one IP address pair of 8 bytes was available.

RETURNS

OK if in bound state, or **ERROR** otherwise.

ERRNO

S_dhcpcLib_BAD_COOKIE, S_dhcpcLib_NOT_INITIALIZED, S_dhcpcLib_NOT_BOUND

SEE ALSO

dhcpcLib

dhcpcParamsShow( )

NAME
dhcpcParamsShow( ) – display current lease parameters

SYNOPSIS
```
STATUS dhcpcParamsShow
    (
    void * pCookie            /* identifier returned by dhcpcInit() */
    )
```

DESCRIPTION
This routine prints all lease parameters for the lease identified by *pCookie*. It has no effect if the indicated lease is not currently active.

RETURNS
OK, or **ERROR** if lease identifier unknown.

ERRNO
S_dhcpcLib_BAD_COOKIE

SEE ALSO
dhcpcShow

dhcpcRelease( )

NAME
dhcpcRelease( ) – relinquish specified lease

SYNOPSIS
```
STATUS dhcpcRelease
    (
    void * pCookie            /* identifier returned by dhcpcInit() */
    )
```

DESCRIPTION
This routine schedules the lease identified by the *pCookie* parameter for immediate release, regardless of time remaining, and removes all the associated data structures. After the release completes, a new call to **dhcpcInit( )** is required before attempting another lease.

NOTE: This routine will disable the underlying network interface if automatic configuration was requested. This may occur without warning if no event hook is installed.

RETURNS
OK if release scheduled, or **ERROR** otherwise.

ERRNO
S_dhcpcLib_BAD_COOKIE, S_dhcpcLib_NOT_INITIALIZED

SEE ALSO
dhcpcLib

dhcpcServerGet( )

NAME

dhcpcServerGet( ) – retrieve the current DHCP server

SYNOPSIS

```
STATUS dhcpcServerGet
    (
    void *           pCookie,    /* identifier returned by dhcpcInit() */
    struct in_addr * pServerAddr /* location for address of server */
    )
```

DESCRIPTION

This routine returns the DHCP server that supplied the configuration parameters for the lease specified by the *pCookie* argument. This information is available only if the lease is in the bound state.

RETURNS

OK if in bound state and server available, or **ERROR** otherwise.

ERRNO

S_dhcpcLib_BAD_COOKIE, **S_dhcpcLib_NOT_INITIALIZED**, **S_dhcpcLib_NOT_BOUND**

SEE ALSO

dhcpcLib

dhcpcServerShow( )

NAME

dhcpcServerShow( ) – display current DHCP server

SYNOPSIS

```
STATUS dhcpcServerShow
    (
    void * pCookie            /* identifier returned by dhcpcInit() */
    )
```

DESCRIPTION

This routine prints the IP address of the DHCP server that provided the parameters for the lease identified by *pCookie*. It has no effect if the indicated lease is not currently active.

RETURNS

OK, or **ERROR** if lease identifier unknown.

ERRNO

S_dhcpcLib_BAD_COOKIE

SEE ALSO

dhcpcShow

dhcpcShowInit( )

NAME
dhcpcShowInit( ) – initialize the DHCP show facility

SYNOPSIS

```
void dhcpcShowInit (void)
```

DESCRIPTION
This routine links the DHCP show facility into the VxWorks system image. It is called from **usrNetwork.c** automatically if **INCLUDE_DHCP** and **INCLUDE_NET_SHOW** are defined at the time the image is constructed.

SEE ALSO
dhcpcShow

dhcpcShutdown( )

NAME
dhcpcShutdown( ) – disable DHCP client library

SYNOPSIS

```
STATUS dhcpcShutdown (void)
```

DESCRIPTION
This routine schedules the lease monitor task to clean up memory and exit, after releasing all currently active leases. The network boot device will be disabled if the DHCP client was used to obtain the VxWorks boot parameters and the resulting lease is still active. Any other interfaces using the addressing information from leases set for automatic configuration will also be disabled. Notification of a disabled interface will not occur unless an event hook has been installed. After the processing started by this request completes, the DHCP client library is unavailable until restarted with the **dhcpcLibInit( )** routine.

RETURNS
OK if shutdown scheduled, or **ERROR** otherwise.

ERRNO
S_dhcpcLib_NOT_INITIALIZED

SEE ALSO
dhcpcLib

D

dhcpcTimerGet( )

NAME

dhcpcTimerGet( ) – retrieve current lease timers

SYNOPSIS

```
STATUS dhcpcTimerGet
    (
    void * pCookie,          /* identifier returned by dhcpcInit() */
    int *  pT1,              /* time until lease renewal */
    int *  pT2               /* time until lease rebinding */
    )
```

DESCRIPTION

This routine returns the number of clock ticks remaining on the timers governing the DHCP lease specified by the *pCookie* argument. This information is only available if the lease is in the bound state. Therefore, this routine will return **ERROR** if a BOOTP reply was accepted.

RETURNS

OK if in bound state and values available, or **ERROR** otherwise.

ERRNO

S_dhcpcLib_BAD_COOKIE, **S_dhcpcLib_NOT_INITIALIZED**, **S_dhcpcLib_NOT_BOUND**, **S_dhcpcLib_OPTION_NOT_PRESENT**, **S_dhcpcLib_TIMER_ERROR**

SEE ALSO

dhcpcLib

dhcpcTimersShow( )

NAME

dhcpcTimersShow( ) – display current lease timers

SYNOPSIS

```
STATUS dhcpcTimersShow
    (
    void * pCookie           /* identifier returned by dhcpcInit() */
    )
```

DESCRIPTION

This routine prints the time remaining with each of the DHCP lease timers for the lease identified by *pCookie*. It has no effect if the indicated lease is not currently active.

RETURNS

OK if show routine completes, or **ERROR** otherwise.

ERRNO

S_dhcpcLib_BAD_COOKIE

SEE ALSO

dhcpcShow

dhcpcVerify( )

NAME

dhcpcVerify( ) – renew an established lease

SYNOPSIS

```
STATUS dhcpcVerify
    (
    void * pCookie             /* identifier returned by dhcpcInit() */
    )
```

DESCRIPTION

This routine schedules the lease identified by the *pCookie* parameter for immediate renewal according to the process described in RFC 1541. If the renewal is unsuccessful, the lease negotiation process restarts. The routine is valid as long as the lease is currently active. The routine is also called automatically in response to a **dhcpcBind( )** call for an existing lease.

NOTE: This routine is only intended for active leases obtained with the **dhcpcBind( )** routine. It should not be used for parameters resulting from the **dhcpcInformGet( )** routine.

NOTE: This routine will disable the underlying network interface if the verification fails and automatic configuration was requested. This may occur without warning if no event hook is installed.

RETURNS

OK if verification scheduled, or **ERROR** otherwise.

ERRNO

S_dhcpcLib_BAD_COOKIE, S_dhcpcLib_NOT_INITIALIZED, S_dhcpcLib_NOT_BOUND

SEE ALSO

dhcpcLib

dhcpsAddressHookAdd( )

NAME

dhcpsAddressHookAdd( ) – assign a permanent address storage hook for the server

SYNOPSIS

```
STATUS dhcpsAddressHookAdd
    (
    FUNCPTR pCacheHookRtn      /* routine to store/retrieve lease entries */
    )
```

DESCRIPTION

This routine allows the server to access some form of permanent storage to preserve additional address entries across restarts. This routine is not required, but leases using unsaved addresses are not renewed. The only argument provided is the name of a function with the following interface:

```
STATUS dhcpsAddressStorageHook (int op,
                                char *name, char *start, char *end,
                                char *params);
```


The first parameter of this storage routine specifies one of the following operations:

DHCPS_STORAGE_START
DHCPS_STORAGE_READ
DHCPS_STORAGE_WRITE
DHCPS_STORAGE_STOP

In response to a START, the storage routine should prepare to return data or overwrite data provided by earlier WRITE operations. For a WRITE, the storage routine must save the contents of the four buffers to permanent storage. Those buffers contain the **NULL**-terminated strings received by the **dhcpsLeaseEntryAdd( )** routine. For a READ, the storage routine should copy previously stored data (as **NULL**-terminated strings) into the provided buffers in the order received by earlier WRITE operations. For a STOP, the storage routine should do any necessary cleanup. After a STOP, the storage routine should return an **ERROR** for all operations except START. However, the STOP operation does not normally occur since the server only deliberately exits following an unrecoverable error. This storage routine must not rely on that operation to handle READ, WRITE, or new START attempts.

The storage routine should return **OK** if successful, **ERROR** otherwise.

Note that, unlike the lease storage routine, there is no CLEAR operation.

Before the server is initialized, VxWorks calls this routine automatically passing in the function named in **DHCPS_ADDRESS_HOOK**.

RETURNS **OK**, or **ERROR** if function pointer is **NULL**.

ERRNO N/A

SEE ALSO **dhcpsLib**

dhcpsInit( )

NAME **dhcpsInit( )** – set up the DHCP server parameters and data structures

SYNOPSIS

```
STATUS dhcpsInit
    (
    DHCPS_CFG_PARAMS * pDhcpsCfg /* configuration parameters */
    )
```

DESCRIPTION This routine creates the necessary data structures, builds the server address pool, retrieves any lease or address information from permanent storage through the user-provided hooks, and initializes the network interfaces for monitoring. It is called at system startup if **INCLUDE_DHCPS** is defined at the time the VxWorks image is built.

The *maxSize* parameter specifies the maximum length supported for any DHCP message, including the UDP and IP headers and the largest link level header for all supported devices. The smallest valid value is 576 bytes, corresponding to the minimum IP datagram a host must accept. Larger values will allow the server to handle longer DHCP messages.

RETURNS **OK**, or **ERROR** if could not initialize.

SEE ALSO **dhcpsLib**

dhcpsLeaseEntryAdd( )

NAME **dhcpsLeaseEntryAdd( )** – add another entry to the address pool

SYNOPSIS

```
STATUS dhcpsLeaseEntryAdd
    (
    char * pName,             /* name of lease entry */
    char * pStartIp,          /* first IP address to assign */
    char * pEndIp,            /* last IP address in assignment range */
    char * pParams            /* formatted string of lease parameters */
    )
```

DESCRIPTION This routine allows the user to add new entries to the address pool without rebuilding the VxWorks image. The routine requires a unique entry name of up to eight characters, starting and ending IP addresses, and a colon-separated list of parameters. Possible values for the parameters are listed in the reference entry for **dhcpsLib**. The parameters also determine the type of lease, which the server uses to determine priority when assigning lease addresses. For examples of possible lease types, see the reference entry for **dhcpsLib**.

RETURNS **OK** if entry read successfully, or **ERROR** otherwise.

ERRNO N/A

SEE ALSO **dhcpsLib**

dhcpsLeaseHookAdd( )

NAME **dhcpsLeaseHookAdd( )** – assign a permanent lease storage hook for the server

SYNOPSIS

```
STATUS dhcpsLeaseHookAdd
    (
    FUNCPTR pCacheHookRtn     /* routine to store/retrieve lease records */
    )
```

DESCRIPTION This routine allows the server to access some form of permanent storage that it can use to store current lease information across restarts. The only argument to **dhcpsLeaseHookAdd( )** is a pointer to a storage routine with the following interface:

```
STATUS dhcpsStorageHook (int op, char *buffer, int datalen);
```

The first parameter of the storage routine specifies one of the following operations:

DHCPS_STORAGE_START
DHCPS_STORAGE_READ
DHCPS_STORAGE_WRITE
DHCPS_STORAGE_STOP
DHCPS_STORAGE_CLEAR

In response to START, the storage routine should prepare to return data or overwrite data provided by earlier WRITEs. For a WRITE, the storage routine must save the contents of the buffer to permanent storage. For a READ, it should copy data previously stored into the provided buffer as a **NULL**-terminated string in FIFO order. For a CLEAR, the storage routine should discard currently stored data. After a CLEAR, the READ operation must return **ERROR** until additional data is stored. For a STOP, the storage routine must handle cleanup. After a STOP, READ and WRITE operations must return error until a START is received. Each of these operations must return **OK** if successful, or **ERROR** otherwise.

Before the server is initialized, VxWorks automatically calls **dhcpsLeaseHookAdd( )**, passing in the routine name defined by **DHCPS_LEASE_HOOK**.

RETURNS **OK**, or **ERROR** if routine is **NULL**.

ERRNO N/A

SEE ALSO **dhcpsLib**

difftime( )

NAME

difftime( ) – compute the difference between two calendar times (ANSI)

SYNOPSIS

```
double difftime
    (
    time_t time1,             /* later time, in seconds */
    time_t time0              /* earlier time, in seconds */
    )
```

DESCRIPTION

This routine computes the difference between two calendar times: *time1 - time0*.

INCLUDE FILES

time.h

RETURNS

The time difference in seconds, expressed as a double.

SEE ALSO

ansiTime

dirList( )

NAME

dirList( ) – list contents of a directory (multi-purpose)

SYNOPSIS

```
STATUS dirList
    (
    int    fd,                /* file descriptor to write on */
    char * dirName,           /* 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.

SEE ALSO **usrFsLib**, **dirLib**, **dosFsLib**, **ls( )**, **ll( )**, **lsr( )**, **llr( )**

---

diskFormat( )

NAME **diskFormat( )** – format a disk

SYNOPSIS

```
STATUS diskFormat
    (
    const char * pDevName     /* name of the device to initialize */
    )
```

DESCRIPTION This command formats a disk and creates a file system on it. The device must already have been created by the device driver and initialized for use with a particular file system, via **dosFsDevInit( )**.

This command calls **ioctl( )** to perform the **FIODISKFORMAT** function.

EXAMPLE

```
    -> diskFormat "/fd0/"
```

RETURNS **OK**, or **ERROR** if the device cannot be opened or formatted.

SEE ALSO **usrFsLib**, **dosFsLib**, *VxWorks Programmer's Guide: Target Shell*

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, use of **dosFsVolFormat( )** is recommended.

This command creates a new, blank file system on a block device. The device must already have been created by the device driver and initialized for use with a particular file system, via **dosFsDevCreate( )**.

EXAMPLE

```
    -> diskInit "/fd0/"
```

Note that if the disk is unformatted, it can not be mounted, thus **open( )** will return error, in which case use the **dosFsVolFormat( )** routine manually.

This routine performs the **FIODISKINIT** ioctl operation.

RETURNS

OK, or **ERROR** if the device cannot be opened or initialized.

SEE ALSO

usrFsLib, **dosFsLib**, *VxWorks Programmer's Guide: Target Shell*

distCtl( )

NAME

distCtl( ) – perform a distributed objects control function (VxFusion Opt.)

SYNOPSIS

```
int distCtl
    (
    int function,             /* function code */
    int argument              /* arbitrary argument */
    )
```

DESCRIPTION

This routine sets various parameters and hooks that control the system. It uses a syntax similar to that of the **ioctl( )** routine. It accepts the following functions:

DIST_CTL_LOG_HOOK
This function sets a routine to be called each time a log message is produced. By default, the log hook writes the message to standard output. The prototype of the

log( ) routine should look like this:

```
void log (char *logMsg);
```

DIST_CTL_PANIC_HOOK
This function sets a routine to be called when the system panics. By default, the panic hook writes the panic message to standard output. The **panic( )** routine must not return. The prototype of the **panic( )** routine should look like this:

```
void panic (char *panicMsg);
```

DIST_CTL_RETRY_TIMEOUT
This function sets the initial send retry timeout in clock ticks. If no ACK is received within a timeout period, the packet is resent. The default value and granularity of **DIST_CTL_RETRY_TIMEOUT** is system dependent.

vxWorks Version	Default Value	Granularity
5.4 and below	1000ms	500ms
5.5 and AE	200ms	100ms

DIST_CTL_RETRY_TIMEOUT is designated in ticks, but rounded down to a multiple of the system's granularity. The timeout period for the *n*th send is:

n * **DIST_CTL_RETRY_TIMEOUT**

DIST_CTL_MAX_RETRIES
This function sets a limit for the number of retries when sending fails. The default value is system dependent, but is set to 5 for all current versions of vxWorks.

DIST_CTL_NACK_SUPPORT
This function enables or disables the sending of negative acknowledgments (NACKs). NACKs are used to request a resend of a single missing fragment from a packet. They are sent immediately after a fragment is found to be missing. If *arg* is **FALSE** (0), the sending of negative acknowledgments is disabled. If *arg* is **TRUE** (1), the sending of NACKs is enabled. By default, NACKs are enabled.

DIST_CTL_PGGYBAK_UNICST_SUPPORT
This function enables or disables unicast piggy-backing. When unicast piggy-backing is enabled, the system waits some time until it sends an acknowledgment for a previously received packet. In the meantime, if a data packet is sent to a host already awaiting an acknowledgment, the acknowledgment is delivered (that is, piggy-backed) with the data packet. Enabling piggy-backing is useful for reducing the number of packets sent; however, it increases latency if no data packets are sent while the system waits. When unicast piggy-backing is disabled, an acknowledgment is delivered immediately in its own packet. This function turns piggy-backing on and off for unicast communication only. If *arg* is **FALSE** (0), unicast piggy-backing is disabled. If *arg* is **TRUE** (1), unicast piggy-backing is enabled. By default, piggy-backing is disabled for unicast communication.

DIST_CTL_PGGYBAK_BRDCST_SUPPORT
This function enables or disables broadcast piggy-backing. When broadcast piggy-backing is enabled, the system waits some time until it sends an acknowledgment for a previously received packet. In the meantime, if a data packet is sent to a host already awaiting an acknowledgment, the acknowledgment is delivered (that is, piggy-backed) with the data packet. Enabling piggy-backing is useful for reducing the number of packets sent; however, it increases latency if no broadcast data packets are sent while the system waits. When broadcast piggy-backing is disabled, an acknowledgment is delivered immediately in its own packet. This function turns piggy-backing on and off for broadcast communication only. If *arg* is **FALSE** (0), broadcast piggy-backing is disabled. If *arg* is **TRUE** (1), broadcast piggy-backing is enabled. By default, piggy-backing is disabled for broadcast communication.

DIST_CTL_OPERATIONAL_HOOK
This function adds a routine to a list of routines to be called each time a node shifts to the operational state. A maximum of 8 routines can be added to the list. The prototype of each **operational( )** routine should look as follows:

```
void operational (DIST_NODE_ID nodeStateChanged);
```

DIST_CTL_CRASHED_HOOK
This function adds a routine to a list of routines to be called each time a node shifts to the crashed state. A node shifts to the crashed state when it does not acknowledge a message within the maximum number of retries.   The list can contain a maximum of 8 routines; however VxFusion supplies one routine, leaving room for only 7 user-supplied routines. The prototype of each **crashed( )** routine should look as follows:

```
void crashed (DIST_NODE_ID nodeStateChanged);
```

DIST_CTL_GET_LOCAL_ID
This function returns the local node ID.

DIST_CTL_GET_LOCAL_STATE
This function returns the state of the local node.

DIST_CTL_SERVICE_HOOK
This function sets a routine to be called each time a service fails, for a service invoked by a remote node. The *argument* parameter is a pointer to a **servError( )** routine. The prototype of the **servError( )** routine should look as follows:

```
void servError (int servId, int status);
```

The system is aware of the following services:

```
DIST_ID_MSG_Q_SERV     (0)  /* message queue service       */
DIST_ID_MSG_Q_GRP_SERV (1)  /* message queue group service */
DIST_ID_DNDB_SERV      (2)  /* distributed name database   */
DIST_ID_DGDB_SERV      (3)  /* distributed group database  */
DIST_ID_INCO_SERV      (4)  /* incorporation protocol      */
DIST_ID_GAP_SERV       (5)  /* group agreement protocol    */
```

DIST_CTL_SERVICE_CONF

This function configures a specified service. The *argument* parameter is a pointer to a **DIST_SERV_CONF** structure which holds the service ID and its configuration to be set. **DIST_SERV_CONF** is defined as follows:

```
typedef struct
    {
    int servId;     /* ID of service to configure  */
    int taskPrio;   /* priority of service task    */
    int netPrio;    /* network priority of service */
    } DIST_SERV_CONF;
```

The system is aware of the following services:

```
DIST_ID_MSG_Q_SERV     (0)  /* message queue service       */
DIST_ID_MSG_Q_GRP_SERV (1)  /* message queue group service */
DIST_ID_DNDB_SERV      (2)  /* distributed name database   */
DIST_ID_DGDB_SERV      (3)  /* distributed group database  */
DIST_ID_INCO_SERV      (4)  /* incorporation protocol      */
DIST_ID_GAP_SERV       (5)  /* group agreement protocol    */
```

If one of the configuration parameters is -1, it remains unchanged. The parameter *taskPrio* can range from 0 to 255; *netPrio* can range from 0 to 7.

A service's configuration can be changed at any time.

AVAILABILITY
This routine is distributed as a component of the unbundled distributed message queues option, VxFusion.

RETURNS
OK or the value requested if *function* is known; **ERROR** if *function* is unknown or the argument is invalid.

ERRNO
S_distLib_UNKNOWN_REQUEST
The control function is unknown.

SEE ALSO
distLib

distIfShow( )

NAME

distIfShow( ) – display information about the installed interface adapter (VxFusion Opt.)

SYNOPSIS

```
STATUS distIfShow (void)
```

DESCRIPTION

This routine displays information about the installed interface adapter. It displays the configuration parameters, as well as some statistical data.

EXAMPLE

```
-> distIfShow
Interface Name                  : "UDP adapter"
MTU                             : 1500
Network Header Size             : 14
SWP Buffer                      : 32
Maximum Number of Fragments     : 10
Maximum Length of Packet        : 14860
Broadcast Address               : 0x930b26ff
Telegrams received              : 23
Telegrams received for sending  : 62
Incoming Telegrams discarded    : 0
Outgoing Telegrams discarded    : 0
```

In this example, the installed interface adapter has the name "UDP adapter." The largest telegram that can be transmitted without fragmentation is 1500 bytes long. The network header requires fourteen (14) of those bytes; therefore the largest amount of user data that can be transmitted without fragmentation is 1486 bytes. The sliding window protocol's buffer has 32 entries, which results in a window of size 16. The number of fragments that the packet can be broken into is limited by the size of the sequence field in the network header. The example interface adapter can handle up to 10 fragments, which results in a maximum packet length of 14860 ((1500 - 14) * 128) bytes. The broadcast address of this driver is 0x930b26ff (147.11.38.255). The last four lines of output show statistical data.

AVAILABILITY

This routine is distributed as a component of the unbundled distributed message queues option, VxFusion.

RETURNS

OK, or **ERROR** if there is no interface installed.

SEE ALSO

distIfShow

distInit( )

NAME

distInit( ) – initialize and bootstrap the current node (VxFusion Opt.)

SYNOPSIS

```
STATUS distInit
    (
    DIST_NODE_ID myNodeId,      /* node ID of this node */
    FUNCPTR      ifInitRtn,     /* interface adapter init routine */
    void *       pIfInitConf,   /* ptr to interface configuration */
    int          maxTBufsLog2,  /* max number of telegram buffers */
    int          maxNodesLog2,  /* max number of nodes in node db */
    int          maxQueuesLog2, /* max number of queues on this node */
    int          maxGroupsLog2, /* max number of groups in db */
    int          maxNamesLog2,  /* max bindings in name db */
    int          waitNTicks     /* wait n ticks when bootstrapping */
    )
```

DESCRIPTION

This routine initializes VxFusion on the current node. The routine begins by initializing the local databases and other internal services. As part of this process, the current node is given the address specified by the *myNodeId* argument.

Secondly, this routine links a network driver to the stack by calling the interface adapter initialization routine specified by the *ifInitRtn* argument. If the interface adapter initialization is successful, this routine then initializes the telegram buffer library which is needed for manipulating telegram buffers--the buffers that hold the packets sent between nodes.

Thirdly, this routine attempts to determine what other VxFusion nodes are active on the network. This is done by continually sending a **BOOTSTRAP** telegram, which indicates to other nodes that VxFusion is starting up on this node. Nodes that receive a **BOOTSTRAP** telegram answer by sending an **XACK** telegram. The **XACK** telegram contains information about the remote node. The sender of the first **XACK** received is the godfather for the current node. The purpose of the godfather is to update local databases. If no **XACK** is received within the amount of time specified by the *waitNTicks* argument, it is assumed that this node is the first node to come up on the network.

As soon as a godfather is located or it is assumed that a node sending an **XACK** is the first to do so on the network, the state of the node shifts from the *booting* state to the *network* state. In the network state, all packets are sent using reliable communication channels; therefore all packets must be now acknowledged by the receiver(s).

If a godfather has been located, the current node asks it to update the local databases by sending an **INCO_REQ** packet. The godfather then begins updating the local databases. When the godfather finishes the update, it sends an **INCO_DONE** packet to the node being updated.

Once the database updates have completed, the node moves into the *operational* state and broadcasts an **INCO_UPNOW** packet.

The number of telegram buffers pre-allocated is equal to 2^*maxTBufsLog2*.

Up to 2^*maxNodesLog2* nodes can be handled by the node database.

The number of distributed message queues is limited to 2^*maxQueuesLog2*.

Distributed message queue groups may not exceed 2^*maxGroupsLog2* groups.

The distributed name database can work with up to 2^*maxNamesLog2* entries.

EXAMPLE

```
-> distInit (0x930b2610, distIfUdpInit, "ln0", 9, 5, 7, 6, 8,
(4*sysClkRateGet())
```

This command sets the ID of the local node to 0x930b2610 (147.11.38.16). The **distIfUdpInit( )** routine is called to initialize the interface adapter (in this case, a UDP adapter). The UDP adapter requires a pointer to the hardware interface name as configuration data (in this case, "ln0"). When starting up, 512 (2^9) telegram buffers are pre-allocated. The node database is configured to hold as many as 32 (2^5) nodes, including the current node. 128 (2^7) distributed message queues can be created on the local node. The local group database can hold up to 64 (2^6) groups, while the name database is limited to 256 (2^8) entries.

When the node bootstraps, it waits for 4 seconds (4***sysClkRateGet( )**) to allow other nodes to respond.

NOTE: This routine is called automatically with default parameters when a target boots using a VxWorks image with VxFusion installed.

AVAILABILITY

This routine is distributed as a component of the unbundled distributed message queues option, VxFusion.

RETURNS

OK, or **ERROR** if the initialization fails.

SEE ALSO

distLib, **distLib**

distNameAdd( )

NAME

distNameAdd( ) – add an entry to the distributed name database (VxFusion Opt.)

SYNOPSIS

```
STATUS distNameAdd
    (
    char *         name,      /* name to enter in database */
    void *         value,     /* ptr to value to associate with name */
    int            valueLen,  /* size of value in bytes */
    DIST_NAME_TYPE type       /* type associated with name */
    )
```

DESCRIPTION

This routine adds the name of a specified object, along with its type and value, to the distributed objects distributed name database. All copies of the distributed name database within the system are updated.

The *name* parameter is an arbitrary, null-terminated string with a maximum of 20 characters, including the null terminator.

The value associated with *name* is located at *value* and is of length *valueLen*, currently limited to 8 bytes.

By convention, *type* values of less than 0x1000 are reserved by VxWorks; all other values are user definable. The following types are pre-defined in **distNameLib.h**:

Type Name	Value	Datum
_DIST_MSG_Q	=0	distributed message queue
_DIST_NODE	= 16	node ID
_DIST_UINT8	= 64	8-bit unsigned integer
_DIST_UINT16	= 65	16-bit unsigned integer
_DIST_UINT32	= 66	32-bit unsigned integer
T_DIST_UINT64	= 67	64-bit unsigned integer
T_DIST_FLOAT	= 68	float (32-bit)
T_DIST_DOUBLE	= 69	double (64-bit)

The byte-order of pre-defined types is preserved in a byte-order-heterogeneous network.

The value (and type!) bound to a symbolic name can be changed by calling **distNameAdd( )** with a new value (and type).

This routine returns **OK**, even if some nodes on the system do not respond to the add request broadcast. A node that does not acknowledge a transmission is assumed to have crashed. You can use the **distCtl( )** routine in **distLib** to set a routine to be called in the event that a node crashes.

NOTE: If you add a distributed object ID (**T_DIST_MSG_Q**) to the database, another reference to the object is built. This reference is stored in the database. After the return of **distNameAdd( )**, *value* holds the reference (a new object ID). Use the ID returned by **distNameAdd( )** each time you want to address the global object bound to *name*. Subsequent updates of the binding in the database are transparent. The original object ID specifies exactly the locally created object.

AVAILABILITY
This routine is distributed as a component of the unbundled distributed message queues option, VxFusion.

RETURNS
OK, or **ERROR** if the operation fails.

ERRNO

S_distNameLib_NAME_TOO_LONG
The name being added to the database is too long.

S_distNameLib_ILLEGAL_LENGTH
The argument *valueLen* is not in the range 1 to 8.

S_distNameLib_DATABASE_FULL
The database is full.

S_distNameLib_INCORRECT_LENGTH
The argument *valueLen* is incorrect for the pre-defined *type*.

SEE ALSO
distNameLib, **distLib**

distNameFilterShow( )

NAME
distNameFilterShow( ) – display the distributed name database filtered by type (VxFusion Opt.)

SYNOPSIS

```
void distNameFilterShow
    (
    DIST_NAME_TYPE type        /* type to filter the database by */
    )
```

DESCRIPTION
This routine displays the contents of the distributed name database filtered by *type*. The data displayed includes the symbolic ASCII name, the type, and the value. If the type is not pre-defined, it is printed in decimal and the value shown in a hex dump.

NOTE: Option **VX_FP_TASK** should be set when spawning any task in which **distNameFilterShow( )** is called unless it is certain that no floating point values will be displayed. The target shell has this option set.

D

EXAMPLE

```
-> distNameFilterShow(0)
        NAME                  TYPE                VALUE
-------------------- -------------- -------------------------
dmq-01                 T_DIST_MSG_Q 0x3ff9fb
dmq-02                 T_DIST_MSG_Q 0x3ff98b
dmq-03                 T_DIST_MSG_Q 0x3ff94b
dmq-04                 T_DIST_MSG_Q 0x3ff8db
dmq-05                 T_DIST_MSG_Q 0x3ff89b
grp1                   T_DIST_MSG_Q 0x3ff9bb
grp2                   T_DIST_MSG_Q 0x3ff90b
value = 0 = 0x0
```

AVAILABILITY This routine is distributed as a component of the unbundled distributed message queues option, VxFusion.

RETURNS N/A

SEE ALSO **distNameShow**

distNameFind( )

NAME **distNameFind( )** – find an object by name in the local database (VxFusion Opt.)

SYNOPSIS

```
STATUS distNameFind
    (
    char *           name,    /* name to search for */
    void * *         pValue,  /* where to return ptr to value */
    DIST_NAME_TYPE * pType,   /* where to return type */
    int              waitType /* NO_WAIT or WAIT_FOREVER */
    )
```

DESCRIPTION This routine searches the distributed name database for an object matching a specified *name*. If the object is found, a pointer to the value and its type are copied to the address pointed to by *pValue* and *pType*. If the type is **T_DIST_MSG_Q**, the identifier returned can be used with generic message queue handling routines in **msgQLib**, such as **msgQSend( )**, **msgQReceive( )**, and **msgQNumMsgs( )**.

AVAILABILITY This routine is distributed as a component of the unbundled distributed message queues option, VxFusion.

RETURNS **OK**, or **ERROR** if the search fails.

ERRNO
S_distNameLib_NAME_TOO_LONG
The name to be found in the database is too long.

S_distNameLib_INVALID_WAIT_TYPE
The wait type should be either **NO_WAIT** or **WAIT_FOREVER**.

SEE ALSO
distNameLib

distNameFindByValueAndType( )

NAME
distNameFindByValueAndType( ) – look up the name of an object by value and type (VxFusion Opt.)

SYNOPSIS

```
STATUS distNameFindByValueAndType
    (
    void *         value,      /* value to search for */
    DIST_NAME_TYPE type,       /* type of object for which to search */
    char *         name,       /* where to return name */
    int            waitType    /* NO_WAIT or WAIT_FOREVER */
    )
```

DESCRIPTION
This routine searches the distributed name database for an object matching a specified *value* and *type*. If the object is found, its name is copied to the address pointed to by *name*.

NOTE: Unlike the **smNameFindByValue( )** routine, used with the shared-memory objects name database, this routine must know the type of the object being searched for. Searching on the value only might not return a unique object.

AVAILABILITY
This routine is distributed as a component of the unbundled distributed message queues option, VxFusion.

RETURNS
OK, or **ERROR** if the search fails.

ERRNO
S_distNameLib_INVALID_WAIT_TYPE
The wait type should be either **NO_WAIT** or **WAIT_FOREVER**.

SEE ALSO
distNameLib

D

distNameRemove( )

NAME **distNameRemove( )** – remove an entry from the distributed name database (VxFusion Opt.)

SYNOPSIS

```
STATUS distNameRemove
    (
    char * name                 /* name of object to remove */
    )
```

DESCRIPTION This routine removes an object, that is bound to *name*, from the distributed name database. All copies of the distributed name database get updated.

This routine returns **OK**, even if some nodes on the system do not respond to the remove request broadcast. A node that does not acknowledge a transmission is assumed to have crashed. You can use the **distCtl( )** routine in **distLib** to set a routine to be called in the event that a node crashes.

Removing the name of a distributed object ID (**T_DIST_MSG_Q**) does not invalidate the object ID.

AVAILABILITY This routine is distributed as a component of the unbundled distributed message queues option, VxFusion.

TURNS **OK**, or **ERROR** if the operation fails.

ERRNO **S_distNameLib_NAME_TOO_LONG**
The name to be removed from the database is too long.

SEE ALSO **distNameLib**, **distLib**

distNameShow( )

NAME **distNameShow( )** – display the entire distributed name database (VxFusion Opt.)

SYNOPSIS

```
void distNameShow (void)
```

DESCRIPTION This routine displays the entire contents of the distributed name database. The data displayed includes the symbolic ASCII name, the type, and the value. If the type is not pre-defined, it is printed in decimal and the value shown in a hex dump.

NOTE: Option **VX_FP_TASK** should be set when spawning any task in which **distNameShow( )** is called unless it is certain that no floating point values will be in the database. The target shell has this option set.

EXAMPLE

```
-> distNameShow()
        NAME              TYPE               VALUE
-------------------- -------------- -------------------------
nile                     T_DIST_NODE 0x930b2617 (2466981399)
columbia                 T_DIST_NODE 0x930b2616 (2466981398)
dmq-01                  T_DIST_MSG_Q 0x3ff9fb
dmq-02                  T_DIST_MSG_Q 0x3ff98b
dmq-03                  T_DIST_MSG_Q 0x3ff94b
dmq-04                  T_DIST_MSG_Q 0x3ff8db
dmq-05                  T_DIST_MSG_Q 0x3ff89b
gData                           4096 0x48 0x65 0x6c 0x6c 0x6f 0x00
gCount                 T_DIST_UINT32 0x2d (45)
grp1                    T_DIST_MSG_Q 0x3ff9bb
grp2                    T_DIST_MSG_Q 0x3ff90b
value = 0 = 0x0
```

AVAILABILITY This routine is distributed as a component of the unbundled distributed message queues option, VxFusion.

RETURNS N/A

SEE ALSO **distNameShow**

distTBufAlloc( )

NAME **distTBufAlloc( )** – allocate a telegram buffer from the pool of buffers (VxFusion Opt.)

SYNOPSIS

```
DIST_TBUF * distTBufAlloc (void)
```

DESCRIPTION This routine allocates a telegram buffer from a pre-allocated pool of telegram buffers.

It is the responsibility of the caller to use the **distTBufFree( )** routine to free the buffer when the caller is finished with it.

AVAILABILITY This routine is distributed as a component of the unbundled distributed message queues option, VxFusion.

RETURNS A pointer to a **DIST_TBUF**, or **NULL** if the allocation fails.

SEE ALSO **distTBufLib**, **distTBufFree( )**

distTBufFree( )

NAME **distTBufFree( )** – return a telegram buffer to the pool of buffers (VxFusion Opt.)

SYNOPSIS

```
void distTBufFree
    (
    DIST_TBUF * pTBuf         /* ptr to buffer to be returned to pool */
    )
```

DESCRIPTION This routine returns a buffer previously allocated to a caller back to the pool of free telegram buffers.

AVAILABILITY This routine is distributed as a component of the unbundled distributed message queues option, VxFusion.

RETURNS N/A

SEE ALSO **distTBufLib**, **distTBufAlloc( )**

div( )

NAME **div( )** – compute a quotient and remainder (ANSI)

SYNOPSIS

```
div_t div
    (
    int numer,                /* numerator */
    int denom                 /* denominator */
    )
```

DESCRIPTION This routine computes the quotient and remainder of *numer*/*denom*. If the division is inexact, the resulting quotient is the integer of lesser magnitude that is the nearest to the algebraic quotient. If the result cannot be represented, the behavior is undefined; otherwise, **quot** * *denom* + **rem** equals *numer*.

This routine is not reentrant. For a reentrant version, see **div_r( )**.

INCLUDE FILES **stdlib.h**

RETURNS A structure of type **div_t**, containing both the quotient and the remainder.

SEE ALSO **ansiStdlib**

div_r( )

NAME

div_r( ) – compute a quotient and remainder (reentrant)

SYNOPSIS

```
void div_r
    (
    int      numer,            /* numerator */
    int      denom,            /* denominator */
    div_t *  divStructPtr      /* div_t structure */
    )
```

DESCRIPTION

This routine computes the quotient and remainder of *numer*/*denom*. The quotient and remainder are stored in the **div_t** structure pointed to by *divStructPtr*.

This routine is the reentrant version of **div( )**.

INCLUDE FILES

stdlib.h

RETURNS

N/A

SEE ALSO

ansiStdlib

dosFsChkDsk( )

NAME

dosFsChkDsk( ) – make volume integrity checking.

SYNOPSIS

```
STATUS dosFsChkDsk
    (
    DOS_FILE_DESC_ID pFd,      /* file descriptor of root dir */
    u_int            params    /* check level and verbosity */
    )
```

DESCRIPTION

This library does not makes integrity check process itself, but instead uses routine provided by **dosChkLib**. This routine prepares parameters and invokes checking routine via a pre-initialized function pointer. If **dosChkLib** does not configured into vxWorks, this routine returns **ERROR**.

Ownership on device should be taken by an upper level routine.

RETURNS

STATUS as returned by volume checking routine or **ERROR**, if such routine does not installed.

ERRNO

S_dosFsLib_UNSUPPORTED.

SEE ALSO

dosFsLib

dosFsDevCreate( )

NAME

dosFsDevCreate( ) – create file system device.

SYNOPSIS

```
STATUS dosFsDevCreate
    (
    char *          pDevName,     /* device name */
    CBIO_DEV_ID     cbio,         /* CBIO or cast blkIo device */
    u_int           maxFiles,     /* max no. of simultaneously open files */
    u_int           autoChkLevel  /* automate volume integrity check level */
                                  /* via mounting 0 - default: DOS_CHK_REPAIR
    DOS_CHK_VERB_1 */
    )
```

DESCRIPTION

This routine associates a CBIO device with a logical I/O device name and prepare it to perform file system functions. It takes a **CBIO_DEV_ID** device handle, typically created by **dcacheDevCreate( )**, and defines it as a dosFs volume. As a result, when high-level I/O operations (*e.g.*, **open( )**, **write( )**) are performed on the device, the calls will be routed through **dosFsLib**. The *pCbio* parameter is the handle of the underlying cache or block device.

The argument *maxFiles* specifies the number of files that can be opened at once on the device.

The volume structure integrity can be automatically checked during volume mounting. Parameter *autoChkLevel* defines checking level (**DOS_CHK_ONLY** or **DOS_CHK_REPAIR**), that can be bitwise or-ed with check verbosity level value (**DOS_CHK_VERB_SILENT**, **DOS_CHK_VERB_1** or **DOS_CHK_VERB_2**). If value of *autoChkLevel* is 0, this means default level, that is **DOS_CHK_REPAIR | DOS_CHK_VERB_1**. To prevent check disk autocall, set *autoChkLevel* to NONE.

Note that actual disk accesses are deferred to the time when **open( )** or **creat( )** are first called. That is also when the automatic disk checking will take place. Therefore this function will succeed in cases where a removable disk is not present in the drive.

RETURNS

OK, or **ERROR** if the device name is already in use or insufficient memory.

SEE ALSO

dosFsLib

dosFsLastAccessDateEnable( )

NAME

dosFsLastAccessDateEnable( ) – enable last access date updating for this volume

SYNOPSIS

```
STATUS dosFsLastAccessDateEnable
    (
    DOS_VOLUME_DESC_ID dosVolDescId, /* dosfs volume ID to alter */
    BOOL               enable        /* TRUE = enable update, FALSE = */
                                     /* disable update */
    )
```

DESCRIPTION

This function enables or disables updating of the last access date directory entry field on open-read-close operations for the given dosFs volume. The last access date file indicates the last date that a file has been read or written. When the optional last access date field update is enabled, read operations on a file will cause a write to the media.

RETURNS

OK or **ERROR** if the volume is invalid or enable is not **TRUE** or **FALSE**.

SEE ALSO

dosFsLib

dosFsLibInit( )

NAME

dosFsLibInit( ) – prepare to use the dosFs library

SYNOPSIS

```
STATUS dosFsLibInit
    (
    int ignored
    )
```

DESCRIPTION

This routine initializes the dosFs library. This routine installs **dosFsLib** as a driver in the I/O system driver table, and allocates and sets up the necessary structures. The driver number assigned to **dosFsLib** is placed in the global variable *dosFsDrvNum*.

RETURNS

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

SEE ALSO

dosFsLib

dosFsShow( )

NAME **dosFsShow( )** – display dosFs volume configuration data.

SYNOPSIS
```
STATUS dosFsShow
    (
    void * pDevName,          /* name of device */
    u_int  level              /* detail level */
    )
```

DESCRIPTION This routine obtains the dosFs volume configuration for the named device, formats the data, and displays it on the standard output.

If no device name is specified, the current default device is described.

RETURNS **OK** or **ERROR**, if no valid device specified.

SEE ALSO **dosFsLib**

dosFsVolDescGet( )

NAME **dosFsVolDescGet( )** – convert a device name into a DOS volume descriptor pointer.

SYNOPSIS
```
DOS_VOLUME_DESC_ID dosFsVolDescGet
    (
    void *  pDevNameOrPVolDesc, /* device name or pointer to dos vol desc */
    u_char * * ppTail           /* return ptr for name, used in iosDevFind */
    )
```

DESCRIPTION This routine validates *pDevNameOrPVolDesc* to be a DOS volume descriptor pointer else a path to a DOS device. This routine uses the standard **iosLib** function **iosDevFind( )** to obtain a pointer to the device descriptor. If device is eligible, *ppTail* is filled with the pointer to the first character following the device name. Note that *ppTail* is passed to **iosDevFind( )**. *ppTail* may be passed as **NULL**, in which case it is ignored.

RETURNS A **DOS_VOLUME_DESC_ID** or **NULL** if not a DOSFS device.

ERRNO **S_dosFsLib_INVALID_PARAMETER**

SEE ALSO **dosFsLib**

dosFsVolFormat( )

NAME

dosFsVolFormat( ) – format an MS-DOS compatible volume

SYNOPSIS

```
STATUS dosFsVolFormat
    (
    void *   device,           /* device name or volume or CBIO pointer */
    int      opt,              /* bit-wise or'ed options */
    FUNCPTR pPromptFunc        /* interactive parameter change callback */
    )
```

DESCRIPTION

This utility routine performs the initialization of file system data structures on a disk. It supports FAT12 for small disks, FAT16 for medium size and FAT32 for large volumes. The *device* argument may be either a device name known to the I/O system, or a **dosFsLib** Volume descriptor or a CBIO device handle.

The *opt* argument is a bit-wise or'ed combination of options controlling the operation of this routine as follows:

DOS_OPT_DEFAULT
If the current volume boot block is reasonably intact, use existing parameters, else calculate parameters based only on disk size, possibly reusing only the volume label and serial number.

DOS_OPT_PRESERVE
Attempt to preserve the current volume parameters even if they seem to be somewhat unreliable.

DOS_OPT_BLANK
Disregard the current volume parameters, and calculate new parameters based only on disk size.

DOS_OPT_QUIET
Do not produce any diagnostic output during formatting.

DOS_OPT_FAT16
Format the volume with FAT16 format even if the disk is larger then 2 Gbytes, which would normally be formatted with FAT32.

DOS_OPT_FAT32
Format the volume with FAT32, even if the disk is smaller then 2 Gbytes, but is larger then 512 Mbytes.

DOS_OPT_VXLONGNAMES
Format the volume to use Wind River proprietary case-sensitive Long File Names. Note that this format is incompatible with any other implementation of the MS-DOS file system.

The third argument, *pPromptFunc* is an optional pointer to a function that may interactively prompt the user to change any of the modifiable volume parameters before formatting:

```
void formatPromptFunc( DOS_VOL_CONFIG *pConfig );
```

The <*pConfig< structure upon entry to **formatPromptFunc( )** will contain the initial volume parameters, some of which can be changed before it returns. *pPromptFunc* should be **NULL** if no interactive prompting is required.

COMPATIBILITY

Although this routine tries to format the disk to be compatible with Microsoft implementations of the FAT and FAT32 file systems, there may be differences which are not under WRS control. For this reason, it is highly recommended that any disks which are expected to be interchanged between vxWorks and Windows should be formatted under Windows to provide the best interchangeability. The WRS implementation is more flexible, and should be able to handle the differences when formatting is done on Windows, but Windows implementations may not be able to handle minor differences between their implementation and ours.

AVAILABILITY

This function is an optional part of the MS-DOS file system, and may be included in a target system if it is required to be able to format new volumes.

RETURNS

OK or **ERROR** if was unable to format the disk.

SEE ALSO

dosFsFmtLib

---

dosSetVolCaseSens( )

NAME

dosSetVolCaseSens( ) – set case sensitivity of volume

SYNOPSIS

```
STATUS dosSetVolCaseSens
    (
    DOS_VOLUME_DESC_ID pVolDesc,
    BOOL               sensitivity
    )
```

DESCRIPTION

Pass **TRUE** to setup a case sensitive volume. Pass **FALSE** to setup a case insensitive volume. Note this affects rename lookups only.

RETURNS

TRUE if *pVolDesc* pointed to a DOS volume.

SEE ALSO

dosFsLib

dpartDevCreate( )

NAME

dpartDevCreate( ) – initialize a partitioned disk

SYNOPSIS

```
CBIO_DEV_ID dpartDevCreate
    (
    CBIO_DEV_ID subDev,         /* lower level CBIO device */
    int         nPart,          /* # of partitions */
    FUNCPTR     pPartDecodeFunc /* function to decode partition table */
    )
```

DESCRIPTION

To handle a partitioned disk, this function should be called, with *subDev* as the handle returned from **dcacheDevCreate( )**, It is recommended that for efficient operation a single disk cache be allocated for the entire disk and shared by its partitions.

nPart is the maximum number of partitions which are expected for the particular disk drive. Up to 24 (C-Z) partitions per disk are supported.

PARTITION DECODE FUNCTION

An external partition table decode function is provided via the *pPartDecodeFunc* argument, which implements a particular style and format of partition tables, and fill in the results into a table defined as Pn array of **PART_TABLE_ENTRY** types. See **dpartCbio.h**for definition of **PART_TABLE_ENTRY**. The prototype for this function is as follows:

```
STATUS parDecodeFunc
  (
  CBIO_DEV_ID dev,        /* device from which to read blocks */
  PART_TABLE_ENTRY *pPartTab, /* table where to fill results */
  int nPart               /* # of entries in <pPartTable> */
  )
```

RETURNS

CBIO_DEV_ID or **NULL** if error creating CBIO device.

SEE ALSO

dpartCbio, **dosFsDevCreate( )**.

D

dpartPartGet( )

NAME **dpartPartGet( )** – retrieve handle for a partition

SYNOPSIS

```
CBIO_DEV_ID dpartPartGet
    (
    CBIO_DEV_ID masterHandle, /* CBIO handle of the master partition */
    int         partNum       /* partition number from 0 to nPart */
    )
```

DESCRIPTION This function retrieves a CBIO handle into a particular partition of a partitioned device. This handle is intended to be used with **dosFsDevCreate( )**.

RETURNS **CBIO_DEV_ID** or **NULL** if partition is out of range, or *masterHandle* is invalid.

SEE ALSO **dpartCbio**, **dosFsDevCreate( )**

dspInit( )

NAME **dspInit( )** – initialize DSP support

SYNOPSIS

```
void dspInit (void)
```

DESCRIPTION This routine initializes DSP support and must be called before using the DSP. This is done automatically by the root task, **usrRoot( )**, in **usrConfig.c** when **INCLUDE_DSPis** defined in **configAll.h**.

RETURNS N/A

SEE ALSO **dspLib**

dspShowInit( )

NAME

dspShowInit( ) – initialize the DSP show facility

SYNOPSIS

```
void dspShowInit (void)
```

DESCRIPTION

This routine links the DSP show facility into the VxWorks system. The facility is included automatically when **INCLUDE_SHOW_ROUTINES** and **INCLUDE_DSP** are defined in **configAll.h**.

RETURNS

N/A

SEE ALSO

dspShow

dspTaskRegsShow( )

NAME

dspTaskRegsShow( ) – print the contents of a task's DSP registers

SYNOPSIS

```
void dspTaskRegsShow
    (
    int task                  /* task to display dsp registers for */
    )
```

DESCRIPTION

This routine prints to standard output the contents of a task's DSP registers.

RETURNS

N/A

SEE ALSO

dspShow

E

e( )

NAME

e( ) – set or display eventpoints (WindView)

SYNOPSIS

```
STATUS e
    (
    INSTR * addr,             /* where to set eventpoint, or 0 means */
                              /* display all eventpoints */
    event_t eventId,          /* event ID */
    int     taskNameOrId,     /* task affected; 0 means all tasks */
    FUNCPTR evtRtn,           /* function to be invoked; NULL means no */
                              /* function is invoked */
    int     arg               /* argument to be passed to evtRtn */
    )
```

DESCRIPTION

This routine sets "eventpoints"--that is, breakpoint-like instrumentation markers that can be inserted in code to generate and log an event for use with WindView. Event logging must be enabled with **wvEvtLogEnable( )** for the eventpoint to be logged.

eventId selects the eventpoint number that will be logged: it is in the user event ID range (0-25536).

If *addr* is **NULL**, then all eventpoints and breakpoints are displayed. If *taskNameOrId* is 0, then this event is logged in all tasks. The *evtRtn* routine is called when this eventpoint is hit. If *evtRtn* returns **OK**, then the eventpoint is logged; otherwise, it is ignored. If *evtRtn* is a **NULL** pointer, then the eventpoint is always logged.

Eventpoints are exactly like breakpoints (which are set with the **b( )** command) except in how the system responds when the eventpoint is hit. An eventpoint typically records an event and continues immediately (if *evtRtn* is supplied, this behavior may be different). Eventpoints cannot be used at interrupt level.

To delete an eventpoint, use **bd( )**.

RETURNS

OK, or **ERROR** if *addr* is odd or nonexistent in memory, or if the breakpoint table is full.

SEE ALSO

dbgLib, **wvEvent( )**

edi( )

NAME

edi( ) – return the contents of register **edi** (also **esi** - **eax**) (x86/SimNT)

SYNOPSIS

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

DESCRIPTION

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

Similar routines are provided for all address registers (**edi** - **eax**): **edi( )** - **eax( )**.

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

RETURNS

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

SEE ALSO

dbgArchLib, *VxWorks Programmer's Guide: Debugging*

eflags( )

NAME

eflags( ) – return the contents of the status register (x86/SimNT)

SYNOPSIS

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

DESCRIPTION

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

RETURNS

The contents of the status register.

SEE ALSO

dbgArchLib, *VxWorks Programmer's Guide: Debugging*

E

endFindByName( )

NAME **endFindByName( )** – find a device using its string name

SYNOPSIS

```
END_OBJ * endFindByName
    (
    char * pName,              /* device name to search for */
    int    unit
    )
```

DESCRIPTION This routine takes a string name and a unit number and finds the device that has that name/unit combination.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call **endFindByName( )** from within the kernel protection domain only, and the data referenced in the *pName* parameter must reside in the kernel protection domain. In addition, the returned **END_OBJ** is valid in the kernel protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS A pointer to an **END_OBJ**; or **NULL**, if the device is not found.

SEE ALSO **muxLib**

envLibInit( )

NAME **envLibInit( )** – initialize environment variable facility

SYNOPSIS

```
STATUS envLibInit
    (
    BOOL installHooks
    )
```

DESCRIPTION If *installHooks* is **TRUE**, task create and delete hooks are installed that will optionally create and destroy private environments for the task being created or destroyed, depending on the state of **VX_PRIVATE_ENV** in the task options word. If *installHooks* is **FALSE** and a task requires a private environment, it is the application's responsibility to create and destroy the private environment, using **envPrivateCreate( )** and **envPrivateDestroy( )**.

RETURNS **OK**, or **ERROR** if an environment cannot be allocated or the hooks cannot be installed.

SEE ALSO **envLib**

envPrivateCreate( )

NAME **envPrivateCreate( )** – create a private environment

SYNOPSIS

```
STATUS envPrivateCreate
    (
    int taskId,          /* task to have private environment */
    int envSource        /* -1 = make an empty private environment 0 */
                         /* = copy global env to new private env taskId */
                         /* = copy the specified task's env */
    )
```

DESCRIPTION This routine creates a private set of environment variables for a specified task, if the environment variable task create hook is not installed.

RETURNS **OK**, or **ERROR** if memory is insufficient.

SEE ALSO **envLib**, **envLibInit( )**, **envPrivateDestroy( )**

envPrivateDestroy( )

NAME **envPrivateDestroy( )** – destroy a private environment

SYNOPSIS

```
STATUS envPrivateDestroy
    (
    int taskId                /* task with private env to destroy */
    )
```

DESCRIPTION This routine destroys a private set of environment variables that were created with **envPrivateCreate( )**. Calling this routine is unnecessary if the environment variable task create hook is installed and the task was spawned with **VX_PRIVATE_ENV**.

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

SEE ALSO **envLib**, **envPrivateCreate( )**

envShow( )

NAME **envShow( )** – display the environment for a task

SYNOPSIS

```
void envShow
    (
    int taskId                /* task for which environment is printed */
    )
```

DESCRIPTION This routine prints to standard output all the environment variables for a specified task. If *taskId* is **NULL**, then the calling task's environment is displayed.

RETURNS N/A

SEE ALSO **envLib**

errnoGet( )

NAME **errnoGet( )** – get the error status value of the calling task

SYNOPSIS

```
int errnoGet (void)
```

DESCRIPTION This routine gets the error status stored in **errno**. It is provided for compatibility with previous versions of VxWorks and simply accesses **errno** directly.

RETURNS The error status value contained in **errno**.

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, and the value currently in **errno** is returned.

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.

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 for a specified task. If *taskId* is zero, the calling task is assumed, and **errno** is set with the specified error status.

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.

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 **errno** variable with a specified error status. It is provided for compatibility with previous versions of VxWorks and simply accesses **errno** directly.

RETURNS **OK**, or **ERROR** if the interrupt nest level is too deep.

SEE ALSO **errnoLib**, **errnoGet( )**, **errnoOfTaskSet( )**

etherMultiAdd( )

NAME **etherMultiAdd( )** – add multicast address to a multicast address list

SYNOPSIS

```
int etherMultiAdd
    (
    LIST * pList,             /* pointer to list of multicast addresses */
    char*  pAddress           /* address you want to add to list */
    )
```

DESCRIPTION This routine adds an Ethernet multicast address list for a given END. The address is a six-byte value pointed to by *pAddress*.

RETURNS **OK** or **ENETRESET**.

SEE ALSO **etherMultiLib**

etherMultiDel( )

NAME

etherMultiDel( ) – delete an Ethernet multicast address record

SYNOPSIS

```
int etherMultiDel
    (
    LIST * pList,             /* pointer to list of multicast addresses */
    char*  pAddress           /* address you want to add to list */
    )
```

DESCRIPTION

This routine deletes an Ethernet multicast address from the list. The address is a six-byte value pointed to by *pAddress*.

RETURNS

OK or **ENETRESET**.

SEE ALSO

etherMultiLib

etherMultiGet( )

NAME

etherMultiGet( ) – retrieve a table of multicast addresses from a driver

SYNOPSIS

```
int etherMultiGet
    (
    LIST*        pList,       /* pointer to list of multicast addresses */
    MULTI_TABLE* pTable       /* table into which to copy addresses */
    )
```

DESCRIPTION

This routine runs down the multicast address list stored in a driver and places all the entries it finds into the multicast table structure passed to it.

RETURNS

OK or **ERROR**.

SEE ALSO

etherMultiLib

eventClear( )

NAME

eventClear( ) – clear all events for current 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( ) – wait for event(s)

SYNOPSIS

```
STATUS eventReceive
    (
    UINT32   events,          /* events task is waiting to occur */
    UINT8    options,         /* user options */
    int      timeout,         /* ticks to wait */
    UINT32 * pEventsReceived  /* events occured are returned through this */
    )
```

DESCRIPTION

Pends task until one or all specified *events* have occurred. When they have, *pEventsReceived* will be filled with those that did occur.

The *options* parameter is used for three 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. Performing a bitwise-OR of the following:

EVENTS_RETURN_ALL (0x2)
causes the function to return received events, both wanted and unwanted.

Thirdly, it can be used to retrieve what events have been received by the current task. If the option

EVENTS_FETCH (0x80)
is chosen by the user, 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.

It must also be noted that events sent to the receiving task are cleared prior to returning, as if a call to **eventClear( )** was done.

The parameter *pEventsReceived* is always filled with the events received even when the function returns an error, except if a value of **NULL** was passed.

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_objLib_OBJ_DELETED
Task is waiting for some events from a resource that is subsequently deleted.

S_intLib_NOT_ISR_CALLABLE
Function has been called from ISR.

S_eventLib_ZERO_EVENTS
The *events* parameter has been passed a value of 0.

SEE ALSO

eventLib, **semEvLib**, **msgQEvLib**, **eventSend( )**

E

eventSend( )

NAME **eventSend( )** – send event(s)

SYNOPSIS

```
STATUS eventSend
    (
    int    taskId,             /* task events will be sent to */
    UINT32 events              /* events to send */
    )
```

DESCRIPTION Sends specified event(s) to specified task. Passing a taskId of **NULL** sends events to the calling task.

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( )**

excConnect( )

NAME **excConnect( )** – connect a C routine to an exception vector (PowerPC)

SYNOPSIS

```
STATUS excConnect
    (
    VOIDFUNCPTR * vector,     /* exception vector to attach to */
    VOIDFUNCPTR   routine     /* routine to be called */
    )
```

DESCRIPTION This routine connects a specified C routine to a specified exception vector. An exception stub is created and in placed at *vector* in the exception table. The address of *routine* is stored in the exception stub code. When an exception occurs, the processor jumps to the exception stub code, saves the registers, and calls the C routines.

The routine can be any normal C code, except that it must not invoke certain operating system functions that may block or perform I/O operations.

The registers are saved to an Exception Stack Frame (ESF) placed on the stack of the task that has produced the exception. The structure of the ESF used to save the registers is defined in **h/arch/ppc/esfPpc.h**.

The only argument passed by the exception stub to the C routine is a pointer to the ESF containing the registers values. The prototype of this C routine is described below:

```
void excHandler (ESFPPC *);
```

When the C routine returns, the exception stub restores the registers saved in the ESF and continues execution of the current task.

RETURNS **OK**, always.

SEE ALSO **excArchLib**, **excIntConnect( )**, **excVecSet( )**

excCrtConnect( )

NAME **excCrtConnect( )** – connect a C routine to a critical exception vector (PowerPC 403)

SYNOPSIS

```
STATUS excCrtConnect
    (
    VOIDFUNCPTR * vector,     /* exception vector to attach to */
    VOIDFUNCPTR   routine     /* routine to be called */
    )
```

DESCRIPTION This routine connects a specified C routine to a specified critical exception vector. An exception stub is created and in placed at *vector* in the exception table. The address of *routine* is stored in the exception stub code. When an exception occurs, the processor jumps to the exception stub code, saves the registers, and call the C routines.

The routine can be any normal C code, except that it must not invoke certain operating system functions that may block or perform I/O operations.

The registers are saved to an Exception Stack Frame (ESF) which is placed on the stack of the task that has produced the exception. The ESF structure is defined in **h/arch/ppc/esfPpc.h**.

The only argument passed by the exception stub to the C routine is a pointer to the ESF containing the register values. The prototype of this C routine is as follows:

```
void excHandler (ESFPPC *);
```

When the C routine returns, the exception stub restores the registers saved in the ESF and continues execution of the current task.

RETURNS **OK**, always.

SEE ALSO **excArchLib**, **excIntConnect( )**, **excIntCrtConnect( )**, **excVecSet( )**

excHookAdd( )

NAME **excHookAdd( )** – specify a routine to be called with exceptions

SYNOPSIS

```
void excHookAdd
    (
    FUNCPTR excepHook         /* routine to call when exceptions occur */
    )
```

DESCRIPTION This routine specifies a routine that will be called when hardware exceptions occur. The specified routine is called after normal exception handling, which includes displaying information about the error. Upon return from the specified routine, the task that incurred the error is suspended.

The exception handling routine should be declared as:

```
void myHandler
    (
    int     task,    /* ID of offending task             */
    int     vecNum,  /* exception vector number          */
    ESFxx * pEsf     /* pointer to exception stack frame */
    )
```

where *task* is the ID of the task that was running when the exception occurred. *ESFxx* is architecture-specific and can be found by examining **/target/h/arch/***arch***/esf***arch***.h**; for example, the PowerPC uses ESFPPC.

This facility is normally used by **dbgLib( )** to activate its exception handling mechanism. If an application provides its own exception handler, it will supersede the **dbgLib** mechanism.

RETURNS N/A

SEE ALSO **excLib**, **excTask( )**

excInit( )

NAME **excInit( )** – initialize the exception handling package

SYNOPSIS

```
STATUS excInit (void)
```

DESCRIPTION This routine installs the exception handling facilities and spawns **excTask( )**, which performs special exception handling functions that need to be done at task level. It also creates the message queue used to communicate with **excTask( )**.

NOTE: The exception handling facilities should be installed as early as possible during system initialization in the root task, **usrRoot( )**, in **usrConfig.c**.

RETURNS **OK**, or **ERROR** if a message queue cannot be created or **excTask( )** cannot be spawned.

SEE ALSO **excLib**, **excTask( )**

excIntConnect( )

NAME **excIntConnect( )** – connect a C routine to an asynchronous exception vector (PowerPC, ARM)

SYNOPSIS

```
STATUS excIntConnect
    (
    VOIDFUNCPTR * vector,     /* exception vector to attach to */
    VOIDFUNCPTR   routine     /* routine to be called */
    )
```

DESCRIPTION This routine connects a specified C routine to a specified asynchronous exception vector.

When the C routine is invoked, interrupts are still locked. It is the responsibility of the C routine to re-enable the interrupt.

The routine can be any normal C code, except that it must not invoke certain operating system functions that may block or perform I/O operations.

NOTE: On PowerPC, the vector is typically the external interrupt vector 0x500 and the decrementer vector 0x900. An interrupt stub is created and placed at *vector* in the exception table. The address of *routine* is stored in the interrupt stub code. When the asynchronous exception occurs the processor jumps to the interrupt stub code, saves only the requested registers, and calls the C routines. Before saving the requested registers, the interrupt stub switches from the current task stack to the interrupt stack. For nested interrupts, no stack-switching is performed, because the interrupt is already set.

NOTE: On the ARM, the address of *routine* is stored in a function pointer to be called by the stub installed on the IRQ exception vector following an asynchronous exception. This routine is responsible for determining the interrupt source and despatching the correct handler for that source. Before calling the routine, the interrupt stub switches to SVC mode, changes to a separate interrupt stack and saves necessary registers. In the case of a nested interrupt, no SVC stack switch occurs.

RETURNS **OK**, always.

SEE ALSO **excArchLib**, **excConnect( )**, **excVecSet( )**

excIntCrtConnect( )

NAME **excIntCrtConnect( )** – connect a C routine to a critical interrupt vector (PowerPC 403)

SYNOPSIS

```
STATUS excIntCrtConnect
    (
    VOIDFUNCPTR * vector,     /* exception vector to attach to */
    VOIDFUNCPTR   routine     /* routine to be called */
    )
```

DESCRIPTION This routine connects a specified C routine to a specified asynchronous critical exception vector such as the critical external interrupt vector (0x100), or the watchdog timer vector (0x1020). An interrupt stub is created and placed at *vector* in the exception table. The address of *routine* is stored in the interrupt stub code. When the asynchronous exception occurs, the processor jumps to the interrupt stub code, saves only the requested registers, and calls the C routines.

When the C routine is invoked, interrupts are still locked. It is the C routine's responsibility to re-enable interrupts.

The routine can be any normal C routine, except that it must not invoke certain operating system functions that may block or perform I/O operations.

Before the requested registers are saved, the interrupt stub switches from the current task stack to the interrupt stack. In the case of nested interrupts, no stack switching is performed, because the interrupt stack is already set.

RETURNS **OK**, always.

SEE ALSO **excArchLib**, **excConnect( )**, **excCrtConnect( )**, **excVecSet( )**

excTask( )

NAME **excTask( )** – handle task-level exceptions

SYNOPSIS

```
void excTask ()
```

DESCRIPTION This routine is spawned as a task by **excInit( )** to perform functions that cannot be performed at interrupt or trap level. It has a priority of 0. Do not suspend, delete, or change the priority of this task.

RETURNS N/A

SEE ALSO **excLib**, **excInit( )**

excVecGet( )

NAME **excVecGet( )** – get a CPU exception vector (PowerPC, ARM)

SYNOPSIS

```
FUNCPTR excVecGet
    (
    FUNCPTR * vector            /* vector offset */
    )
```

DESCRIPTION This routine returns the address of the C routine currently connected to *vector*.

RETURNS The address of the C routine.

SEE ALSO **excArchLib**, **excVecSet( )**

excVecInit( )

NAME **excVecInit( )** – initialize the exception/interrupt vectors

SYNOPSIS

```
STATUS excVecInit (void)
```

DESCRIPTION This routine sets all exception vectors to point to the appropriate default exception handlers. These handlers will safely trap and report exceptions caused by program errors or unexpected hardware interrupts.

MC680x0:
All vectors from vector 2 (address 0x0008) to 255 (address 0x03fc) are initialized. Vectors 0 and 1 contain the reset stack pointer and program counter.

x86:
All vectors from vector 0 (address (0x0000) to 255 (address 0x07f8) are initialized to default handlers.

MIPS:
All MIPS exception, trap, and interrupt vectors are set to default handlers.

x86:
All vectors from vector 0 (address (0x0000) to 255 (address 0x07f8) are initialized to default handlers.

PowerPC:
There are 48 vectors and only vectors that are used are initialized.

SH:
There are 256 vectors, initialized with the default exception handler (for exceptions) or the uninitialized interrupt handler (for interrupts). On SH-2, vectors 0 and 1 contain the power-on reset program counter and stack pointer. Vectors 2 and 3 contain the manual reset program counter and stack pointer. On SH-3 and SH-4 processors the vector table is located at (vbr + 0x800), and the (exception code / 8) value is used as vector offset. The first two vectors are reserved for special use: "trapa #0" (offset 0x0) to implement software breakpoint, and "trapa #1" (offset 0x4) to detect integer zero divide exception.

ARM:
All exception vectors are initialized to default handlers except 0x14 (Address) which is now reserved on the ARM and 0x1C (FIQ), which is not used by VxWorks.

SimSolaris/SimNT:
This routine does nothing on both simulators and always returns **OK**.

NOTE: This routine is usually called from the system start-up routine, **usrInit( )**, in **usrConfig.c**. It must be called before interrupts are enabled.

RETURNS **OK**, always.

SEE ALSO **excArchLib**, **excLib**

excVecSet( )

NAME **excVecSet( )** – set a CPU exception vector (PowerPC, ARM)

SYNOPSIS

```
void excVecSet
    (
    FUNCPTR * vector,         /* vector offset */
    FUNCPTR   function        /* address to place in vector */
    )
```

DESCRIPTION This routine specifies the C routine that will be called when the exception corresponding to *vector* occurs. This routine does not create the exception stub; it simply replaces the C routine to be called in the exception stub.

NOTE: On the ARM, there is no **excConnect( )** routine, unlike the PowerPC. The C routine is attached to a default stub using **excVecSet( )**.

RETURNS N/A

SEE ALSO **excArchLib**, **excVecGet( )**, **excConnect( )**, **excIntConnect( )**

exit( )

NAME **exit( )** – exit a task (ANSI)

SYNOPSIS

```
void exit
    (
    int code                  /* code stored in TCB for delete hooks */
    )
```

DESCRIPTION This routine is called by a task to cease to exist as a task. It is called implicitly when the "main" routine of a spawned task is exited. The *code* parameter will be stored in the **WIND_TCB** for possible use by the delete hooks, or post-mortem debugging.

ERRNO N/A

SEE ALSO **taskLib**, **taskDelete( )**, *American National Standard for Information Systems -Programming Language - C, ANSI X3.159-1989: Input/Output (**stdlib.h**), VxWorks Programmer's Guide: Basic OS*

exp( )

NAME **exp( )** – compute an exponential value (ANSI)

SYNOPSIS

```
double exp
    (
    double x                  /* exponent */
    )
```

DESCRIPTION This routine returns the exponential value of *x* in double precision (IEEE double, 53 bits).

A range error occurs if *x* is too large.

INCLUDE FILES math.h

RETURNS The double-precision exponential value of *x*.

Special cases:
  If *x* is +INF or NaN, **exp( )** returns *x*.
  If *x* is -INF, it returns 0.

SEE ALSO **ansiMath**, **mathALib**

expf( )

NAME **expf( )** – compute an exponential value (ANSI)

SYNOPSIS

```
float expf
    (
    float x   /* exponent */
    )
```

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

INCLUDE FILES **math.h**

RETURNS The single-precision exponential value of *x*.

SEE ALSO **mathALib**

fabs( )

NAME
fabs( ) – compute an absolute value (ANSI)

SYNOPSIS

```
double fabs
    (
    double v    /* number to return the absolute value of */
    )
```

DESCRIPTION
This routine returns the absolute value of *v* in double precision.

INCLUDE FILES
math.h

RETURNS
The double-precision absolute value of *v*.

ERRNO
EDOM, **ERANGE**

SEE ALSO
ansiMath, **mathALib**

fabsf( )

NAME
fabsf( ) – compute an absolute value (ANSI)

SYNOPSIS

```
float fabsf
    (
    float v   /* number to return the absolute value of */
    )
```

DESCRIPTION
This routine returns the absolute value of *v* in single precision.

INCLUDE FILES
math.h

RETURNS
The single-precision absolute value of *v*.

SEE ALSO
mathALib

F

fclose( )

NAME **fclose( )** – close a stream (ANSI)

SYNOPSIS

```
int fclose
    (
    FILE * fp                  /* stream to close */
    )
```

DESCRIPTION This routine flushes a specified stream and closes the associated file. Any unwritten buffered data is delivered to the host environment to be written to the file; any unread buffered data is discarded. The stream is disassociated from the file. If the associated buffer was allocated automatically, it is deallocated.

INCLUDE FILES **stdio.h**

RETURNS Zero if the stream is closed successfully, or EOF if errors occur.

ERRNO **EBADF**

SEE ALSO **ansiStdio**, **fflush( )**

fdopen( )

NAME **fdopen( )** – open a file specified by a file descriptor (POSIX)

SYNOPSIS

```
FILE * fdopen
    (
    int          fd,           /* file descriptor */
    const char * mode          /* mode to open with */
    )
```

DESCRIPTION This routine opens the file specified by the file descriptor *fd* and associates a stream with it. The *mode* argument is used just as in the **fopen( )** function.

INCLUDE FILES **stdio.h**

RETURNS A pointer to a stream, or a null pointer if an error occurs, with **errno** set to indicate the error.

ERRNO **EINVAL**

SEE ALSO **ansiStdio**, **fopen( )**, **freopen( )**, *Information Technology - POSIX - Part 1: System API [C Language], IEEE Std 1003.1*

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.

SEE ALSO

fioLib, **printf( )**

feof( )

NAME

feof( ) – test the end-of-file indicator for a stream (ANSI)

SYNOPSIS

```
int feof
    (
    FILE * fp                 /* stream to test */
    )
```

DESCRIPTION

This routine tests the end-of-file indicator for a specified stream.

INCLUDE FILES

stdio.h

RETURNS

Non-zero if the end-of-file indicator is set for *fp*.

SEE ALSO

ansiStdio, **clearerr( )**

F

ferror( )

NAME **ferror( )** – test the error indicator for a file pointer (ANSI)

SYNOPSIS

```
int ferror
    (
    FILE * fp                   /* stream to test */
    )
```

DESCRIPTION This routine tests the error indicator for the stream pointed to by *fp*.

INCLUDE FILES **stdio.h**

RETURNS Non-zero if the error indicator is set for *fp*.

SEE ALSO **ansiStdio**, **clearerr( )**

fflush( )

NAME **fflush( )** – flush a stream (ANSI)

SYNOPSIS

```
int fflush
    (
    FILE * fp                   /* stream to flush */
    )
```

DESCRIPTION This routine writes to the file any unwritten data for a specified output or update stream for which the most recent operation was not input; for an input stream the behavior is undefined.

WARNING: ANSI specifies that if *fp* is a null pointer, **fflush( )** performs the flushing action on all streams for which the behavior is defined; however, this is not implemented in VxWorks.

INCLUDE FILES **stdio.h**

RETURNS Zero, or EOF if a write error occurs.

ERRNO **EBADF**

SEE ALSO **ansiStdio**, **fclose( )**

fgetc( )

NAME

fgetc( ) – return the next character from a stream (ANSI)

SYNOPSIS

```
int fgetc
    (
    FILE * fp                    /* stream to read from */
    )
```

DESCRIPTION

This routine returns the next character (converted to an **int**) from the specified stream, and advances the file position indicator for the stream.

If the stream is at end-of-file, the end-of-file indicator for the stream is set; if a read error occurs, the error indicator is set.

INCLUDE FILES

stdio.h

RETURNS

The next character from the stream, or EOF if the stream is at end-of-file or a read error occurs.

SEE ALSO

ansiStdio, **fgets( )**, **getc( )**

fgetpos( )

NAME

fgetpos( ) – store the current value of the file position indicator for a stream (ANSI)

SYNOPSIS

```
int fgetpos
    (
    FILE *   fp,                 /* stream */
    fpos_t * pos                 /* where to store position */
    )
```

DESCRIPTION

This routine stores the current value of the file position indicator for a specified stream *fp* in the object pointed to by *pos*. The value stored contains unspecified information usable by **fsetpos( )** for repositioning the stream to its position at the time **fgetpos( )** was called.

INCLUDE FILES

stdio.h

RETURNS

Zero, or non-zero if unsuccessful, with **errno** set to indicate the error.

SEE ALSO

ansiStdio, **fsetpos( )**

fgets( )

NAME

fgets( ) – read a specified number of characters from a stream (ANSI)

SYNOPSIS

```
char * fgets
    (
    char * buf,              /* where to store characters */
    size_t n,                /* no. of bytes to read + 1 */
    FILE * fp                /* stream to read from */
    )
```

DESCRIPTION

This routine stores in the array *buf* up to *n*-1 characters from a specified stream. No additional characters are read after a new-line or end-of-line. A null character is written immediately after the last character read into the array.

If end-of-file is encountered and no characters have been read, the contents of the array remain unchanged. If a read error occurs, the array contents are indeterminate.

INCLUDE FILES

stdio.h

RETURNS

A pointer to *buf*, or a null pointer if an error occurs or end-of-file is encountered and no characters have been read.

SEE ALSO

ansiStdio, **fread( )**, **fgetc( )**

fileno( )

NAME

fileno( ) – return the file descriptor for a stream (POSIX)

SYNOPSIS

```
int fileno
    (
    FILE * fp                /* stream */
    )
```

DESCRIPTION

This routine returns the file descriptor associated with a specified stream.

INCLUDE FILES

stdio.h

RETURNS

The file descriptor, or -1 if an error occurs, with **errno** set to indicate the error.

SEE ALSO

ansiStdio, *Information Technology - POSIX - Part 1: System API [C Language], IEEE Std 1003.1*

fileUploadPathClose( )

NAME

fileUploadPathClose( ) – close the event-destination file (WindView)

SYNOPSIS

```
void fileUploadPathClose
    (
    UPLOAD_ID pathId          /* generic upload-path descriptor */
    )
```

DESCRIPTION

This routine closes the file associated with *pathId* that is serving as a destination for event data.

RETURNS

N/A

SEE ALSO

wvFileUploadPathLib, **fileUploadPathCreate( )**

fileUploadPathCreate( )

NAME

fileUploadPathCreate( ) – create a file for depositing event data (Windview)

SYNOPSIS

```
UPLOAD_ID fileUploadPathCreate
    (
    char * fname,             /* name of file to create */
    int    openFlags          /* O_CREAT, O_TRUNC */
    )
```

DESCRIPTION

This routine opens and initializes a file to receive uploaded events. The *openFlags* argument is passed on as the flags argument to the actual open call so that the caller can specify things like **O_TRUNC** and **O_CREAT**. The file is always opened as **O_WRONLY**, regardless of the value of *openFlags*.

RETURNS

The **UPLOAD_ID**, or **NULL** if the file can not be opened or memory for the ID is not available.

SEE ALSO

wvFileUploadPathLib, **fileUploadPathClose( )**

fileUploadPathLibInit( )

NAME **fileUploadPathLibInit( )** – initialize the **wvFileUploadPathLib** library (Windview)

SYNOPSIS

```
STATUS fileUploadPathLibInit (void)
```

DESCRIPTION This routine initializes the library by pulling in the routines in this file for use with WindView. It is called during system configuration from **usrWindview.c**.

RETURNS **OK**.

SEE ALSO **wvFileUploadPathLib**

F

fileUploadPathWrite( )

NAME **fileUploadPathWrite( )** – write to the event-destination file (WindView)

SYNOPSIS

```
int fileUploadPathWrite
    (
    UPLOAD_ID pathId,          /* generic upload-path descriptor */
    char *    pStart,          /* address of data to write */
    size_t    size             /* number of bytes of data at pStart */
    )
```

DESCRIPTION This routine writes *size* bytes of data beginning at *pStart* to the file indicated by *pathId*.

RETURNS The number of bytes written, or **ERROR**.

SEE ALSO **wvFileUploadPathLib**

fioFormatV( )

NAME

fioFormatV( ) – convert a format string

SYNOPSIS

```
int fioFormatV
    (
    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**.

SEE ALSO

fioLib

fioLibInit( )

NAME **fioLibInit( )** – initialize the formatted I/O support library

SYNOPSIS

```
void fioLibInit (void)
```

DESCRIPTION This routine initializes the formatted I/O support library. It should be called once in **usrRoot( )** when formatted I/O functions such as **printf( )** and **scanf( )** are used.

RETURNS N/A

SEE ALSO **fioLib**

fioRdString( )

NAME **fioRdString( )** – read a string from a file

SYNOPSIS

```
int fioRdString
    (
    int  fd,                    /* fd of device to read */
    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.

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.

SEE ALSO **fioLib**, **read( )**

floatInit( )

NAME **floatInit( )** – initialize floating-point I/O support

SYNOPSIS

```
void floatInit (void)
```

DESCRIPTION This routine must be called if floating-point format specifications are to be supported by the **printf( )**/**scanf( )** family of routines. If the configuration macro **INCLUDE_FLOATING_POINT** is defined, it is called by the root task, **usrRoot( )**, in **usrConfig.c**.

RETURNS N/A

SEE ALSO **floatLib**

floor( )

NAME
floor( ) – compute the largest integer less than or equal to a specified value (ANSI)

SYNOPSIS
```
double floor
    (
    double v                  /* value to find the floor of */
    )
```

DESCRIPTION
This routine returns the largest integer less than or equal to *v*, in double precision.

INCLUDE FILES
math.h

RETURNS
The largest integral value less than or equal to *v*, in double precision.

SEE ALSO
ansiMath, **mathALib**

floorf( )

NAME
floorf( ) – compute the largest integer less than or equal to a specified value (ANSI)

SYNOPSIS
```
float floorf
    (
    float v    /* value to find the floor of */
    )
```

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

INCLUDE FILES
math.h

RETURNS
The largest integral value less than or equal to *v*, in single precision.

SEE ALSO
mathALib

fmod( )

NAME

fmod( ) – compute the remainder of x/y (ANSI)

SYNOPSIS

```
double fmod
    (
    double x,                 /* numerator */
    double y                  /* denominator */
    )
```

DESCRIPTION

This routine returns the remainder of *x/y* with the sign of *x*, in double precision.

INCLUDE FILES

math.h

RETURNS

The value *x - i * y*, for some integer *i*. If *y* is non-zero, the result has the same sign as *x* and magnitude less than the magnitude of *y*. If *y* is zero, **fmod( )** returns zero.

ERRNO

EDOM

SEE ALSO

ansiMath, **mathALib**

fmodf( )

NAME

fmodf( ) – compute the remainder of x/y (ANSI)

SYNOPSIS

```
float fmodf
    (
    float x,    /* numerator   */
    float y     /* denominator */
    )
```

DESCRIPTION

This routine returns the remainder of *x/y* with the sign of *x*, in single precision.

INCLUDE FILES

math.h

RETURNS

The single-precision modulus of *x/y*.

SEE ALSO

mathALib

fopen( )

NAME

fopen( ) – open a file specified by name (ANSI)

SYNOPSIS

```
FILE * fopen
    (
    const char * file,         /* name of file */
    const char * mode          /* mode */
    )
```


DESCRIPTION

This routine opens a file whose name is the string pointed to by *file*and associates a stream with it. The argument *mode* points to a string beginning with one of the following sequences:

r
: open text file for reading

w
: truncate to zero length or create text file for writing

a
: append; open or create text file for writing at end-of-file

rb
: open binary file for reading

wb
: truncate to zero length or create binary file for writing

ab
: append; open or create binary file for writing at end-of-file

r+
: open text file for update (reading and writing)

w+
: truncate to zero length or create text file for update.

a+
: append; open or create text file for update, writing at end-of-file

r+b / rb+
: open binary file for update (reading and writing)

w+b / wb+
: truncate to zero length or create binary file for update

a+b / ab+
: append; open or create binary file for update, writing at end-of-file

Opening a file with read mode (**r** as the first character in the *mode* argument) fails if the file does not exist or cannot be read.

Opening a file with append mode (**a** as the first character in the *mode* argument) causes all subsequent writes to the file to be forced to the then current end-of-file, regardless of intervening calls to **fseek( )**. In some implementations, opening a binary file with append mode (**b** as the second or third character in the *mode* argument) may initially position the file position indicator for the stream beyond the last data written, because of null character padding. In VxWorks, whether append mode is supported is device-specific.

When a file is opened with update mode (**+** as the second or third character in the *mode* argument), both input and output may be performed on the associated stream. However, output may not be directly followed by input without an intervening call to **fflush( )** or to a file positioning function (**fseek( )**, **fsetpos( )**, or **rewind( )**), and input may not be directly followed by output without an intervening call to a file positioning function, unless the input operation encounters end-of-file. Opening (or creating) a text file with update mode may instead open (or create) a binary stream in some implementations.

When opened, a stream is fully buffered if and only if it can be determined not to refer to an interactive device. The error and end-of-file indicators for the stream are cleared.

INCLUDE FILES **stdio.h**

RETURNS A pointer to the object controlling the stream, or a null pointer if the operation fails.

SEE ALSO **ansiStdio**, **fdopen( )**, **freopen( )**

fppInit( )

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

SYNOPSIS

```
void fppInit (void)
```

DESCRIPTION This routine initializes floating-point coprocessor support and must be called before using the floating-point coprocessor. This is done automatically by the root task, **usrRoot( )**, in **usrConfig.c** when the configuration macro **INCLUDE_HW_FP** is defined.

RETURNS N/A

SEE ALSO **fppLib**

fppProbe( )

NAME

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

SYNOPSIS

```
STATUS fppProbe (void)
```

DESCRIPTION

This routine determines whether there is a floating-point coprocessor in the system.

The implementation of this routine is architecture-dependent:

MC680x0, **x86**, **SH-4**:
This routine sets the illegal coprocessor opcode trap vector and executes a coprocessor instruction. If the instruction causes an exception, **fppProbe( )** returns **ERROR**. Note that this routine saves and restores the illegal coprocessor opcode trap vector that was there prior to this call.

The probe is only performed the first time this routine is called. The result is stored in a static and returned on subsequent calls without actually probing.

MIPS:
This routine simply reads the R-Series status register and reports the bit that indicates whether coprocessor 1 is usable. This bit must be correctly initialized in the BSP.

ARM:
This routine currently returns **ERROR** to indicate no floating-point coprocessor support.

SimNT, **SimSolaris**:
This routine currently returns **OK**.

RETURNS

OK, or **ERROR** if there is no floating-point coprocessor.

SEE ALSO

fppArchLib

fppRestore( )

NAME

fppRestore( ) – restore the floating-point coprocessor context

SYNOPSIS

```
void fppRestore
    (
    FP_CONTEXT * pFpContext   /* where to restore context from */
    )
```

DESCRIPTION This routine restores the floating-point coprocessor context. The context restored is:

MC680x0:
- registers **fpcr**, **fpsr**, and **fpiar**
- registers **f0** - **f7**
- internal state frame (if **NULL**, the other registers are not saved.)

MIPS:
- register **fpcsr**
- registers **fp0** - **fp31**

SH-4:
- registers **fpcsr** and **fpul**
- registers **fr0** - **fr15**
- registers **xf0** - **xf15**

x86:

108 byte old context with fsave and frstor instruction

- control word, status word, tag word,
- instruction pointer,
- instruction pointer selector,
- last FP instruction op code,
- data pointer,
- data pointer selector,
- registers **st/mm0** - **st/mm7** (10 bytes * 8)

512 byte new context with fxsave and fxrstor instruction

- control word, status word, tag word,
- last FP instruction op code,
- instruction pointer,
- instruction pointer selector,
- data pointer,
- data pointer selector,
- registers **st/mm0** - **st/mm7** (10 bytes * 8)
- registers **xmm0** - **xmm7** (16 bytes * 8)

ARM:
- currently, on this architecture, this routine does nothing.

SimSolaris:
- register **fsr**
- registers **f0** - **f31**

SimNT:
- this routine does nothing on Windows simulator.

RETURNS N/A

SEE ALSO **fppArchLib**, **fppSave( )**

fppSave( )

NAME **fppSave( )** – save the floating-point coprocessor context

SYNOPSIS

```
void fppSave
    (
    FP_CONTEXT * pFpContext    /* where to save context */
    )
```

F

DESCRIPTION This routine saves the floating-point coprocessor context. The context saved is:

MC680x0:
- registers **fpcr**, **fpsr**, and **fpiar**
- registers **f0** - **f7**
- internal state frame (if **NULL**, the other registers are not saved.)

MIPS:
- register **fpcsr**
- registers **fp0** - **fp31**

SH-4:
- registers **fpcsr** and **fpul**
- registers **fr0** - **fr15**
- registers **xf0** - **xf15**

x86:

108 byte old context with fsave and frstor instruction

- control word, status word, tag word,
- instruction pointer,
- instruction pointer selector,
- last FP instruction op code,
- data pointer,
- data pointer selector,
- registers **st/mm0** - **st/mm7** (10 bytes * 8)

512 byte new context with fxsave and fxrstor instruction

- control word, status word, tag word,
- last FP instruction op code,
- instruction pointer,
- instruction pointer selector,
- data pointer,
- data pointer selector,
- registers **st/mm0** - **st/mm7** (10 bytes * 8)
- registers **xmm0** - **xmm7** (16 bytes * 8)

ARM:
- currently, on this architecture, this routine does nothing.

SimSolaris:
- register **fsr**
- registers **f0** - **f31**

SimNT:
- this routine does nothing on Windows simulator. Floating point registers are saved by Windows.

RETURNS N/A

SEE ALSO **fppArchLib**, **fppRestore( )**

fppShowInit( )

NAME **fppShowInit( )** – initialize the floating-point show facility

SYNOPSIS

```
void fppShowInit (void)
```

DESCRIPTION This routine links the floating-point show facility into the VxWorks system. It is called automatically when the floating-point show facility is configured into VxWorks using either of the following methods:

– If you use the configuration header files, define

INCLUDE_SHOW_ROUTINES in **config.h**.

– If you use the Tornado project facility, select **INCLUDE_HW_FP_SHOW**.

RETURNS N/A

SEE ALSO **fppShow**

fppTaskRegsGet( )

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

SYNOPSIS

```
STATUS fppTaskRegsGet
    (
    int         task,         /* task to get info about */
    FPREG_SET * pFpRegSet     /* ptr to floating-point register set */
    )
```

DESCRIPTION This routine copies a task's floating-point registers and/or status registers to the locations whose pointers are passed as parameters. The floating-point registers are copied into an array containing all the registers.

NOTE: This routine only works well if *task* is not the calling task. If a task tries to discover its own registers, the values will be stale (that is, left over from the last task switch).

RETURNS **OK**, or **ERROR** if there is no floating-point support or there is an invalid state.

SEE ALSO **fppArchLib**, **fppTaskRegsSet( )**

fppTaskRegsSet( )

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

SYNOPSIS

```
STATUS fppTaskRegsSet
    (
    int         task,         /* task to set registers for */
    FPREG_SET * pFpRegSet     /* ptr to floating-point register set */
    )
```

DESCRIPTION This routine loads the specified values into the TCB of a specified task. The register values are copied from the array at *pFpRegSet*.

RETURNS **OK**, or **ERROR** if there is no floating-point support or there is an invalid state.

SEE ALSO **fppArchLib**, **fppTaskRegsGet( )**

fppTaskRegsShow( )

NAME **fppTaskRegsShow( )** – print the contents of a task's floating-point registers

SYNOPSIS

```
void fppTaskRegsShow
    (
    int task              /* task to display floating point registers for */
    )
```

DESCRIPTION This routine prints to standard output the contents of a task's floating-point registers.

RETURNS N/A

SEE ALSO **fppShow**

fprintf( )

NAME **fprintf( )** – write a formatted string to a stream (ANSI)

SYNOPSIS

```
int fprintf
    (
    FILE *       fp,          /* stream to write to */
    const char * fmt,         /* format string */
                 ...          /* optional arguments to format string */
    )
```

DESCRIPTION This routine writes output to a specified stream under control of the string *fmt*. The string *fmt* contains ordinary characters, which are written unchanged, plus conversion specifications, which cause the arguments that follow *fmt* to be converted and printed as part of the formatted string.

The number of arguments for the format is arbitrary, but they must correspond to the conversion specifications in *fmt*. If there are insufficient arguments, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored. The routine returns when the end of the format string is encountered.

The format is a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: ordinary multibyte characters (not %) that are copied unchanged to the output stream; and conversion specification, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the **%** character. After the **%**, the following appear in sequence:

– Zero or more flags (in any order) that modify the meaning of the conversion specification.

– An optional minimum field width. If the converted value has fewer characters than the field width, it will be padded with spaces (by default) on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of an asterisk (*) (described later) or a decimal integer.

– An optional precision that gives the minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, and **X** conversions, the number of digits to appear after the decimal-point character for **e**, **E**, and **f** conversions, the maximum number of significant digits for the **g** and **G** conversions, or the maximum number of characters to be written from a string in the **s** conversion. The precision takes the form of a period (**.**) followed either by an asterisk (*) (described later) or by an optional decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.

– An optional **h** specifying that a following **d**, **i**, **o**, **u**, **x**, and **X** conversion specifier applies to a **short int** or **unsigned short int** argument (the argument will have been promoted according to the integral promotions, and its value converted to **short int** or **unsigned short int** before printing); an optional **h** specifying that a following **n** conversion specifier applies to a pointer to a **short int** argument; an optional **l** (el) specifying that a following **d**, **i**, **o**, **u**, **x**, and **X** conversion specifier applies to a **long int** or **unsigned long int** argument; or an optional **l** specifying that a following **n** conversion specifier applies to a pointer to a **long int** argument. If an **h** or **l** appears with any other conversion specifier, the behavior is undefined.

WARNING: ANSI C also specifies an optional **L** in some of the same contexts as **l** above, corresponding to a **long double** argument. However, the current release of the VxWorks libraries does not support **long double** data; using the optional **L** gives unpredictable results.

– A character that specifies the type of conversion to be applied.

As noted above, a field width, or precision, or both, can be indicated by an asterisk (*). In this case, an **int** argument supplies the field width or precision. The arguments specifying field width, or precision, or both, should appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a **-** flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

The flag characters and their meanings are:

-

The result of the conversion will be left-justified within the field. (it will be right-justified if this flag is not specified.)

+
: The result of a signed conversion will always begin with a plus or minus sign. (It will begin with a sign only when a negative value is converted if this flag is not specified.)

space
: If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space will be prefixed to the result. If the **space** and **+** flags both appear, the **space** flag will be ignored.

#
: The result is to be converted to an "alternate form." For **o** conversion it increases the precision to force the first digit of the result to be a zero. For **x** (or **X**) conversion, a non-zero result will have "0x" (or "0X") prefixed to it. For **e**, **E**, **f**, **g**, and **G** conversions, the result will always contain a decimal-point character, even if no digits follow it. (Normally, a decimal-point character appears in the result of these conversions only if no digit follows it). For **g** and **G** conversions, trailing zeros will not be removed from the result. For other conversions, the behavior is undefined.

0
: For **d**, **i**, **o**, **u**, **x**, **X**, **e**, **E**, **f**, **g**, and **G** conversions, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the **0** and **-**flags both appear, the **0** flag will be ignored. For **d**, **i**, **o**, **u**, **x**, and **X** conversions, if a precision is specified, the **0** flag will be ignored. For other conversions, the behavior is undefined.

The conversion specifiers and their meanings are:

d, **i**
: The **int** argument is converted to signed decimal in the style **[-]dddd**. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

o, **u**, **x**, **X**
: The **unsigned int** argument is converted to unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal notation (**x** or **X**) in the style **dddd**; the letters abcdef are used for **x** conversion and the letters ABCDEF for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

f
: The **double** argument is converted to decimal notation in the style **[-]ddd.ddd**, where the number of digits after the decimal point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the **#** flag is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is rounded to the

appropriate number of digits.

e, **E**

The **double** argument is converted in the style **[-]d.ddde+/-dd**, where there is one digit before the decimal-point character (which is non-zero if the argument is non-zero) and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero and the **#** flag is not specified, no decimal-point character appears. The value is rounded to the appropriate number of digits. The **E** conversion specifier will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits. If the value is zero, the exponent is zero.

g, **G**

The **double** argument is converted in style **f** or **e** (or in style **E** in the case of a **G** conversion specifier), with the precision specifying the number of significant digits. If the precision is zero, it is taken as 1. The style used depends on the value converted; style **e** (or **E**) will be used only if the exponent resulting from such a conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result; a decimal-point character appears only if it is followed by a digit.

c

The **int** argument is converted to an **unsigned char**, and the resulting character is written.

s

The argument should be a pointer to an array of character type. Characters from the array are written up to (but not including) a terminating null character; if the precision is specified, no more than that many characters are written. If the precision is not specified or is greater than the size of the array, the array will contain a null character.

p

The argument should be a pointer to **void**. The value of the pointer is converted to a sequence of printable characters, in hexadecimal representation (prefixed with "0x").

n

The argument should be a pointer to an integer into which the number of characters written to the output stream so far by this call to **fprintf( )** is written. No argument is converted.

%

A **%** is written. No argument is converted. The complete conversion specification is %%.

If a conversion specification is invalid, the behavior is undefined.

If any argument is, or points to, a union or an aggregate (except for an array of character type using **s** conversion, or a pointer using **p** conversion), the behavior is undefined.

In no case does a non-existent or small field width cause truncation of a field if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

INCLUDE FILES **stdio.h**

RETURNS The number of characters written, or a negative value if an output error occurs.

SEE ALSO **ansiStdio**, **printf( )**

fputc( )

NAME **fputc( )** – write a character to a stream (ANSI)

SYNOPSIS

```
int fputc
    (
    int    c,                 /* character to write */
    FILE * fp                 /* stream to write to */
    )
```

DESCRIPTION This routine writes a character *c* to a specified stream, at the position indicated by the stream's file position indicator (if defined), and advances the indicator appropriately.

If the file cannot support positioning requests, or if the stream was opened in append mode, the character is appended to the output stream.

INCLUDE FILES **stdio.h**

RETURNS The character written, or EOF if a write error occurs, with the error indicator set for the stream.

SEE ALSO **ansiStdio**, **fputs( )**, **putc( )**

fputs( )

NAME **fputs( )** – write a string to a stream (ANSI)

SYNOPSIS

```
int fputs
    (
    const char * s,           /* string */
    FILE *       fp           /* stream to write to */
    )
```

DESCRIPTION This routine writes the string *s*, minus the terminating **NULL** character, to a specified stream.

INCLUDE FILES **stdio.h**

RETURNS A non-negative value, or EOF if a write error occurs.

SEE ALSO **ansiStdio**, **fputc( )**

fread( )

NAME **fread( )** – read data into an array (ANSI)

SYNOPSIS

```
int fread
    (
    void * buf,               /* where to copy data */
    size_t size,              /* element size */
    size_t count,             /* no. of elements */
    FILE * fp                 /* stream to read from */
    )
```

DESCRIPTION This routine reads, into the array *buf*, up to *count* elements of size *size*, from a specified stream *fp*. The file position indicator for the stream (if defined) is advanced by the number of characters successfully read. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate. If a partial element is read, its value is indeterminate.

INCLUDE FILES **stdio.h**

RETURNS
The number of elements successfully read, which may be less than *count* if a read error or end-of-file is encountered; or zero if *size* or *count* is zero, with the contents of the array and the state of the stream remaining unchanged.

SEE ALSO
ansiStdio

free( )

NAME
free( ) – free a block of memory (ANSI)

SYNOPSIS

```
void free
    (
    void * ptr                /* pointer to block of memory to free */
    )
```

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

RETURNS
N/A

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

freopen( )

NAME
freopen( ) – open a file specified by name (ANSI)

SYNOPSIS

```
FILE * freopen
    (
    const char * file,        /* name of file */
    const char * mode,        /* mode */
    FILE *       fp           /* stream */
    )
```

DESCRIPTION
This routine opens a file whose name is the string pointed to by *file* and associates it with a specified stream *fp*. The *mode* argument is used just as in the **fopen( )** function.

This routine first attempts to close any file that is associated with the specified stream. Failure to close the file successfully is ignored. The error and end-of-file indicators for the stream are cleared.

Typically, **freopen( )** is used to attach the already-open streams **stdin**, **stdout**, and **stderr** to other files.

INCLUDE FILES **stdio.h**

RETURNS The value of *fp*, or a null pointer if the open operation fails.

SEE ALSO **ansiStdio**, **fopen( )**

frexp( )

NAME **frexp( )** – break a floating-point number into a normalized fraction and power of 2 (ANSI)

SYNOPSIS

```
double frexp
    (
    double value,            /* number to be normalized */
    int *  pexp              /* pointer to the exponent */
    )
```

DESCRIPTION This routine breaks a double-precision number *value* into a normalized fraction and integral power of 2. It stores the integer exponent in *pexp*.

INCLUDE FILES **math.h**

RETURNS The double-precision value *x*, such that the magnitude of *x* is in the interval [1/2,1) or zero, and *value* equals *x* times 2 to the power of *pexp*. If *value* is zero, both parts of the result are zero.

ERRNO EDOM

SEE ALSO **ansiMath**

fscanf( )

NAME

fscanf( ) – read and convert characters from a stream (ANSI)

SYNOPSIS

```
int fscanf
    (
    FILE *       fp,          /* stream to read from */
    char const * fmt,         /* format string */
                 ...          /* arguments to format string */
    )
```

DESCRIPTION

This routine reads characters from a specified stream, and interprets them according to format specifications in the string *fmt*, which specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

The format is a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: one or more white-space characters; an ordinary multibyte character (neither % nor a white-space character); or a conversion specification. Each conversion specification is introduced by the % character. After the %, the following appear in sequence:

– An optional assignment-suppressing character *.

– An optional non-zero decimal integer that specifies the maximum field width.

– An optional **h** or **l** (el) indicating the size of the receiving object. The conversion specifiers **d**, **i**, and **n** should be preceded by **h** if the corresponding argument is a pointer to **short int** rather than a pointer to **int**, or by **l** if it is a pointer to **long int**. Similarly, the conversion specifiers **o**, **u**, and **x** shall be preceded by **h** if the corresponding argument is a pointer to **unsigned short int** rather than a pointer to **unsigned int**, or by **l** if it is a pointer to **unsigned long int**. Finally, the conversion specifiers **e**, **f**, and **g** shall be preceded by **l** if the corresponding argument is a pointer to **double** rather than a pointer to **float**. If an **h** or **l** appears with any other conversion specifier, the behavior is undefined.

WARNING: ANSI C also specifies an optional **L** in some of the same contexts as **l** above, corresponding to a **long double** * argument. However, the current release of the VxWorks libraries does not support **long double** data; using the optional **L** gives unpredictable results.

– A character that specifies the type of conversion to be applied. The valid conversion specifiers are described below.

The **fscanf( )** routine executes each directive of the format in turn. If a directive fails, as detailed below, **fscanf( )** returns. Failures are described as input failures (due to the unavailability of input characters), or matching failures (due to inappropriate input).

A directive composed of white-space character(s) is executed by reading input up to the first non-white-space character (which remains unread), or until no more characters can be read.

A directive that is an ordinary multibyte character is executed by reading the next characters of the stream. If one of the characters differs from one comprising the directive, the directive fails, and the differing and subsequent characters remain unread.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

Input white-space characters (as specified by the **isspace( )** function) are skipped, unless the specification includes a **[**, **c**, or **n** specifier.

An input item is read from the stream, unless the specification includes an **n** specifier. An input item is defined as the longest matching sequence of input characters, unless that exceeds a specified field width, in which case it is the initial subsequence of that length in the sequence. The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails: this condition is a matching failure, unless an error prevented input from the stream, in which case it is an input failure.

Except in the case of a **%** specifier, the input item is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a *****, the result of the conversion is placed in the object pointed to by the first argument following the *fmt* argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

The following conversion specifiers are valid:

d

Matches an optionally signed decimal integer whose format is the same as expected for the subject sequence of the **strtol( )** function with the value 10 for the *base* argument. The corresponding argument should be a pointer to **int**.

i

Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the **strtol( )** function with the value 0 for the *base* argument. The corresponding argument should be a pointer to **int**.

o

Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the **strtoul( )** function with the value 8 for the *base* argument. The corresponding argument should be a pointer to **unsigned int**.

u
: Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **strtoul( )** function with the value 10 for the *base* argument. The corresponding argument should be a pointer to **unsigned int**.

x
: Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the **strtoul( )** function with the value 16 for the *base* argument. The corresponding argument should be a pointer to **unsigned int**.

e, **f**, **g**
: Match an optionally signed floating-point number, whose format is the same as expected for the subject string of the **strtod( )** function. The corresponding argument should be a pointer to **float**.

s
: Matches a sequence of non-white-space characters. The corresponding argument should be a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which will be added automatically.

[
: Matches a non-empty sequence of characters from a set of expected characters (the **scanset**). The corresponding argument should be a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which is added automatically. The conversion specifier includes all subsequent character in the format string, up to and including the matching right bracket (**]**). The characters between the brackets (the **scanlist**) comprise the scanset, unless the character after the left bracket is a circumflex (**^**) in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with "[]" or "[^]", the right bracket character is in the scanlist and the next right bracket character is the matching right bracket that ends the specification; otherwise the first right bracket character is the one that ends the specification.

c
: Matches a sequence of characters of the number specified by the field width (1 if no field width is present in the directive). The corresponding argument should be a pointer to the initial character of an array large enough to accept the sequence. No null character is added.

p
: Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the %p conversion of the **fprintf( )** function. The corresponding argument should be a pointer to a pointer to **void**. VxWorks defines its pointer input field to be consistent with pointers written by the **fprintf( )** function ("0x" hexadecimal notation). If the input item is a value converted

earlier during the same program execution, the pointer that results should compare equal to that value; otherwise the behavior of the %p conversion is undefined.

n

No input is consumed. The corresponding argument should be a pointer to **int** into which the number of characters read from the input stream so far by this call to **fscanf( )** is written. Execution of a %n directive does not increment the assignment count returned when **fscanf( )** completes execution.

%

Matches a single **%**; no conversion or assignment occurs. The complete conversion specification is %%.

If a conversion specification is invalid, the behavior is undefined.

The conversion specifiers **E**, **G**, and **X** are also valid and behave the same as **e**, **g**, and **x**, respectively.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (if any) is terminated with an input failure.

If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream. Trailing white space (including new-line characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the %n directive.

INCLUDE FILES **stdio.h**

RETURNS The number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure; or EOF if an input failure occurs before any conversion.

SEE ALSO **ansiStdio**, **scanf( )**, **sscanf( )**

fseek( )

NAME

fseek( ) – set the file position indicator for a stream (ANSI)

SYNOPSIS

```
int fseek
    (
    FILE * fp,                /* stream */
    long   offset,            /* offset from whence */
    int    whence             /* position to offset from: SEEK_SET = */
                              /* beginning SEEK_CUR = current position */
                              /* SEEK_END = end-of-file */
    )
```

DESCRIPTION

This routine sets the file position indicator for a specified stream. For a binary stream, the new position, measured in characters from the beginning of the file, is obtained by adding *offset* to the position specified by *whence*, whose possible values are:

SEEK_SET
the beginning of the file.

SEEK_CUR
the current value of the file position indicator.

SEEK_END
the end of the file.

A binary stream does not meaningfully support **fseek( )** calls with a *whence* value of **SEEK_END**.

For a text stream, either *offset* is zero, or *offset* is a value returned by an earlier call to **ftell( )** on the stream, in which case *whence* should be **SEEK_SET**.

A successful call to **fseek( )** clears the end-of-file indicator for the stream and undoes any effects of **ungetc( )** on the same stream. After an **fseek( )** call, the next operation on an update stream can be either input or output.

INCLUDE FILES

stdio.h

RETURNS

Non-zero only for a request that cannot be satisfied.

ERRNO

EINVAL

SEE ALSO

ansiStdio, **ftell( )**

F

fsetpos( )

NAME **fsetpos( )** – set the file position indicator for a stream (ANSI)

SYNOPSIS

```
int fsetpos
    (
    FILE *          iop,        /* stream */
    const fpos_t * pos          /* position, obtained by fgetpos() */
    )
```

DESCRIPTION This routine sets the file position indicator for a specified stream *iop* according to the value of the object pointed to by *pos*, which is a value obtained from an earlier call to **fgetpos( )** on the same stream.

A successful call to **fsetpos( )** clears the end-of-file indicator for the stream and undoes any effects of **ungetc( )** on the same stream. After an **fsetpos( )** call, the next operation on an update stream may be either input or output.

INCLUDE FILES **stdio.h**

RETURNS Zero, or non-zero if the call fails, with **errno** set to indicate the error.

SEE ALSO **ansiStdio**, **fgetpos( )**

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.

On return, fields in the **stat** structure are updated to reflect the characteristics of the file.

RETURNS

OK or **ERROR**.

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.

RETURNS

OK or **ERROR**.

SEE ALSO

dirLib, **statfs( )**, **ls( )**

ftell( )

NAME

ftell( ) – return the current value of the file position indicator for a stream (ANSI)

SYNOPSIS

```
long ftell
    (
    FILE * fp                 /* stream */
    )
```

DESCRIPTION

This routine returns the current value of the file position indicator for a specified stream. For a binary stream, the value is the number of characters from the beginning of the file. For a text stream, the file position indicator contains unspecified information, usable by

fseek( ) for returning the file position indicator to its position at the time of the **ftell( )** call; the difference between two such return values is not necessary a meaningful measure of the number of characters written or read.

INCLUDE FILES **stdio.h**

RETURNS The current value of the file position indicator, or -1L if unsuccessful, with **errno** set to indicate the error.

SEE ALSO **ansiStdio**, **fseek( )**

---

ftpCommand( )

NAME **ftpCommand( )** – send an FTP command and get the reply

SYNOPSIS

```
int ftpCommand
    (
    int    ctrlSock,          /* fd of control connection socket */
    char * fmt,               /* format string of command to send */
    int    arg1,              /* first of six args to format string */
    int    arg2,
    int    arg3,
    int    arg4,
    int    arg5,
    int    arg6
    )
```

DESCRIPTION This command has been superseded by **ftpCommandEnhanced( )**

This routine sends the specified command on the specified socket, which should be a control connection to a remote FTP server. The command is specified as a string in **printf( )** format with up to six arguments.

After the command is sent, **ftpCommand( )** waits for the reply from the remote server. The FTP reply code is returned in the same way as in **ftpReplyGet( )**.

EXAMPLE

```
ftpCommand (ctrlSock, "TYPE I", 0, 0, 0, 0, 0, 0);     /* image-type xfer */
ftpCommand (ctrlSock, "STOR %s", file, 0, 0, 0, 0, 0); /* init file write */
```

RETURNS 1 = **FTP_PRELIM** (positive preliminary)
2 = **FTP_COMPLETE** (positive completion)
3 = **FTP_CONTINUE** (positive intermediate)

4 = **FTP_TRANSIENT** (transient negative completion)
5 = **FTP_ERROR** (permanent negative completion)

ERROR if there is a read/write error or an unexpected EOF.

SEE ALSO **ftpLib**, **ftpReplyGet( )**

ftpCommandEnhanced( )

NAME **ftpCommandEnhanced( )** – send an FTP command and get the complete RFC reply code

SYNOPSIS

```
int ftpCommandEnhanced
    (
    int    ctrlSock,           /* fd of control connection socket */
    char * fmt,                /* format string of command to send */
    int    arg1,               /* first of six args to format string */
    int    arg2,
    int    arg3,
    int    arg4,
    int    arg5,
    int    arg6,
    char * replyString,        /* storage for the last line of the server */
                               /* response or NULL */
    int  replyStringLength  /* Maximum character length of the replyString */
    )
```

DESCRIPTION This command supersedes **ftpCommand( )**

This routine sends the specified command on the specified socket, which should be a control connection to a remote FTP server. The command is specified as a string in **printf( )** format with up to six arguments.

After the command is sent, **ftpCommand( )** waits for the reply from the remote server. The FTP reply code is returned in the same way as in **ftpReplyGetEnhanced( )**.

EXAMPLE

```
ftpCommandEnhanced (ctrlSock, "TYPE I", 0, 0, 0, 0, 0, 0, 0, 0);
                                                     /* image-type xfer */
ftpCommandEnhanced (ctrlSock, "STOR %s", file, 0, 0, 0, 0, 0, 0, 0);
                                                     /* init file write */
ftpCommandEnhanced (ctrlSock, "PASV", file, 0, 0, 0, 0, 0, reply, rplyLen);
                                                     /* Get port */
```

RETURNS The complete FTP response code (see RFC #959)

ERROR if there is a read/write error or an unexpected EOF.

SEE ALSO **ftpLib**, **ftpReplyGetEnhanced( )**, **ftpReplyGet( )**

ftpDataConnGet( )

NAME **ftpDataConnGet( )** – get a completed FTP data connection

SYNOPSIS

```
int ftpDataConnGet
    (
    int dataSock              /* fd of data socket on which to await */
                              /* connection */
    )
```

DESCRIPTION This routine completes a data connection initiated by a call to **ftpDataConnInit( )**. It waits for a connection on the specified socket from the remote FTP server. The specified socket should be the one returned by **ftpDataConnInit( )**. The connection is established on a new socket, whose file descriptor is returned as the result of this function. The original socket, specified in the argument to this routine, is closed.

Usually this routine is called after **ftpDataConnInit( )** and **ftpCommand( )** to initiate a data transfer from/to the remote FTP server.

RETURNS The file descriptor of the new data socket, or **ERROR** if the connection failed.

SEE ALSO **ftpLib**, **ftpDataConnInit( )**, **ftpCommand( )**

ftpDataConnInit( )

NAME **ftpDataConnInit( )** – initialize an FTP data connection using PORT mode

SYNOPSIS

```
int ftpDataConnInit
    (
    int ctrlSock              /* fd of associated control socket */
    )
```

DESCRIPTION This routine sets up the client side of a data connection for the specified control connection using the PORT command. It creates the data port, informs the remote FTP server of the data port address, and listens on that data port. The server will then connect to this data port in response to a subsequent data-transfer command sent on the control connection (see the manual entry for **ftpCommand( )**).

This routine must be called *before* the data-transfer command is sent; otherwise, the server's connect may fail.

This routine is called after **ftpHookup( )** and **ftpLogin( )** to establish a connection with a remote FTP server at the lowest level. (For a higher-level interaction with a remote FTP server, see **ftpXfer( )**.)

Please note that **ftpDataConnInitPassiveMode( )** is recommended instead of **ftpDataConnInit( )**.

RETURNS The file descriptor of the data socket created, or **ERROR**.

SEE ALSO **ftpLib**, **ftpDataConnInitPassiveMode( )**, **ftpHookup( )**, **ftpLogin( )**, **ftpCommand( )**, **ftpXfer( )**

ftpDataConnInitPassiveMode( )

NAME **ftpDataConnInitPassiveMode( )** – initialize an FTP data connection using PASV mode

SYNOPSIS

```
int ftpDataConnInitPassiveMode
    (
    int ctrlSock              /* fd of associated control socket */
    )
```

DESCRIPTION This routine sets up the client side of a data connection for the specified control connection. It issues a PASV command and attempts to connect to the host-specified port. If the host responds that it can not process the PASV command (command not supported) or fails to recognize the command, it will return **ERROR**.

This routine must be called *before* the data-transfer command is sent; otherwise, the server's connect may fail.

This routine is called after **ftpHookup( )** and **ftpLogin( )** to establish a connection with a remote FTP server at the lowest level. (For a higher-level interaction with a remote FTP server, see **ftpXfer( )**.)

This function is preferred over **ftpDataConnInit( )** because the remote system must preserve old port connection pairs even if the target system suffers from a reboot (2MSL). Using PASV we encourage the host's selection of a fresh port.

RETURNS The file descriptor of the data socket created, or **ERROR**.

SEE ALSO **ftpLib**, **ftpHookup( )**, **ftpLogin( )**, **ftpCommandEnhanced( )**, **ftpXfer( )**, **ftpConnInit( )**

ftpdDelete( )

NAME **ftpdDelete( )** – terminate the FTP server task

SYNOPSIS

```
STATUS ftpdDelete (void)
```

DESCRIPTION This routine halts the FTP server and closes the control connection. All client sessions are removed after completing any commands in progress. When this routine executes, no further client connections will be accepted until the server is restarted. This routine is not reentrant and must not be called from interrupt level.

NOTE: If any file transfer operations are in progress when this routine is executed, the transfers will be aborted, possibly leaving incomplete files on the destination host.

RETURNS **OK** if shutdown completed, or **ERROR** otherwise.

ERRNO N/A

SEE ALSO **ftpdLib**

ftpdInit( )

NAME **ftpdInit( )** – initialize the FTP server task

SYNOPSIS

```
STATUS ftpdInit
    (
    FUNCPTR pLoginRtn,        /* user verification routine, or NULL */
    int     stackSize         /* task stack size, or 0 for default */
    )
```

DESCRIPTION This routine installs the password verification routine indicated by *pLoginRtn* and establishes a control connection for the primary FTP server task, which it then creates. It is called automatically during system startup if **INCLUDE_FTP_SERVER** is defined. The primary server task supports simultaneous client sessions, up to the limit specified by the global variable **ftpsMaxClients**. The default value allows a maximum of four simultaneous connections. The *stackSize* argument specifies the stack size for the primary server task. It is set to the value specified in the **ftpdWorkTaskStackSize** global variable by default.

RETURNS **OK** if server started, or **ERROR** otherwise.

ERRNO N/A

SEE ALSO **ftpdLib**

ftpHookup( )

NAME **ftpHookup( )** – get a control connection to the FTP server on a specified host

SYNOPSIS

```
int ftpHookup
    (
    char * host              /* server host name or inet address */
    )
```

DESCRIPTION This routine establishes a control connection to the FTP server on the specified host. This is the first step in interacting with a remote FTP server at the lowest level. (For a higher-level interaction with a remote FTP server, see the manual entry for **ftpXfer( )**.)

RETURNS The file descriptor of the control socket, or **ERROR** if the Internet address or the host name is invalid, if a socket could not be created, or if a connection could not be made.

SEE ALSO **ftpLib**, **ftpLogin( )**, **ftpXfer( )**

ftpLibDebugOptionSet( )

NAME **ftpLibDebugOptionSet( )** – set the debug level of the ftp library routines

SYNOPSIS

```
void ftpLibDebugOptionSet
    (
    UINT32 debugLevel
    )
```

DESCRIPTION This routine enables the debugging of ftp transactions using the ftp library.

Debugging Level	Meaning
FTPL_DEBUG_OFF	No debugging messages.
FTPL_DEBUG_INCOMING	Display all incoming responses.
FTPL_DEBUG_OUTGOING	Display all outgoing commands.
FTPL_DEBUG_ERRORS	Display warnings and errors

EXAMPLE

```
ftpLibDebugOptionsSet (FTPL_DEBUG_ERRORS);   /* Display any runtime errors */
ftpLibDebugOptionsSet (FTPL_DEBUG_OUTGOING);  /* Display outgoing commands */
ftpLibDebugOptionsSet (FTPL_DEBUG_INCOMING);  /* Display incoming replies */
ftpLibDebugOptionsSet (FTPL_DEBUG_INCOMING |  /* Display both commands and */
                       FTPL_DEBUG_OUTGOING);  /*         replies */
```

RETURNS N/A

SEE ALSO **ftpLib**

ftpLogin( )

NAME **ftpLogin( )** – log in to a remote FTP server

SYNOPSIS

```
STATUS ftpLogin
    (
    int    ctrlSock,          /* fd of login control socket */
    char * user,              /* user name for host login */
    char * passwd,            /* password for host login */
    char * account            /* account for host login */
    )
```

DESCRIPTION This routine logs in to a remote server with the specified user name, password, and account name, as required by the specific remote host. This is typically the next step after calling **ftpHookup( )** in interacting with a remote FTP server at the lowest level. (For a higher-level interaction with a remote FTP server, see the manual entry for **ftpXfer( )**).

RETURNS **OK**, or **ERROR** if the routine is unable to log in.

SEE ALSO **ftpLib**, **ftpHookup( )**, **ftpXfer( )**

ftpLs( )

NAME **ftpLs( )** – list directory contents via FTP

SYNOPSIS

```
STATUS ftpLs
    (
    char * dirName            /* name of directory to list */
    )
```

DESCRIPTION This routine lists the contents of a directory. The content list is obtained via an NLST FTP transaction.

The local device name must be the same as the remote host name with a colon ":" as a suffix. (For example "wrs:" is the device name for the "wrs" host.)

RETURNS **OK**, or **ERROR** if could not open directory.

SEE ALSO **ftpLib**

ftpReplyGet( )

NAME **ftpReplyGet( )** – get an FTP command reply

SYNOPSIS

```
int ftpReplyGet
    (
    int  ctrlSock,            /* control socket fd of FTP connection */
    BOOL expecteof            /* TRUE = EOF expected, FALSE = EOF is error */
    )
```

DESCRIPTION This routine has been superseded by **ftpReplyGetEnhanced( )**

This routine gets a command reply on the specified control socket.

The three-digit reply code from the first line is saved and interpreted. The left-most digit of the reply code identifies the type of code (see RETURNS below).

The caller's error status is always set to the complete three-digit reply code regardless of the actual reply value (see the manual entry for **errnoGet( )**). If the reply code indicates an error, the entire reply is printed if the ftp error printing is enabled (see the manual entry for **ftpLibDebugOptionsSet( )**).

If an EOF is encountered on the specified control socket, but no EOF was expected (*expecteof* == **FALSE**), then **ERROR** is returned.

RETURNS 1 = **FTP_PRELIM** (positive preliminary)
2 = **FTP_COMPLETE** (positive completion)
3 = **FTP_CONTINUE** (positive intermediate)
4 = **FTP_TRANSIENT** (transient negative completion)
5 = **FTP_ERROR** (permanent negative completion)

ERROR if there is a read/write error or an unexpected EOF.

SEE ALSO **ftpLib**

F

ftpReplyGetEnhanced( )

NAME

ftpReplyGetEnhanced( ) – get an FTP command reply

SYNOPSIS

```
int ftpReplyGetEnhanced
    (
    int    ctrlSock,       /* control socket fd of FTP connection */
    BOOL   expecteof       /* TRUE = EOF expected, FALSE = EOF is error */
    char * replyString,    /* Location to store text of reply, or NULL */
    int    stringLengthMax /* Maximum length of reply (not including NULL) */
    )
```

DESCRIPTION

This routine supersedes **ftpReplyGet( )**

This routine gets a command reply on the specified control socket.

The three-digit reply code from the first line is saved and interpreted. The left-most digit of the reply code identifies the type of code (see RETURNS below).

The caller's error status is always set to the complete three-digit reply code (see the manual entry for **errnoGet( )**). If the reply code indicates an error, the entire reply is printed if the ftp error printing is enabled (see the manual entry for **ftpLibDebugOptionsSet( )**).

The last line of text retrieved from the servers response is stored in the location specified by *replyString*. If *replyString* is **NULL** the parameter is ignored.

If an EOF is encountered on the specified control socket, but no EOF was expected (*expecteof* == **FALSE**), then **ERROR** is returned.

RETURNS

The complete FTP response code (see RFC #959)

ERROR if there is a read/write error or an unexpected EOF.

SEE ALSO

ftpLib

ftpTransientConfigGet( )

NAME

ftpTransientConfigGet( ) – get parameters for host **FTP_TRANSIENT** responses

SYNOPSIS

```
STATUS ftpTransientConfigGet
    (
    UINT32 * maxRetryCount, /* The maximum number of attempts to retry */
    UINT32 * retryInterval /* time (in system clock ticks) between retries */
    )
```

DESCRIPTION

This routine retrieves the delay between retries in response to receiving **FTP_TRANSIENT** and the maximum retry count permitted before failing.

RETURNS

OK

SEE ALSO

ftpLib, **ftpTransientConfigSet( )**, **tickLib**

ftpTransientConfigSet( )

NAME

ftpTransientConfigSet( ) – set parameters for host **FTP_TRANSIENT** responses

SYNOPSIS

```
STATUS ftpTransientConfigSet
    (
    UINT32 maxRetryCount,  /* The maximum number of attempts to retry */
    UINT32 retryInterval   /* time (in system clock ticks) between retries */
    )
```

DESCRIPTION

This routine adjusts the delay between retries in response to receiving **FTP_PRELIM** and the maximum retry count permitted before failing.

RETURNS

OK

SEE ALSO

ftpLib

ftpTransientFatalInstall( )

NAME

ftpTransientFatalInstall( ) – set applette to stop FTP transient host responses

SYNOPSIS

```
STATUS ftpTransientFatalInstall
    (
    FUNCPTR pApplette /* function that returns TRUE or FALSE */
    )
```

DESCRIPTION

The routine installs a function which will determine if a transient resonse should be fatal. Some FTP servers incorrectly use **transient** responses instead of **error** to describe conditions such as **disk full**.

RETURNS

OK if the installation is successful, or **ERROR** if the installation fails.

SEE ALSO

ftpLib, **ftpTransientConfigSet( )**, **ftpTransientFatal( )** in **target/config/comps/src/net/usrFtp.c**.

ftpXfer( )

NAME

ftpXfer( ) – initiate a transfer via FTP

SYNOPSIS

```
STATUS ftpXfer
    (
    char * host,              /* name of server host */
    char * user,              /* user name for host login */
    char * passwd,            /* password for host login */
    char * acct,              /* account for host login */
    char * cmd,               /* command to send to host */
    char * dirname,           /* directory to cd to before sending command */
    char * filename,          /* filename to send with command */
    int *  pCtrlSock,         /* where to return control socket fd */
    int *  pDataSock          /* where to return data socket fd, (NULL == */
                              /* don't open data connection) */
    )
```

DESCRIPTION

This routine initiates a transfer via a remote FTP server in the following order:

(1) Establishes a connection to the FTP server on the specified host.
(2) Logs in with the specified user name, password, and account, as necessary for the particular host.

(3) Sets the transfer type to image by sending the command "TYPE I".
(4) Changes to the specified directory by sending the command "CWD *dirname*".
(5) Sends the specified transfer command with the specified filename as an argument, and establishes a data connection. Typical transfer commands are "STOR %s", to write to a remote file, or "RETR %s", to read a remote file.

The resulting control and data connection file descriptors are returned via *pCtrlSock* and *pDataSock*, respectively.

After calling this routine, the data can be read or written to the remote server by reading or writing on the file descriptor returned in *pDataSock*. When all incoming data has been read (as indicated by an EOF when reading the data socket) and/or all outgoing data has been written, the data socket fd should be closed. The routine **ftpReplyGet( )** should then be called to receive the final reply on the control socket, after which the control socket should be closed.

If the FTP command does not involve data transfer, *pDataSock* should be **NULL**, in which case no data connection will be established. The only FTP commands supported for this case are DELE, RMD, and MKD.

EXAMPLE

The following code fragment reads the file **/usr/fred/myfile** from the host "server", logged in as user "fred", with password "magic" and no account name.

```
#include "vxWorks.h"
#include "ftpLib.h"
int       ctrlSock;
int       dataSock;
char      buf [512];
int       nBytes;
STATUS    status;
if (ftpXfer ("server", "fred", "magic", "",
             "RETR %s", "/usr/fred", "myfile",
             &ctrlSock, &dataSock) == ERROR)
    return (ERROR);
while ((nBytes = read (dataSock, buf, sizeof (buf))) > 0)
    {
    ...
    }
close (dataSock);
if (nBytes < 0)             /* read error? */
    status = ERROR;
if (ftpReplyGet (ctrlSock, TRUE) != FTP_COMPLETE)
    status = ERROR;
if (ftpCommand (ctrlSock, "QUIT", 0, 0, 0, 0, 0, 0) != FTP_COMPLETE)
    status = ERROR;
close (ctrlSock);
```

RETURNS **OK**, or **ERROR** if any socket cannot be created or if a connection cannot be made.

SEE ALSO **ftpLib**, **ftpReplyGet( )**

ftruncate( )

NAME **ftruncate( )** – truncate a file (POSIX)

SYNOPSIS

```
int ftruncate
    (
    int    fildes,             /* fd of file to truncate */
    off_t length               /* length to truncate file */
    )
```

DESCRIPTION This routine truncates a file to a specified size.

RETURNS 0 (**OK**) or -1 (**ERROR**) if unable to truncate file.

ERRNO **EROFS** - File resides on a read-only file system.
EBADF - File is open for reading only.
EINVAL - File descriptor refers to a file on which this operation is impossible.

SEE ALSO **ftruncate**

fwrite( )

NAME **fwrite( )** – write from a specified array (ANSI)

SYNOPSIS

```
int fwrite
    (
    const void * buf,          /* where to copy from */
    size_t       size,         /* element size */
    size_t       count,        /* no. of elements */
    FILE *       fp            /* stream to write to */
    )
```

DESCRIPTION This routine writes, from the array *buf*, up to *count* elements whose size is *size*, to a specified stream. The file position indicator for the stream (if defined) is advanced by the number of characters successfully written. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate.

INCLUDE FILES **stdio.h**

RETURNS The number of elements successfully written, which will be less than *count* only if a write error is encountered.

SEE ALSO **ansiStdio**

getc( )

NAME **getc( )** – return the next character from a stream (ANSI)

SYNOPSIS

```
int getc
    (
    FILE * fp                    /* input stream */
    )
```

DESCRIPTION This routine is equivalent to **fgetc( )**, except that if it is implemented as a macro, it may evaluate *fp* more than once; thus the argument should never be an expression with side effects.

If the stream is at end-of-file, the end-of-file indicator for the stream is set; if a read error occurs, the error indicator is set.

INCLUDE FILES **stdio.h**

RETURNS The next character from the stream, or EOF if the stream is at end-of-file or a read error occurs.

SEE ALSO **ansiStdio**, **fgetc( )**

getchar( )

NAME **getchar( )** – return the next character from the standard input stream (ANSI)

SYNOPSIS

```
int getchar (void)
```

DESCRIPTION This routine returns the next character from the standard input stream and advances the file position indicator.

It is equivalent to **getc( )** with the stream argument **stdin**.

If the stream is at end-of-file, the end-of-file indicator is set; if a read error occurs, the error indicator is set.

INCLUDE FILES **stdio.h**

RETURNS The next character from the standard input stream, or EOF if the stream is at end-of-file or a read error occurs.

SEE ALSO **ansiStdio**, **getc( )**, **fgetc( )**

getcwd( )

NAME

getcwd( ) – get the current default path (POSIX)

SYNOPSIS

```
char *getcwd
    (
    char * buffer,            /* where to return the pathname */
    int    size               /* size in bytes of buffer */
    )
```

DESCRIPTION

This routine copies the name of the current default path to *buffer*. It provides the same functionality as **ioDefPathGet( )** and is provided for POSIX compatibility.

RETURNS

A pointer to the supplied buffer, or **NULL** if *size* is too small to hold the current default path.

SEE ALSO

ioLib, **ioDefPathSet( )**, **ioDefPathGet( )**, **chdir( )**

getenv( )

NAME

getenv( ) – get an environment variable (ANSI)

SYNOPSIS

```
char *getenv
    (
    const char * name         /* env variable to get value for */
    )
```

DESCRIPTION

This routine searches the environment list (see the UNIX BSD 4.3 manual entry for **environ(5V)**) for a string of the form "name=value" and returns the value portion of the string, if the string is present; otherwise it returns a **NULL** pointer.

RETURNS

A pointer to the string value, or a **NULL** pointer.

SEE ALSO

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

gethostname( )

NAME

gethostname( ) – get the symbolic name of this machine

SYNOPSIS

```
int gethostname
    (
    char * name,                /* machine name */
    int    nameLen              /* length of name */
    )
```

DESCRIPTION

This routine gets the target machine's symbolic name, which can be used for identification.

RETURNS

OK or **ERROR**.

SEE ALSO

hostLib

getpeername( )

NAME

getpeername( ) – get the name of a connected peer

SYNOPSIS

```
STATUS getpeername
    (
    int               s,       /* socket descriptor */
    struct sockaddr * name,    /* where to put name */
    int *             namelen  /* space available in name, later filled in */
                               /* with actual name size */
    )
```

DESCRIPTION

This routine gets the name of the peer connected to socket *s*. The *namelen* parameter should be initialized to indicate the amount of space referenced by *name*. On return, the name of the socket is copied to *name* and the actual size of the socket name is copied to *namelen*.

RETURNS

OK, or **ERROR** if the socket is invalid or not connected.

SEE ALSO

sockLib

gets( )

NAME **gets( )** – read characters from the standard input stream (ANSI)

SYNOPSIS

```
char * gets
    (
    char * buf                /* output array */
    )
```

DESCRIPTION This routine reads characters from the standard input stream into the array *buf* until end-of-file is encountered or a new-line is read. Any new-line character is discarded, and a null character is written immediately after the last character read into the array.

If end-of-file is encountered and no characters have been read, the contents of the array remain unchanged. If a read error occurs, the array contents are indeterminate.

INCLUDE FILES **stdio.h**

RETURNS A pointer to *buf*, or a null pointer if (1) end-of-file is encountered and no characters have been read, or (2) there is a read error.

SEE ALSO **ansiStdio**

getsockname( )

NAME **getsockname( )** – get a socket name

SYNOPSIS

```
STATUS getsockname
    (
    int               s,      /* socket descriptor */
    struct sockaddr * name,   /* where to return name */
    int *             namelen /* space available in name, later filled in */
                              /* with actual name size */
    )
```

DESCRIPTION This routine gets the current name for the specified socket *s*. The *namelen* parameter should be initialized to indicate the amount of space referenced by *name*. On return, the name of the socket is copied to *name* and the actual size of the socket name is copied to *namelen*.

RETURNS **OK**, or **ERROR** if the socket is invalid.

SEE ALSO **sockLib**

getsockopt( )

NAME **getsockopt( )** – get socket options

SYNOPSIS

```
STATUS getsockopt
    (
    int    s,                /* socket */
    int    level,            /* protocol level for options */
    int    optname,          /* name of option */
    char * optval,           /* where to put option */
    int *  optlen            /* where to put option length */
    )
```

DESCRIPTION This routine returns relevant option values associated with a socket. To manipulate options at the "socket" level, *level* should be **SOL_SOCKET**. Any other levels should use the appropriate protocol number. The *optlen* parameter should be initialized to indicate the amount of space referenced by *optval*. On return, the value of the option is copied to *optval* and the actual size of the option is copied to *optlen*.

Although *optval* is passed as a char *, the actual variable whose address gets passed in should be an integer or a structure, depending on which *optname* is being passed. Refer to **setsockopt( )** to determine the correct type of the actual variable (whose address should then be cast to a char *).

RETURNS **OK**, or **ERROR** if there is an invalid socket, an unknown option, or the call is unable to get the specified option.

EXAMPLE Because **SO_REUSEADDR** has an integer parameter, the variable to be passed to **getsockopt( )** should be declared as

```
int reuseVal;
```

and passed in as

```
(char *)&reuseVal.
```

Otherwise the user might mistakenly declare **reuseVal** as a character, in which case **getsockopt( )** will only return the first byte of the integer representing the state of this option. Then whether the return value is correct or always 0 depends on the endianness of the machine.

SEE ALSO **sockLib**, **setsockopt( )**

getw( )

NAME

getw( ) – read the next word (32-bit integer) from a stream

SYNOPSIS

```
int getw
    (
    FILE * fp                 /* stream to read from */
    )
```

DESCRIPTION

This routine reads the next 32-bit quantity from a specified stream. It returns EOF on end-of-file or an error; however, this is also a valid integer, thus **feof( )** and **ferror( )** must be used to check for a true end-of-file.

This routine is provided for compatibility with earlier VxWorks releases.

INCLUDE FILES

stdio.h

RETURN

A 32-bit number from the stream, or EOF on either end-of-file or an error.

SEE ALSO

ansiStdio, **putw( )**

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.

SEE ALSO

ioLib

G

gmtime( )

NAME

gmtime( ) – convert calendar time into UTC broken-down time (ANSI)

SYNOPSIS

```
struct tm *gmtime
    (
    const time_t * timer       /* calendar time in seconds */
    )
```

DESCRIPTION

This routine converts the calendar time pointed to by *timer* into broken-down time, expressed as Coordinated Universal Time (UTC).

This routine is not reentrant. For a reentrant version, see **gmtime_r( )**.

INCLUDE FILES

time.h

RETURNS

A pointer to a broken-down time structure (**tm**), or a null pointer if UTC is not available.

SEE ALSO

ansiTime

gmtime_r( )

NAME

gmtime_r( ) – convert calendar time into broken-down time (POSIX)

SYNOPSIS

```
int gmtime_r
    (
    const time_t * timer,      /* calendar time in seconds */
    struct tm *    timeBuffer /* buffer for broken down time */
    )
```

DESCRIPTION

This routine converts the calendar time pointed to by *timer* into broken-down time, expressed as Coordinated Universal Time (UTC). The broken-down time is stored in *timeBuffer*.

This routine is the POSIX re-entrant version of **gmtime( )**.

INCLUDE FILES

time.h

RETURNS

OK.

SEE ALSO

ansiTime

h( )

NAME **h( )** – display or set the size of shell history

SYNOPSIS

```
void h
    (
    int size                  /* 0 = display, >0 = set history to new size */
    )
```

DESCRIPTION This command displays or sets the size of VxWorks shell history. If no argument is specified, shell history is displayed. If *size* is specified, that number of the most recent commands is saved for display. The value of *size* is initially 20.

RETURNS N/A

SEE ALSO **usrLib**, **shellHistory( )**, **ledLib**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

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 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)

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)

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 mumbo jumbo)

RETURNS

integer between 0 and (elements - 1)

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 ingnored */
    )
```

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.

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 ingnored */
    )
```

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.

SEE ALSO **hashLib**

hashLibInit( )

NAME **hashLibInit( )** – initialize hash table library

SYNOPSIS

```
STATUS hashLibInit (void)
```

DESCRIPTION This routine initializes the hash table package.

SEE ALSO **hashLib**

hashTblCreate( )

NAME **hashTblCreate( )** – create a hash table

SYNOPSIS

```
HASH_ID hashTblCreate
    (
    int     sizeLog2,        /* number of elements in hash table log 2 */
    FUNCPTR keyCmpRtn,       /* function to test keys for equivalence */
    FUNCPTR keyRtn,          /* hashing function to generate hash from key */
    int     keyArg           /* argument to hashing function */
    )
```

DESCRIPTION This routine creates a hash table 2^sizeLog2 number of elements. The hash table is carved from the system memory pool via **malloc (2)**. To accommodate the list structures associated with the table, the actual amount of memory allocated will be 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 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.

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.

SEE ALSO **hashLib**

H

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.

SEE ALSO **hashLib**

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.

SEE ALSO

hashLib

hashTblFind( )

NAME

hashTblFind( ) – find a hash node that matches the specified key

SYNOPSIS

```
HASH_NODE *hashTblFind
    (
    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.

SEE ALSO

hashLib

hashTblInit( )

NAME **hashTblInit( )** – initialize a hash table

SYNOPSIS

```
STATUS hashTblInit
    (
    HASH_TBL * pHashTbl,     /* pointer to 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.

RETURNS **OK**

SEE ALSO **hashLib**

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.

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 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 or no matching hash node is found.

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 hash table is marked as invalid.

RETURNS

OK, or **ERROR** if hashId is invalid.

SEE ALSO

hashLib

help( )

NAME

help( ) – print a synopsis of selected routines

SYNOPSIS

```
void help (void)
```

DESCRIPTION

This command prints the following list of the calling sequences for commonly used routines, mostly contained in **usrLib**.

```
help                           Print this list
ioHelp                         Print I/O utilities help info
dbgHelp                        Print debug help info
nfsHelp                        Print nfs help info
netHelp                        Print network help info
spyHelp                        Print task histogrammer help info
timexHelp                      Print execution timer help info
h         [n]                  Print (or set) shell history
i         [task]               Summary of tasks' TCBs
ti        task                 Complete info on TCB for task
sp        adr,args...          Spawn a task, pri=100, opt=0x19, stk=20000
taskSpawn name,pri,opt,stk,adr,args... Spawn a task
td        task                 Delete a task
ts        task                 Suspend a task
tr        task                 Resume a task
d         [adr[,nunits[,width]]]   Display memory
m         adr[,width]          Modify memory
mRegs     [reg[,task]]         Modify a task's registers interactively
pc        [task]               Return task's program counter
version                        Print VxWorks version info, and boot line
iam     "user"[,"passwd"]      Set user name and passwd
whoami                         Print user name
devs                           List devices
ld        [syms[,noAbort][,"name"]] Load std in into memory
                                 (syms = add symbols to table:
                                  -1 = none, 0 = globals, 1 = all)
lkup      ["substr"]           List symbols in system symbol table
lkAddr    address              List symbol table entries near address
checkStack  [task]             List task stack sizes and usage
printErrno  value              Print the name of a status value
period    secs,adr,args...     Spawn task to call function periodically
repeat    n,adr,args...        Spawn task to call function n times
                                 (0=forever)
NOTE:  Arguments specifying <task> can be either task ID or name.
```

RETURNS

N/A

SEE ALSO

usrLib, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

hostAdd( )

NAME

hostAdd( ) – add a host to the host table

SYNOPSIS

```
STATUS hostAdd
    (
    char * hostName,          /* host name */
    char * hostAddr           /* host addr in standard Internet format */
    )
```

DESCRIPTION

This routine adds a host name to the local host table. This must be called before sockets on the remote host are opened, or before files on the remote host are accessed via **netDrv** or **nfsDrv**.

The host table has one entry per Internet address. More than one name may be used for an address. Additional host names are added as aliases.

EXAMPLE

```
-> hostAdd "wrs", "90.2"
-> hostShow
hostname          inet address        aliases
--------          ------------        -------
localhost         127.0.0.1
yuba              90.0.0.3
wrs               90.0.0.2
value = 12288 = 0x3000 = _bzero + 0x18
```

RETURNS

OK, or **ERROR** if the host table is full, the host name/inet address pair is already entered, the Internet address is invalid, or memory is insufficient.

SEE ALSO

hostLib, **netDrv**, **nfsDrv**

hostDelete( )

NAME

hostDelete( ) – delete a host from the host table

SYNOPSIS

```
STATUS hostDelete
    (
    char * name,              /* host name or alias */
    char * addr               /* host addr in standard Internet format */
    )
```

DESCRIPTION This routine deletes a host name from the local host table. If *name* is a host name, the host entry is deleted. If *name* is a host name alias, the alias is deleted.

RETURNS **OK**, or **ERROR** if the parameters are invalid or the host is unknown.

ERRNO **S_hostLib_INVALID_PARAMETER**, **S_hostLib_UNKNOWN_HOST**

SEE ALSO **hostLib**

hostGetByAddr( )

NAME **hostGetByAddr( )** – look up a host in the host table by its Internet address

SYNOPSIS

```
STATUS hostGetByAddr
    (
    int    addr,              /* inet address of host */
    char * name               /* buffer to hold name */
    )
```

DESCRIPTION This routine finds the host name by its Internet address and copies it to *name*. The buffer *name* should be pre-allocated with (MAXHOSTNAMELEN + 1) bytes of memory and is **NULL**-terminated unless insufficient space is provided. If the DNS resolver library **resolvLib** has been configured in the vxWorks image, a query for the host name is sent to the DNS server, if the name was not found in the local host table.

WARNING: This routine does not look for aliases. Host names are limited to **MAXHOSTNAMELEN** (from **hostLib.h**) characters.

RETURNS **OK**, or **ERROR** if buffer is invalid or the host is unknown.

SEE ALSO **hostLib**, **hostGetByName( )**

hostGetByName( )

NAME **hostGetByName( )** – look up a host in the host table by its name

SYNOPSIS

```
int hostGetByName
    (
    char * name                /* name of host */
    )
```

DESCRIPTION This routine returns the Internet address of a host that has been added to the host table by **hostAdd( )**. If the DNS resolver library **resolvLib** has been configured in the vxWorks image, a query for the host IP address is sent to the DNS server, if the name was not found in the local host table.

RETURNS The Internet address (as an integer), or **ERROR** if the host is unknown.

ERRNO **S_hostLib_INVALID_PARAMETER**, **S_hostLib_UNKNOWN_HOST**

SEE ALSO **hostLib**

hostShow( )

NAME **hostShow( )** – display the host table

SYNOPSIS

```
void hostShow (void)
```

DESCRIPTION This routine prints a list of remote hosts, along with their Internet addresses and aliases.

RETURNS N/A

SEE ALSO **netShow**, **hostAdd( )**

hostTblInit( )

NAME **hostTblInit( )** – initialize the network host table

SYNOPSIS

```
void hostTblInit (void)
```

DESCRIPTION This routine initializes the host list data structure used by routines throughout this module. It should be called before any other routines in this module. This is done automatically if **INCLUDE_HOST_TBL** is defined.

RETURNS N/A

SEE ALSO **hostLib**, **usrConfig**

i( )

NAME

i( ) – print a summary of each task's TCB

SYNOPSIS

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

DESCRIPTION

This command displays a synopsis of all the tasks in the system. The **ti( )** routine provides more complete information on a specific task.

Both **i( )** and **ti( )** use **taskShow( )**; see the documentation for **taskShow( )** for a description of the output format.

EXAMPLE

```
-> i
   NAME        ENTRY       TID    PRI   STATUS     PC       SP     ERRNO DELAY
---------- ---------- -------- --- --------- ------- -------- ----- -----
tExcTask   _excTask    20fcb00   0 PEND      200c5fc  20fca6c     0     0
tLogTask   _logTask    20fb5b8   0 PEND      200c5fc  20fb520     0     0
tShell     _shell      20efcac   1 READY     201dc90  20ef980     0     0
tRlogind   _rlogind    20f3f90   2 PEND      2038614  20f3db0     0     0
tTelnetd   _telnetd    20f2124   2 PEND      2038614  20f2070     0     0
tNetTask   _netTask    20f7398  50 PEND      2038614  20f7340     0     0
value = 57 = 0x39 = '9'
```

WARNING: This command should be used only as a debugging aid, since the information is obsolete by the time it is displayed.

RETURNS

N/A

SEE ALSO

usrLib, **ti( )**, **taskShow( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

iam( )

NAME **iam( )** – set the remote user name and password

SYNOPSIS

```
STATUS iam
    (
    char * newUser,          /* user name to use on remote */
    char * newPasswd         /* password to use on remote (NULL = none) */
    )
```

DESCRIPTION This routine specifies the user name that will have access privileges on the remote machine. The user name must exist in the remote machine's **/etc/passwd**, and if it has been assigned a password, the password must be specified in *newPasswd*.

Either parameter can be **NULL**, and the corresponding item will not be set.

The maximum length of the user name and the password is **MAX_IDENTITY_LEN**(defined in **remLib.h**).

NOTE: This routine is a more convenient version of **remCurIdSet( )** and is intended to be used from the shell.

RETURNS **OK**, or **ERROR** if the call fails.

SEE ALSO **remLib**, **whoami( )**, **remCurIdGet( )**, **remCurIdSet( )**

icmpShowInit( )

NAME **icmpShowInit( )** – initialize ICMP show routines

SYNOPSIS

```
void icmpShowInit (void)
```

DESCRIPTION This routine links the ICMP show facility into the VxWorks system. These routines are included automatically if **INCLUDE_NET_SHOW** and **INCLUDE_ICMP** are defined.

RETURNS N/A

SEE ALSO **icmpShow**

I

icmpstatShow( )

NAME **icmpstatShow( )** – display statistics for ICMP

SYNOPSIS

```
void icmpstatShow (void)
```

DESCRIPTION This routine displays statistics for the ICMP (Internet Control Message Protocol) protocol.

RETURNS N/A

SEE ALSO **icmpShow**

ifAddrAdd( )

NAME **ifAddrAdd( )** – add an interface address for a network interface

SYNOPSIS

```
STATUS ifAddrAdd
    (
    char * interfaceName,     /* name of interface to configure */
    char * interfaceAddress,  /* Internet address to assign to interface */
    char * broadcastAddress,  /* broadcast address to assign to interface */
    int    subnetMask         /* subnetMask */
    )
```

DESCRIPTION This routine assigns an Internet address to a specified network interface. The Internet address can be a host name or a standard Internet address format (*e.g.*, 90.0.0.4). If a host name is specified, it should already have been added to the host table with **hostAdd( )**.

You must specify both an *interfaceName* and an *interfaceAddress*. A *broadcastAddress* is optional. If *broadcastAddress* is **NULL**, **in_ifinit( )** generates a *broadcastAddress* value based on the *interfaceAddress* value and the netmask. A *subnetMask* value is optional. If *subnetMask* is 0, **in_ifinit( )** uses a *subnetMask* the same as the netmask that is generated by the *interfaceAddress*. The *broadcastAddress* is also *destAddress* in case of **IFF_POINTOPOINT**.

RETURNS **OK**, or **ERROR** if the interface cannot be set.

SEE ALSO **ifLib**, **ifAddrGet( )**, **ifDstAddrSet( )**, **ifDstAddrGet( )**

ifAddrDelete( )

NAME
ifAddrDelete( ) – delete an interface address for a network interface

SYNOPSIS
```
STATUS ifAddrDelete
    (
    char * interfaceName,     /* name of interface to delete addr from */
    char * interfaceAddress   /* Internet address to delete from interface */
    )
```

DESCRIPTION
This routine deletes an Internet address from a specified network interface. The Internet address can be a host name or a standard Internet address format (*e.g.*, 90.0.0.4). If a host name is specified, it should already have been added to the host table with **hostAdd( )**.

RETURNS
OK, or **ERROR** if the interface cannot be deleted.

SEE ALSO
ifLib, **ifAddrGet( )**, **ifDstAddrSet( )**, **ifDstAddrGet( )**

ifAddrGet( )

NAME
ifAddrGet( ) – get the Internet address of a network interface

SYNOPSIS
```
STATUS ifAddrGet
    (
    char * interfaceName,     /* name of interface, i.e. ei0 */
    char * interfaceAddress   /* buffer for Internet address */
    )
```

DESCRIPTION
This routine gets the Internet address of a specified network interface and copies it to *interfaceAddress*. This pointer should point to a buffer large enough to contain **INET_ADDR_LEN** bytes.

RETURNS
OK or **ERROR**.

SEE ALSO
ifLib, **ifAddrSet( )**, **ifDstAddrSet( )**, **ifDstAddrGet( )**

I

ifAddrSet( )

NAME

ifAddrSet( ) – set an interface address for a network interface

SYNOPSIS

```
STATUS ifAddrSet
    (
    char * interfaceName,     /* name of interface to configure, i.e. ei0 */
    char * interfaceAddress   /* Internet address to assign to interface */
    )
```

DESCRIPTION

This routine assigns an Internet address to a specified network interface. The Internet address can be a host name or a standard Internet address format (*e.g.*, 90.0.0.4). If a host name is specified, it should already have been added to the host table with **hostAdd( )**.

A successful call to **ifAddrSet( )** results in the addition of a new route.

The subnet mask used in determining the network portion of the address will be that set by **ifMaskSet( )**, or the default class mask if **ifMaskSet( )** has not been called. It is standard practice to call **ifMaskSet( )** prior to calling **ifAddrSet( )**.

RETURNS

OK, or **ERROR** if the interface cannot be set.

SEE ALSO

ifLib, **ifAddrGet( )**, **ifDstAddrSet( )**, **ifDstAddrGet( )**

ifAllRoutesDelete( )

NAME

ifAllRoutesDelete( ) – delete all routes associated with a network interface

SYNOPSIS

```
int ifAllRoutesDelete
    (
    char * ifName,            /* name of the interface */
    int    unit               /* unit number for this interface */
    )
```

DESCRIPTION

This routine deletes all routes that have been associated with the specified interface. The routes deleted are:

- the network route added when the interface address is initialized
- the static routes added by the administrator
- ARP routes passing through the interface

Routes added by routing protocols are not deleted.

RETURNS The number of routes deleted, or **ERROR** if an interface is not specified.

SEE ALSO **ifLib**

ifBroadcastGet( )

NAME **ifBroadcastGet( )** – get the broadcast address for a network interface

SYNOPSIS

```
STATUS ifBroadcastGet
    (
    char * interfaceName,     /* name of interface, i.e. ei0 */
    char * broadcastAddress   /* buffer for broadcast address */
    )
```

DESCRIPTION This routine gets the broadcast address for a specified network interface. The broadcast address is copied to the buffer *broadcastAddress*.

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

SEE ALSO **ifLib**, **ifBroadcastSet( )**

ifBroadcastSet( )

NAME **ifBroadcastSet( )** – set the broadcast address for a network interface

SYNOPSIS

```
STATUS ifBroadcastSet
    (
    char * interfaceName,     /* name of interface to assign, i.e. ei0 */
    char * broadcastAddress   /* broadcast address to assign to interface */
    )
```

DESCRIPTION This routine assigns a broadcast address for the specified network interface. The broadcast address must be a string in standard Internet address format (*e.g.*, 90.0.0.0).

An interface's default broadcast address is its Internet address with a host part of all ones (*e.g.*, 90.255.255.255). This conforms to current ARPA specifications. However, some older systems use an Internet address with a host part of all zeros as the broadcast address.

NOTE: VxWorks automatically accepts a host part of all zeros as a broadcast address, in addition to the default or specified broadcast address. But if VxWorks is to broadcast to older systems using a host part of all zeros as the broadcast address, this routine should be used to change the broadcast address of the interface.

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

SEE ALSO **ifLib**

ifDstAddrGet( )

NAME **ifDstAddrGet( )** – get the Internet address of a point-to-point peer

SYNOPSIS

```
STATUS ifDstAddrGet
    (
    char * interfaceName,     /* name of interface, i.e. ei0 */
    char * dstAddress         /* buffer for destination address */
    )
```

DESCRIPTION This routine gets the Internet address of a machine connected to the opposite end of a point-to-point network connection. The Internet address is copied to the buffer *dstAddress*.

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

SEE ALSO **ifLib**, **ifDstAddrSet( )**, **ifAddrGet( )**

ifDstAddrSet( )

NAME **ifDstAddrSet( )** – define an address for the other end of a point-to-point link

SYNOPSIS

```
STATUS ifDstAddrSet
    (
    char * interfaceName,     /* name of interface to configure, i.e. ei0 */
    char * dstAddress         /* Internet address to assign to destination */
    )
```

DESCRIPTION This routine assigns the Internet address of a machine connected to the opposite end of a point-to-point network connection, such as a SLIP connection. Inherently, point-to-point connection-oriented protocols such as SLIP require that addresses for both ends of a connection be specified.

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

SEE ALSO **ifLib**, **ifAddrSet( )**, **ifDstAddrGet( )**

ifFlagChange( )

NAME **ifFlagChange( )** – change the network interface flags

SYNOPSIS

```
STATUS ifFlagChange
    (
    char * interfaceName,     /* name of the network interface, i.e. ei0 */
    int    flags,             /* the flag to be changed */
    BOOL   on                 /* TRUE=turn on, FALSE=turn off */
    )
```

DESCRIPTION This routine changes the flags for the specified network interfaces. If the parameter *on* is **TRUE**, the specified flags are turned on; otherwise, they are turned off. The routines **ifFlagGet( )** and **ifFlagSet( )** are called to do the actual work.

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

SEE ALSO **ifLib**, **ifAddrSet( )**, **ifMaskSet( )**, **ifFlagSet( )**, **ifFlagGet( )**

ifFlagGet( )

NAME

ifFlagGet( ) – get the network interface flags

SYNOPSIS

```
STATUS ifFlagGet
    (
    char * interfaceName,     /* name of the network interface, i.e. ei0 */
    int *  flags              /* network flags returned here */
    )
```

DESCRIPTION

This routine gets the flags for a specified network interface. The flags are copied to the buffer *flags*.

RETURNS

OK or **ERROR**.

SEE ALSO

ifLib, **ifFlagSet( )**

ifFlagSet( )

NAME

ifFlagSet( ) – specify the flags for a network interface

SYNOPSIS

```
STATUS ifFlagSet
    (
    char * interfaceName,     /* name of the network interface, i.e. ei0 */
    int    flags              /* network flags */
    )
```

DESCRIPTION

This routine changes the flags for a specified network interface. Any combination of the following flags can be specified:

IFF_UP (0x1)
: Brings the network up or down.

IFF_DEBUG (0x4)
: Turns on debugging for the driver interface if supported.

IFF_LOOPBACK (0x8)
: Set for a loopback network.

IFF_NOTRAILERS (0x20)
: Always set (VxWorks does not use the trailer protocol).

IFF_PROMISC (0x100)
Tells the driver to accept all packets, not just broadcast packets and packets addressed to itself.

IFF_ALLMULTI (0x200)
Tells the driver to accept all multicast packets.

IFF_NOARP (0x80)
Disables ARP for the interface.

NOTE: The following flags can only be set at interface initialization time. Specifying these flags does not change any settings in the interface data structure.

IFF_POINTOPOINT (0x10)
Identifies a point-to-point interface such as PPP or SLIP.

IFF_RUNNING (0x40)
Set when the device turns on.

IFF_BROADCAST (0x2)
Identifies a broadcast interface.

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

SEE ALSO **ifLib**, **ifFlagChange( )**, **ifFlagGet( )**

ifIndexAlloc( )

NAME **ifIndexAlloc( )** – return a unique interface index

SYNOPSIS

```
int ifIndexAlloc (void)
```

DESCRIPTION **ifIndexAlloc( )** returns a unique integer to be used as an interface index. The first index returned is 1. **ERROR** is returned if the library has not been initialized by a call to **ifIndexLibInit( )**.

RETURNS interface index or **ERROR**

SEE ALSO **ifIndexLib**

ifIndexLibInit( )

NAME **ifIndexLibInit( )** – initializes library variables

SYNOPSIS

```
void ifIndexLibInit (void)
```

DESCRIPTION **ifIndexLibInit( )** resets library internal state. This function must be called before any other functions in this library.

RETURNS N/A

SEE ALSO **ifIndexLib**

ifIndexLibShutdown( )

NAME **ifIndexLibShutdown( )** – frees library variables

SYNOPSIS

```
void ifIndexLibShutdown (void)
```

DESCRIPTION **ifIndexLibShutdown( )** frees library internal structures. **ifIndexLibInit( )** must be called before the library can be used again.

RETURNS N/A

SEE ALSO **ifIndexLib**

ifIndexTest( )

NAME **ifIndexTest( )** – returns true if an index has been allocated.

SYNOPSIS
```
BOOL ifIndexTest
    (
    int ifIndex               /* the index to test */
    )
```

DESCRIPTION **ifIndexTest( )** returns **TRUE** if *index* has already been allocated by **ifIndexLibAlloc( )**. Otherwise returns **FALSE**. If the library has not been initialized returns **FALSE**. This function does not check if the index actually belongs to a currently valid interface.

RETURNS **TRUE** or **FALSE**

SEE ALSO **ifIndexLib**

ifIndexToIfName( )

NAME **ifIndexToIfName( )** – returns the interface name given the interface index

SYNOPSIS
```
STATUS ifIndexToIfName
    (
    unsigned short ifIndex,   /* Interface index */
    char *         ifName     /* Where the name is to be stored */
    )
```

DESCRIPTION This routine returns the interface name for the interface referenced by the *ifIndex* parameter.

ifIndex
The index for the interface.

ifName
The location where the interface name is copied

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

SEE ALSO **ifLib**

ifMaskGet( )

NAME
ifMaskGet( ) – get the subnet mask for a network interface

SYNOPSIS

```
STATUS ifMaskGet
    (
    char * interfaceName,      /* name of interface, i.e. ei0 */
    int *  netMask             /* buffer for subnet mask */
    )
```

DESCRIPTION
This routine gets the subnet mask for a specified network interface. The subnet mask is copied to the buffer *netMask*. The subnet mask is returned in host byte order.

RETURNS
OK or **ERROR**.

SEE ALSO
ifLib, **ifAddrGet( )**, **ifFlagGet( )**

ifMaskSet( )

NAME
ifMaskSet( ) – define a subnet for a network interface

SYNOPSIS

```
STATUS ifMaskSet
    (
    char * interfaceName,    /* name of interface to set mask for, i.e. ei0 */
    int    netMask           /* subnet mask (e.g. 0xff000000) */
    )
```

DESCRIPTION
This routine allocates additional bits to the network portion of an Internet address. The network portion is specified with a mask that must contain ones in all positions that are to be interpreted as the network portion. This includes all the bits that are normally interpreted as the network portion for the given class of address, plus the bits to be added. Note that all bits must be contiguous. The mask is specified in host byte order.

In order to correctly interpret the address, a subnet mask should be set for an interface prior to setting the Internet address of the interface with the routine **ifAddrSet( )**.

RETURNS
OK or **ERROR**.

SEE ALSO
ifLib, **ifAddrSet( )**

ifMetricGet( )

NAME **ifMetricGet( )** – get the metric for a network interface

SYNOPSIS

```
STATUS ifMetricGet
    (
    char * interfaceName,     /* name of the network interface, i.e. ei0 */
    int *  pMetric            /* returned interface's metric */
    )
```

DESCRIPTION This routine retrieves the metric for a specified network interface. The metric is copied to the buffer *pMetric*.

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

SEE ALSO **ifLib**, **ifMetricSet( )**

ifMetricSet( )

NAME **ifMetricSet( )** – specify a network interface hop count

SYNOPSIS

```
STATUS ifMetricSet
    (
    char * interfaceName,     /* name of the network interface, i.e. ei0 */
    int    metric             /* metric for this interface */
    )
```

DESCRIPTION This routine configures *metric* for a network interface from the host machine to the destination network. This information is used primarily by the IP routing algorithm to compute the relative distance for a collection of hosts connected to each interface. For example, a higher *metric* for SLIP interfaces can be specified to discourage routing a packet to slower serial line connections. Note that when *metric* is zero, the IP routing algorithm allows for the direct sending of a packet having an IP network address that is not necessarily the same as the local network address.

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

SEE ALSO **ifLib**, **ifMetricGet( )**

ifNameToIfIndex( )

NAME **ifNameToIfIndex( )** – returns the interface index given the interface name

SYNOPSIS

```
unsigned short ifNameToIfIndex
    (
    char * ifName               /* a string describing the full interface */
                                /* name. e.g., "fei0" */
    )
```

DESCRIPTION This routine returns the interface index for the interface named by the *ifName* parameter, which provides a string describing the full interface name. For example, "fei0".

RETURNS The interface index, if the interface could be located, 0, otherwise. 0 is not a valid value for interface index.

SEE ALSO **ifLib**

ifRouteDelete( )

NAME **ifRouteDelete( )** – delete routes associated with a network interface

SYNOPSIS

```
int ifRouteDelete
    (
    char * ifName,              /* name of the interface */
    int    unit                 /* unit number for this interface */
    )
```

DESCRIPTION This routine deletes all routes that have been associated with the specified interface. A route is associated with an interface if its destination equals to the assigned address, or network number. This routine does not remove routes to arbitrary destinations that through the given interface.

RETURNS The number of routes deleted, or **ERROR** if an interface is not specified.

SEE ALSO **ifLib**

ifShow( )

NAME

ifShow( ) – display the attached network interfaces

SYNOPSIS

```
void ifShow
    (
    char * ifName             /* name of the interface to show */
    )
```

DESCRIPTION

This routine displays the attached network interfaces for debugging and diagnostic purposes. If *ifName* is given, only the interfaces belonging to that group are displayed. If *ifName* is omitted, all attached interfaces are displayed.

For each interface selected, the following are shown: Internet address, point-to-point peer address (if using SLIP), broadcast address, netmask, subnet mask, Ethernet address, route metric, maximum transfer unit, number of packets sent and received on this interface, number of input and output errors, and flags (such as loopback, point-to-point, broadcast, promiscuous, ARP, running, and debug).

EXAMPLE

The following call displays all interfaces whose names begin with "ln", (such as "ln0", "ln1", and "ln2"):

```
-> ifShow "ln"
```

The following call displays just the interface "ln0":

```
-> ifShow "ln0"
```

RETURNS

N/A

SEE ALSO

netShow, **routeShow( )**, **ifLib**

ifunit( )

NAME

ifunit( ) – map an interface name to an interface structure pointer

SYNOPSIS

```
struct ifnet *ifunit
    (
    char * ifname             /* name of the interface */
    )
```

DESCRIPTION

This routine returns a pointer to a network interface structure for *name* or **NULL** if no such interface exists. For example:

```
struct ifnet *pIf;
...
pIf = ifunit ("ln0");
```

pIf points to the data structure that describes the first network interface device if ln0 is mapped successfully.

RETURNS

A pointer to the interface structure, or **NULL** if an interface is not found.

SEE ALSO

ifLib

---

ifUnnumberedSet( )

NAME

ifUnnumberedSet( ) – configure an interface to be unnumbered

SYNOPSIS

```
STATUS ifUnnumberedSet
    (
    char * pIfName,            /* Name of interface to configure */
    char * pDstIp,             /* Destination address of the point to */
                               /* point link */
    char * pBorrowedIp,        /* The borrowed IP address/router ID */
    char * pDstMac             /* Destination MAC address */
    )
```

DESCRIPTION

This API sets an interface unnumbered. It sets the **IFF_POINTOPOINT** flags and creates a routing entry through the interface using a user-specified destination IP address. The unnumbered link can then be uniquely referred to by the destination IP address, *pDstIp*, when adding routes. The interface is assigned a "borrowed" IP address--borrowed from another interface on the machine. In RFC 1812 it is also called the router ID. This address will be used to generate any needed ICMP messages or the like. Note that ARP is not able to run on an unnumbered link.

The initialization of the unnumbered device is similar to other network devices, but it does have a few additional steps and concerns. **ifUnnumberedSet( )** must come next after **ipAttach( )**. Please note that the interface using the IP address that the unnumbered interface will borrow must be brought up first and configured with *ifAddrSet* or equivalent. This is required to ensure normal network operation for that IP address/interface. After **ifUnnumberedSet( )**, one must create additional routing entries (using **mRouteAdd( )**, **routeNetAdd( )**, *etc.*) in order to reach other networks, including the network to which the destination IP address belongs.

The *pDstMac* field in **ifUnnumberedSet( )** is used to specify the destination's MAC address. It should be left **NULL** if the destination is not an Ethernet device. If the MAC address is not known, then supply an artificial address. We recommend using "00:00:00:00:00:01" The destination interface can then be set promiscuous to accept this artificial address. This is accomplished using the *ifpromisc* command.

Example:

```
ipAttach (1, "fei")
ifUnnumberedSet ("fei1", "120.12.12.12", "140.34.78.94", "00:a0:d0:d8:c8:14")
routeNetAdd ("120.12.0.0","120.12.12.12") <One possible network>
routeNetAdd ("178.45.0.0","120.12.12.12") <Another possible network>
```

RETURNS **OK**, or **ERROR** if the interface cannot be set.

SEE ALSO **ifLib**

---

igmpShowInit( )

NAME **igmpShowInit( )** – initialize IGMP show routines

SYNOPSIS

```
void igmpShowInit (void)
```

DESCRIPTION This routine links the IGMP show facility into the VxWorks system. These routines are included automatically if **INCLUDE_NET_SHOW** and **INCLUDE_IGMP** are defined.

RETURNS N/A

SEE ALSO **igmpShow**

igmpstatShow( )

NAME **igmpstatShow( )** – display statistics for IGMP

SYNOPSIS

```
void igmpstatShow (void)
```

DESCRIPTION This routine displays statistics for the IGMP (Internet Group Management Protocol) protocol.

RETURNS N/A

SEE ALSO **igmpShow**

index( )

NAME **index( )** – find the first occurrence of a character in a string

SYNOPSIS

```
char *index
    (
    const char * s,           /* string in which to find character */
    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.

SEE ALSO **bLib**, **strchr( )**

inet_addr( )

NAME
inet_addr( ) – convert a dot notation Internet address to a long integer

SYNOPSIS
```
u_long inet_addr
    (
    char * inetString          /* string inet address */
    )
```

DESCRIPTION
This routine interprets an Internet address. All the network library routines call this routine to interpret entries in the data bases which are expected to be an address. The value returned is in network order. Numbers will be interpreted as octal if preceded by a zero (*e.g.,* "017.0.0.3"), as hexadecimal if preceded by 0x (*e.g.,* "0x17.0.0.4"), and as decimal in all other cases.

EXAMPLE
The following example returns 0x5a000002:

```
    inet_addr ("90.0.0.2");
```

RETURNS
The Internet address, or **ERROR**.

SEE ALSO
inetLib

inet_aton( )

NAME
inet_aton( ) – convert a network address from dot notation, store in a structure

SYNOPSIS
```
STATUS inet_aton
    (
    char *            pString,   /* string containing address, dot notation */
    struct in_addr * inetAddress /* struct in which to store address */
    )
```

DESCRIPTION
This routine interprets an Internet address. All the network library routines call this routine to interpret entries in the data bases that are expected to be an address. The value returned is stored in network byte order in the structure provided.

EXAMPLE
The following example returns 0x5a000002 in the **s_addr** member of the structure pointed to by *pinetAddr*:

```
inet_aton ("90.0.0.2", pinetAddr);
```

RETURNS **OK**, or **ERROR** if address is invalid.

SEE ALSO **inetLib**

inet_lnaof( )

NAME **inet_lnaof( )** – get the local address (host number) from the Internet address

SYNOPSIS

```
int inet_lnaof
    (
    int inetAddress           /* inet addr from which to extract local */
                              /* portion */
    )
```

DESCRIPTION This routine returns the local network address portion of an Internet address. The routine handles class A, B, and C network number formats.

EXAMPLE The following example returns 2:

```
inet_lnaof (0x5a000002);
```

RETURNS The local address portion of *inetAddress*.

SEE ALSO **inetLib**

inet_makeaddr( )

NAME **inet_makeaddr( )** – form an Internet address from network and host numbers

SYNOPSIS

```
struct in_addr inet_makeaddr
    (
    int netAddr,              /* network part of the address */
    int hostAddr              /* host part of the address */
    )
```

DESCRIPTION This routine constructs the Internet address from the network number and local host address.

WARNING: This routine is supplied for UNIX compatibility only. Each time this routine is called, four bytes are allocated from memory. Use **inet_makeaddr_b( )** instead.

EXAMPLE The following example returns the address 0x5a000002 to the structure **in_addr**:

```
inet_makeaddr (0x5a, 2);
```

RETURNS The network address in an **in_addr** structure.

SEE ALSO **inetLib**, **inet_makeaddr_b( )**

inet_makeaddr_b( )

NAME **inet_makeaddr_b( )** – form an Internet address from network and host numbers

SYNOPSIS

```
void inet_makeaddr_b
    (
    int               netAddr,   /* network part of the inet address */
    int               hostAddr,  /* host part of the inet address */
    struct in_addr *  pInetAddr  /* where to return the inet address */
    )
```

DESCRIPTION This routine constructs the Internet address from the network number and local host address. This routine is identical to the UNIX **inet_makeaddr( )** routine except that you must provide a buffer for the resulting value.

EXAMPLE The following copies the address 0x5a000002 to the location pointed to by *pInetAddr*:

```
inet_makeaddr_b (0x5a, 2, pInetAddr);
```

RETURNS N/A

SEE ALSO **inetLib**

inet_netof( )

NAME **inet_netof( )** – return the network number from an Internet address

SYNOPSIS

```
int inet_netof
    (
    struct in_addr inetAddress /* inet address */
    )
```

DESCRIPTION This routine extracts the network portion of an Internet address.

EXAMPLE The following example returns 0x5a:

```
    inet_netof (0x5a000002);
```

RETURNS The network portion of *inetAddress*.

SEE ALSO **inetLib**

inet_netof_string( )

NAME **inet_netof_string( )** – extract the network address in dot notation

SYNOPSIS

```
void inet_netof_string
    (
    char * inetString,        /* inet addr to extract local portion from */
    char * netString          /* net inet address to return */
    )
```

DESCRIPTION This routine extracts the network Internet address from a host Internet address (specified in dotted decimal notation). The routine handles class A, B, and C network addresses. The buffer *netString* should be **INET_ADDR_LEN** bytes long.

NOTE: This is the only routine in **inetLib** that handles subnet masks correctly.

EXAMPLE The following example copies "90.0.0.0" to *netString*:

```
    inet_netof_string ("90.0.0.2", netString);
```

RETURNS N/A

SEE ALSO **inetLib**

inet_network( )

NAME **inet_network( )** – convert an Internet network number from string to address

SYNOPSIS

```
u_long inet_network
    (
    char * inetString         /* string version of inet addr */
    )
```

DESCRIPTION This routine forms a network address from an ASCII string containing an Internet network number.

EXAMPLE The following example returns 0x5a:

```
    inet_network ("90");
```

RETURNS The Internet address for an ASCII string, or **ERROR** if invalid.

SEE ALSO **inetLib**

inet_ntoa( )

NAME **inet_ntoa( )** – convert a network address to dotted decimal notation

SYNOPSIS

```
char *inet_ntoa
    (
    struct in_addr inetAddress /* inet address */
    )
```

DESCRIPTION This routine converts an Internet address in network format to dotted decimal notation.

WARNING: This routine is supplied for UNIX compatibility only. Each time this routine is called, 18 bytes are allocated from memory. Use **inet_ntoa_b( )** instead.

EXAMPLE The following example returns a pointer to the string "90.0.0.2":

```
    struct in_addr iaddr;
     ...
    iaddr.s_addr = 0x5a000002;
     ...
    inet_ntoa (iaddr);
```

RETURNS A pointer to the string version of an Internet address.

SEE ALSO **inetLib**, **inet_ntoa_b( )**

inet_ntoa_b( )

NAME **inet_ntoa_b( )** – convert an network address to dot notation, store it in a buffer

SYNOPSIS

```
void inet_ntoa_b
    (
    struct in_addr inetAddress, /* inet address */
    char *         pString      /* where to return ASCII string */
    )
```

DESCRIPTION This routine converts an Internet address in network format to dotted decimal notation.

This routine is identical to the UNIX **inet_ntoa( )** routine except that you must provide a buffer of size **INET_ADDR_LEN**.

EXAMPLE The following example copies the string "90.0.0.2" to *pString*:

```
    struct in_addr iaddr;
     ...
    iaddr.s_addr = 0x5a000002;
     ...
    inet_ntoa_b (iaddr, pString);
```

RETURNS N/A

SEE ALSO **inetLib**

inetstatShow( )

NAME **inetstatShow( )** – display all active connections for Internet protocol sockets

SYNOPSIS

```
void inetstatShow (void)
```

DESCRIPTION This routine displays a list of all active Internet protocol sockets in a format similar to the UNIX **netstat** command.

If you want **inetstatShow( )** to display TCP socket status, then **INCLUDE_TCP_SHOW** needs to be included.

RETURNS N/A

SEE ALSO **netShow**

infinity( )

NAME **infinity( )** – return a very large double

SYNOPSIS

```
double infinity (void)
```

DESCRIPTION This routine returns a very large double.

INCLUDE FILES **math.h**

RETURNS The double-precision representation of positive infinity.

SEE ALSO **mathALib**

infinityf( )

NAME
infinityf( ) – return a very large float

SYNOPSIS
```
float infinityf (void)
```

DESCRIPTION
This routine returns a very large float.

INCLUDE FILES
math.h

RETURNS
The single-precision representation of positive infinity.

SEE ALSO
mathALib

inflate( )

NAME
inflate( ) – inflate compressed code

SYNOPSIS
```
int inflate
    (
    Byte * src,
    Byte * dest,
    int    nBytes
    )
```

DESCRIPTION
This routine inflates *nBytes* of data starting at address *src*. The inflated code is copied starting at address *dest*. Two sanity checks are performed on the data being decompressed. First, we look for a magic number at the start of the data to verify that it is really a compressed stream. Second, the entire data is optionally check-summed 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**.

SEE ALSO
inflateLib

intConnect( )

NAME

intConnect( ) – connect a C routine to a hardware interrupt

SYNOPSIS

```
STATUS intConnect
    (
    VOIDFUNCPTR * vector,     /* interrupt vector to attach to */
    VOIDFUNCPTR   routine,    /* routine to be called */
    int           parameter   /* parameter to be passed to routine */
    )
```

DESCRIPTION

This routine connects a specified C routine to a specified interrupt vector. The address of *routine* is generally stored at *vector* so that *routine* is called with *parameter* when the interrupt occurs. The routine is invoked in supervisor mode at interrupt level. A proper C environment is established, the necessary registers saved, and the stack set up.

The routine can be any normal C code, except that it must not invoke certain operating system functions that may block or perform I/O operations.

This routine generally simply calls **intHandlerCreate( )** and **intVecSet( )**. The address of the handler returned by **intHandlerCreate( )** is what actually goes in the interrupt vector.

This routine takes an interrupt vector as a parameter, which is the byte offset into the vector table. Macros are provided to convert between interrupt vectors and interrupt numbers, see **intArchLib**.

NOTE ARM

ARM processors generally do not have on-chip interrupt controllers. Control of interrupts is a BSP-specific matter. This routine calls a BSP-specific routine to install the handler such that, when the interrupt occurs, *routine* is called with *parameter*.

NOTE X86

Refer to the special x86 routine **intHandlerCreateI86( )**.

NOTE SH

The on-chip interrupt controller (INTC) design of SH architecture depends on the processor type, but there are some similarities. The number of external interrupt inputs are limited, so it may necessary to multiplex some interrupt requests. However most of them are auto-vectored, thus have only one vector to an external interrupt input. As a framework to handle this type of multiplexed interrupt, you can use your original **intConnect( )** code by hooking it to **_func_intConnectHook** pointer. If **_func_intConnectHook** is set, the SH version of **intConnect( )** simply calls the hooked routine with same arguments, then returns the status of hooked routine. A **sysLib** sample is shown below:

```
#include "intLib.h"
#include "iv.h"                /* INUM_INTR_HIGH for SH7750/SH7700 */
#define SYS_INT_TBL_SIZE       (255 - INUM_INTR_HIGH)
typedef struct
```

```
    {
    VOIDFUNCPTR routine;      /* routine to be called */
    int         parameter;    /* parameter to be passed */
    } SYS_INT_TBL;
LOCAL SYS_INT_TBL sysIntTbl [SYS_INT_TBL_SIZE]; /* local vector table */
LOCAL int sysInumVirtBase = INUM_INTR_HIGH + 1;
STATUS sysIntConnect
    (
    VOIDFUNCPTR *vec,         /* interrupt vector to attach to     */
    VOIDFUNCPTR routine,      /* routine to be called              */
    int         param         /* parameter to be passed to routine */
    )
    {
    FUNCPTR intDrvRtn;

    if (vec >= INUM_TO_IVEC (0) && vec < INUM_TO_IVEC (sysInumVirtBase))
        {
        /* do regular intConnect() process */
        intDrvRtn = intHandlerCreate ((FUNCPTR) routine, param);

        if (intDrvRtn == NULL)
            return ERROR;

        /* make vector point to synthesized code */

        intVecSet ((FUNCPTR *) vec, (FUNCPTR) intDrvRtn);
        }
    else
        {
        int index = IVEC_TO_INUM (vec) - sysInumVirtBase;

        if (index < 0 || index >= SYS_INT_TBL_SIZE)
            return ERROR;

        sysIntTbl [index].routine   = routine;
        sysIntTbl [index].parameter = param;
        }

    return OK;
    }
void sysHwInit (void)
    {
    ...
    _func_intConnectHook = (FUNCPTR) sysIntConnect;
    }
LOCAL void sysVmeIntr (void)
```

```
    {
    volatile UINT32 vec = *VME_VEC_REGISTER;  /* get VME interrupt vector */
    int i = vec - sysInumVirtBase;

    if (i >= 0 && i < SYS_INT_TBL_SIZE && sysIntTbl[i].routine != NULL)
        (*sysIntTbl[i].routine)(sysIntTbl[i].parameter);
    else
        logMsg ("uninitialized VME interrupt: vec = %d\n", vec,0,0,0,0,0);
    }
void sysHwInit2 (void)
    {
    int i;
    ...
    /* initialize VME interrupts dispatch table */
    for (i = 0; i < SYS_INT_TBL_SIZE; i++)
        {
        sysIntTbl[i].routine   = (VOIDFUNCPTR) NULL;
        sysIntTbl[i].parameter = NULL;
        }
    /* connect generic VME interrupts handler */
    intConnect (INT_VEC_VME, sysVmeIntr, NULL);
    ...
    }
```

The used vector numbers of SH processors are limited to certain ranges, depending on the processor type. The **sysInumVirtBase** should be initialized to a value higher than the last used vector number, defined as **INUM_INTR_HIGH**. It is typically safe to set **sysInumVirtBase** to (**INUM_INTR_HIGH** + 1).

The **sysIntConnect( )** routine simply acts as the regular **intConnect( )** if *vector* is smaller than **INUM_TO_IVEC** (sysInumVirtBase), so **sysHwInit2( )** connects a common VME interrupt dispatcher **sysVmeIntr** to the multiplexed interrupt vector. If *vector* is equal to or greater than **INUM_TO_IVEC** (sysInumVirtBase), the **sysIntConnect( )** fills a local vector entry in **sysIntTbl[]** with an individual VME interrupt handler, in a coordinated manner with **sysVmeIntr**.

RETURNS **OK**, or **ERROR** if the interrupt handler cannot be built.

SEE ALSO **intArchLib**, **intHandlerCreate( )**, **intVecSet( )**

intContext( )

NAME **intContext( )** – determine if the current state is in interrupt or task context

SYNOPSIS

```
BOOL intContext (void)
```

DESCRIPTION This routine returns **TRUE** only if the current execution state is in interrupt context and not in a meaningful task context.

RETURNS **TRUE** or **FALSE**.

SEE ALSO **intLib**

intCount( )

NAME **intCount( )** – get the current interrupt nesting depth

SYNOPSIS

```
int intCount (void)
```

DESCRIPTION This routine returns the number of interrupts that are currently nested.

RETURNS The number of nested interrupts.

SEE ALSO **intLib**

intCRGet( )

NAME **intCRGet( )** – read the contents of the cause register (MIPS)

SYNOPSIS

```
int intCRGet (void)
```

DESCRIPTION This routine reads and returns the contents of the MIPS cause register.

RETURNS The contents of the cause register.

SEE ALSO **intArchLib**

intCRSet( )

NAME **intCRSet( )** – write the contents of the cause register (MIPS)

SYNOPSIS

```
void intCRSet
    (
    int value                 /* value to write to cause register */
    )
```

DESCRIPTION This routine writes the contents of the MIPS cause register.

RETURNS N/A

SEE ALSO **intArchLib**

I

intDisable( )

NAME **intDisable( )** – disable corresponding interrupt bits (MIPS, PowerPC, ARM)

SYNOPSIS

```
int intDisable
    (
    int level                 /* new interrupt bits (0x0 - 0xff00) */
    )
```

DESCRIPTION On MIPS and PowerPC architectures, this routine disables the corresponding interrupt bits from the present status register.

NOTE: ARM processors generally do not have on-chip interrupt controllers. Control of interrupts is a BSP-specific matter. This routine calls a BSP-specific routine to disable a particular interrupt level, regardless of the current interrupt mask level.

NOTE: For MIPS, the macros **SR_IBIT1** - **SR_IBIT8** define bits that may be set.

RETURNS **OK** or **ERROR**. (MIPS: The previous contents of the status register).

SEE ALSO **intArchLib**

intEnable( )

NAME

intEnable( ) – enable corresponding interrupt bits (MIPS, PowerPC, ARM)

SYNOPSIS

```
int intEnable
    (
    int level                 /* new interrupt bits (0x00 - 0xff00) */
    )
```

DESCRIPTION

This routine enables the input interrupt bits on the present status register of the MIPS and PowerPC processors.

NOTE: ARM processors generally do not have on-chip interrupt controllers. Control of interrupts is a BSP-specific matter. This routine calls a BSP-specific routine to enable the interrupt. For each interrupt level to be used, there must be a call to this routine before it will be allowed to interrupt.

NOTE: For MIPS, it is strongly advised that the level be a combination of **SR_IBIT1** - **SR_IBIT8**.

RETURNS

OK or **ERROR**. (MIPS: The previous contents of the status register).

SEE ALSO

intArchLib

intHandlerCreate( )

NAME

intHandlerCreate( ) – construct an interrupt handler for a C routine (68K, x86, MIPS, SimSolaris)

SYNOPSIS

```
FUNCPTR intHandlerCreate
    (
    FUNCPTR routine,          /* routine to be called */
    int     parameter         /* parameter to be passed to routine */
    )
```

DESCRIPTION

This routine builds an interrupt handler around the specified C routine. This interrupt handler is then suitable for connecting to a specific vector address with **intVecSet( )**. The interrupt handler is invoked in supervisor mode at interrupt level. A proper C environment is established, the necessary registers saved, and the stack set up.

The routine can be any normal C code, except that it must not invoke certain operating system functions that may block or perform I/O operations.

RETURNS A pointer to the new interrupt handler, or **NULL** if memory is insufficient.

SEE ALSO **intArchLib**

intHandlerCreateI86( )

NAME **intHandlerCreateI86( )** – construct an interrupt handler for a C routine (x86)

SYNOPSIS

```
FUNCPTR intHandlerCreateI86
    (
    FUNCPTR routine,          /* routine to be called */
    int     parameter,        /* parameter to be passed to routine */
    FUNCPTR routineBoi,       /* BOI routine to be called */
    int     parameterBoi,     /* parameter to be passed to routineBoi */
    FUNCPTR routineEoi,       /* EOI routine to be called */
    int     parameterEoi      /* parameter to be passed to routineEoi */
    )
```

DESCRIPTION This routine builds an interrupt handler around a specified C routine. This interrupt handler is then suitable for connecting to a specific vector address with **intVecSet( )**. The interrupt handler is invoked in supervisor mode at interrupt level. A proper C environment is established, the necessary registers saved, and the stack set up.

The routine can be any normal C code, except that it must not invoke certain operating system functions that may block or perform I/O operations.

IMPLEMENTATION This routine builds an interrupt handler of the following form in allocated memory:

```
00  e8 kk kk kk kk          call    _intEnt          * tell kernel
05  50                      pushl   %eax             * save regs
06  52                      pushl   %edx
07  51                      pushl   %ecx
08  68 pp pp pp pp          pushl   $_parameterBoi   * push BOI param
13  e8 rr rr rr rr          call    _routineBoi      * call BOI routine
18  68 pp pp pp pp          pushl   $_parameter      * push param
23  e8 rr rr rr rr          call    _routine         * call C routine
28  68 pp pp pp pp          pushl   $_parameterEoi   * push EOI param
33  e8 rr rr rr rr          call    _routineEoi      * call EOI routine
38  83 c4 0c                addl    $12, %esp        * pop param
41  59                      popl    %ecx             * restore regs
```

```
42  5a                            popl    %edx
43  58                            popl    %eax
44  e9 kk kk kk kk                jmp     _intExit         * exit via kernel
```

Third and fourth parameter of **intHandlerCreateI86( )** are the BOI routine address and its parameter that are inserted into the code as “routineBoi” and “parameterBoi”. Fifth and sixth parameter of **intHandlerCreateI86( )** are the EOI routine address and its parameter that are inserted into the code as “routineEoi” and "parameterEoi". The BOI routine detects if this interrupt is stray/spurious/phantom by interrogating the interrupt controller, and returns from the interrupt if it is. The EOI routine issues End Of Interrupt signal to the interrupt controller, if it is required by the controller. Each interrupt controller has its own BOI and EOI routine. They are located in the BSP, and their address and parameter are taken by the **intEoiGet** function pointer (set to **sysIntEoiGet( )** in the BSP). The Tornado 2, and later, BSPs should use the BOI and EOI mechanism with **intEoiGet** function pointer.

To keep the Tornado 1.0.1 BSP backward compatible, the function pointer **intEOI** is not removed. If **intEoiGet** is **NULL**, it should be set to the **sysIntEoiGet( )** routine in the BSP, **intHandlerCreate( )** and the **intEOI** function pointer (set to **sysIntEOI( )** in the Tornado 101 BSP) is used.

RETURNS

A pointer to the new interrupt handler, or **NULL** if memory is insufficient.

SEE ALSO

intArchLib

---

intLevelSet( )

NAME

intLevelSet( ) – set the interrupt level (68K, x86, ARM, SimSolaris, SimNT and SH)

SYNOPSIS

```
int intLevelSet
    (
    int level                   /* new interrupt level mask */
    )
```

DESCRIPTION

This routine changes the interrupt mask in the status register to take on the value specified by *level*. Interrupts are locked out at or below that level. The value of *level* must be in the following range:

MC680x0:	0 - 7
SH:	0 - 15
ARM:	BSP-specific
SimSolaris:	0 - 1
x86:	interrupt controller specific

On x86 systems, there are no interrupt level in the processor and the external interrupt controller manages the interrupt level. Therefore this routine does nothing and returns **OK** always.

NOTE: With the NT simulator, this routine does nothing.

WARNING: Do not call VxWorks system routines with interrupts locked. Violating this rule may re-enable interrupts unpredictably.

RETURNS The previous interrupt level.

SEE ALSO **intArchLib**

intLock( )

NAME **intLock( )** – lock out interrupts

SYNOPSIS

```
int intLock (void)
```

DESCRIPTION This routine disables interrupts. The **intLock( )** routine returns an architecture-dependent lock-out key representing the interrupt level prior to the call; this key can be passed to **intUnlock( )** to re-enable interrupts.

For MC680x0, x86, and SH architectures, interrupts are disabled at the level set by **intLockLevelSet( )**. The default lock-out level is the highest interrupt level (MC680x0 = 7, x86 = 1, SH = 15).

For SimSolaris architecture, interrupts are masked. Lock-out level returned is 1 if interrupts were already locked, 0 otherwise.

For SimNT, a windows semaphore is used to lock the interrupts. Lock-out level returned is 1 if interrupts were already locked, 0 otherwise.

For MIPS processors, interrupts are disabled at the master lock-out level; this means no interrupt can occur even if unmasked in the *IntMask* bits (15-8) of the status register.

For ARM processors, interrupts (IRQs) are disabled by setting the I bit in the CPSR. This means no IRQs can occur.

For PowerPC processors, there is only one interrupt vector. The external interrupt (vector offset 0x500) is disabled when **intLock( )** is called; this means that the processor cannot be interrupted by any external event.

IMPLEMENTATION The lock-out key is implemented differently for different architectures:

MC680x0:	interrupt field mask
MIPS:	status register
x86:	interrupt enable flag (IF) bit from EFLAGS register
PowerPC:	MSR register value
RM	I bit from the CPSR
H:	status register
SimSolaris:	1 or 0
IMNT:	1 or 0

WARNING: Do not call VxWorks system routines with interrupts locked. Violating this rule may re-enable interrupts unpredictably.

The routine **intLock( )** can be called from either interrupt or task level. When called from a task context, the interrupt lock level is part of the task context. Locking out interrupts does not prevent rescheduling. Thus, if a task locks out interrupts and invokes kernel services that cause the task to block (*e.g.*, **taskSuspend( )** or **taskDelay( )**) or that cause a higher priority task to be ready (*e.g.*, **semGive( )** or **taskResume( )**), then rescheduling occurs and interrupts are unlocked while other tasks run. Rescheduling may be explicitly disabled with **taskLock( )**. Traps must be enabled when calling this routine.

EXAMPLES

```
lockKey = intLock ();
 ... (work with interrupts locked out)
intUnlock (lockKey);
```

To lock out interrupts and task scheduling as well (see WARNING above):

```
if (taskLock() == OK)
    {
    lockKey = intLock ();
    ... (critical section)
    intUnlock (lockKey);
    taskUnlock();
    }
 else
    {
    ... (error message or recovery attempt)
    }
```

RETURNS

An architecture-dependent lock-out key for the interrupt level prior to the call.

SEE ALSO

intArchLib, **intUnlock( )**, **taskLock( )**, **intLockLevelSet( )**

intLockLevelGet( )

NAME

intLockLevelGet( ) – get the current interrupt lock-out level (68K, x86, ARM, SH, SimSolaris, SimNT)

SYNOPSIS

```
int intLockLevelGet (void)
```

DESCRIPTION

This routine returns the current interrupt lock-out level, which is set by **intLockLevelSet( )** and stored in the globally accessible variable **intLockMask**. This is the interrupt level currently masked when interrupts are locked out by **intLock( )**. The default lock-out level (MC680x0 = 7, x86 = 1, SH = 15) is initially set by **kernelInit( )** when VxWorks is initialized.

NOTE: With the NT simulator, this routine does nothing.

RETURNS

The interrupt level currently stored in the interrupt lock-out mask. (ARM = **ERROR** always)

SEE ALSO

intArchLib, **intLockLevelSet( )**

intLockLevelSet( )

NAME

intLockLevelSet( ) – set the current interrupt lock-out level (68K, x86, ARM, SH, SimSolaris, SimNT)

SYNOPSIS

```
void intLockLevelSet
    (
    int newLevel                /* new interrupt level */
    )
```

DESCRIPTION

This routine sets the current interrupt lock-out level and stores it in the globally accessible variable **intLockMask**. The specified interrupt level is masked when interrupts are locked by **intLock( )**. The default lock-out level (MC680x0 = 7, x86 = 1, SH = 15) is initially set by **kernelInit( )** when VxWorks is initialized.

NOTE: With SimSolaris and SimNT, this routine does nothing.

NOTE: On the ARM, this call establishes the interrupt level to be set when **intLock( )** is called.

RETURNS N/A

SEE ALSO **intArchLib**, **intLockLevelGet( )**, **intLock( )**, **taskLock( )**

intSRGet( )

NAME **intSRGet( )** – read the contents of the status register (MIPS)

SYNOPSIS

```
int intSRGet (void)
```

DESCRIPTION This routine reads and returns the contents of the MIPS status register.

RETURNS The previous contents of the status register.

SEE ALSO **intArchLib**

intSRSet( )

NAME **intSRSet( )** – update the contents of the status register (MIPS)

SYNOPSIS

```
int intSRSet
    (
    int value                 /* value to write to status register */
    )
```

DESCRIPTION This routine updates and returns the previous contents of the MIPS status register.

RETURNS The previous contents of the status register.

SEE ALSO **intArchLib**

intStackEnable( )

NAME

intStackEnable( ) – enable or disable the interrupt stack usage (x86)

SYNOPSIS

```
STATUS intStackEnable
    (
    BOOL enable                /* TRUE to enable, FALSE to disable */
    )
```

DESCRIPTION

This routine enables or disables the interrupt stack usage and is only callable from the task level. An Error is returned for any other calling context. The interrupt stack usage is disabled in the default configuration for the backward compatibility. Routines that manipulate the interrupt stack, are located in the file **i86/windALib.s**. These routines include **intStackEnable( )**, **intEnt( )** and **intExit( )**.

RETURNS

OK, or **ERROR** if it is not in the task level.

SEE ALSO

intArchLib

intUninitVecSet( )

NAME

intUninitVecSet( ) – set the uninitialized vector handler (ARM)

SYNOPSIS

```
void intUninitVecSet
    (
    VOIDFUNCPTR routine        /* ptr to user routine */
    )
```

DESCRIPTION

This routine installs a handler for the uninitialized vectors to be called when any uninitialized vector is entered.

RETURNS

N/A.

SEE ALSO

intArchLib

intUnlock( )

NAME

intUnlock( ) – cancel interrupt locks

SYNOPSIS

```
void intUnlock
    (
    int lockKey          /* lock-out key returned by preceding intLock() */
    )
```

DESCRIPTION

This routine re-enables interrupts that have been disabled by **intLock( )**. The parameter *lockKey* is an architecture-dependent lock-out key returned by a preceding **intLock( )** call.

RETURNS

N/A

SEE ALSO

intArchLib, **intLock( )**

intVecBaseGet( )

NAME

intVecBaseGet( ) – get the vector (trap) base address (68K, x86, MIPS, ARM, SimSolaris, SimNT)

SYNOPSIS

```
FUNCPTR *intVecBaseGet (void)
```

DESCRIPTION

This routine returns the current vector base address, which is set with **intVecBaseSet( )**.

RETURNS

The current vector base address (MIPS = 0 always, ARM = 0 always, SimSolaris = 0 always and SimNT = 0 always).

SEE ALSO

intArchLib, **intVecBaseSet( )**

intVecBaseSet( )

NAME **intVecBaseSet( )** – set the vector (trap) base address (68K, x86, MIPS, ARM, SimSolaris, SimNT)

SYNOPSIS

```
void intVecBaseSet
    (
    FUNCPTR * baseAddr        /* new vector (trap) base address */
    )
```

DESCRIPTION This routine sets the vector (trap) base address. The CPU's vector base register is set to the specified value, and subsequent calls to **intVecGet( )** or **intVecSet( )** will use this base address. The vector base address is initially 0, until modified by calls to this routine.

NOTE 68000 The 68000 has no vector base register; thus, this routine is a no-op for 68000 systems.

NOTE MIPS The MIPS processors have no vector base register; thus this routine is a no-op for this architecture.

NOTE SH77XX This routine sets *baseAddr* to vbr, then loads an interrupt dispatch code to (vbr + 0x600). When SH77XX processor accepts an interrupt request, it sets an exception code to INTEVT register and jumps to (vbr + 0x600). Thus this dispatch code is commonly used for all interrupts' handling.

The exception codes are 12bits width, and interleaved by 0x20. VxWorks for SH77XX locates a vector table at (vbr + 0x800), and defines the vector offsets as (exception codes / 8). This vector table is commonly used by all interrupts, exceptions, and software traps.

All SH77XX processors have INTEVT register at address 0xffffffd8. The SH7707 processor has yet another INTEVT2 register at address 0x04000000, to identify its enhanced interrupt sources. The dispatch code obtains the address of INTEVT register from a global constant **intEvtAdrs**. The constant is defined in **sysLib**, thus the selection of INTEVT/INTEVT2 is configurable at BSP level. The **intEvtAdrs** is loaded to (vbr + 4) by **intVecBaseSet( )**.

After fetching the exception code, the interrupt dispatch code applies a new interrupt mask to the status register, and jumps to an individual interrupt handler. The new interrupt mask is taken from **intPrioTable[]**, which is defined in **sysALib**. The **intPrioTable[]** is loaded to (vbr + 0xc00) by **intVecBaseSet( )**.

NOTE ARM The ARM processors have no vector base register; thus this routine is a no-op for this architecture.

NOTE SIMSOLARIS, SIMNT

This routine does nothing.

RETURNS N/A

SEE ALSO **intArchLib**, **intVecBaseGet( )**, **intVecGet( )**, **intVecSet( )**

intVecGet( )

NAME **intVecGet( )** – get an interrupt vector (68K, x86, MIPS, SH, SimSolaris, SimNT)

SYNOPSIS

```
FUNCPTR intVecGet
    (
    FUNCPTR * vector          /* vector offset */
    )
```

DESCRIPTION This routine returns a pointer to the exception/interrupt handler attached to a specified vector. The vector is specified as an offset into the CPU's vector table. This vector table starts, by default, at:

C680x0:	0
MIPS:	**excBsrTbl** in **excArchLib**
86:	0
SH702x/SH703x/SH704x/SH76xx:	**excBsrTbl** in **excArchLib**
SH77xx:	vbr + 0x800
SimSolaris:	0

However, the vector table may be set to start at any address with **intVecBaseSet( )** (on CPUs for which it is available).

This routine takes an interrupt vector as a parameter, which is the byte offset into the vector table. Macros are provided to convert between interrupt vectors and interrupt numbers, see **intArchLib**.

NOTE SIMNT This routine does nothing and always returns 0.

RETURNS A pointer to the exception/interrupt handler attached to the specified vector.

SEE ALSO **intArchLib**, **intVecSet( )**, **intVecBaseSet( )**

intVecGet2( )

NAME **intVecGet2( )** – get a CPU vector, gate type(int/trap), and gate selector (x86)

SYNOPSIS

```
void intVecGet2
    (
    FUNCPTR * vector,          /* vector offset */
    FUNCPTR * pFunction,       /* address to place in vector */
    int *     pIdtGate,        /* IDT_TRAP_GATE or IDT_INT_GATE */
    int *     pIdtSelector     /* sysCsExc or sysCsInt */
    )
```

DESCRIPTION This routine gets a pointer to the exception/interrupt handler attached to a specified vector, the type of the gate, the selector of the gate. The vector is specified as an offset into the CPU's vector table. This vector table starts, by default, at address 0. However, the vector table may be set to start at any address with **intVecBaseSet( )**.

RETURNS N/A

SEE ALSO **intArchLib**, **intVecBaseSet( )**, **intVecGet( )**, **intVecSet( )**, **intVecSet2( )**

intVecSet( )

NAME **intVecSet( )** – set a CPU vector (trap) (68K, x86, MIPS, SH, SimSolaris, SimNT)

SYNOPSIS

```
void intVecSet
    (
    FUNCPTR * vector,          /* vector offset */
    FUNCPTR   function         /* address to place in vector */
    )
```

DESCRIPTION This routine attaches an exception/interrupt/trap handler to a vector. The vector is specified as an offset into the CPU's vector table. By default the vector table starts at:

MC680x0:	0
MIPS:	**excBsrTbl** in **excArchLib**
x86:	0
SH702x/SH703x/SH704x/SH76xx:	**excBsrTbl** in **excArchLib**
SH77xx:	vbr + 0x800
SimSolaris:	0

However, the vector table may be set to start at any address with **intVecBaseSet( )** (on CPUs for which it is available). The vector table is set up in **usrInit( )**.

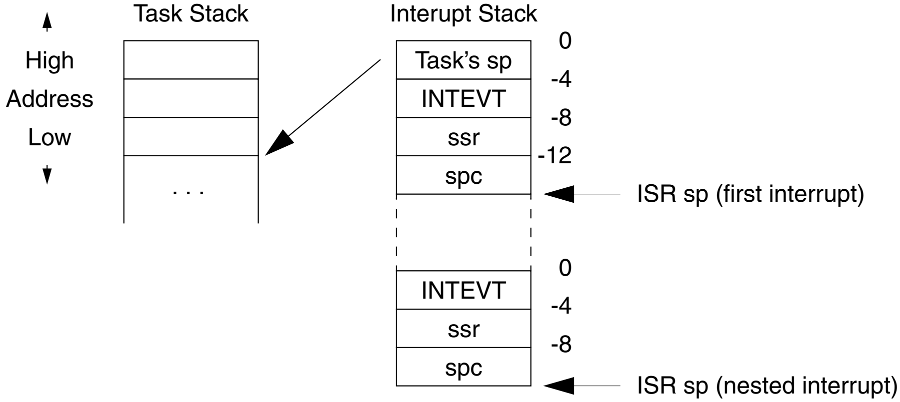
This routine takes an interrupt vector as a parameter, which is the byte offset into the vector table. Macros are provided to convert between interrupt vectors and interrupt numbers, see **intArchLib**.

NOTE MIPS

On MIPS CPUs the vector table is set up statically in software.

NOTE SH77XX

The specified interrupt handler *function* has to coordinate with an interrupt stack frame which is specially designed for the SH77XX version of VxWorks:



This interrupt stack frame is formed by a common interrupt dispatch code which is loaded at (vbr + 0x600). You usually do not have to pay any attention to this stack frame, since **intConnect( )** automatically appends an appropriate stack manipulation code to your interrupt service routine. The **intConnect( )** assumes that your interrupt service routine (ISR) is written in C, thus it also wraps your ISR in minimal register save/restore codes. However if you need a very fast response time to a particular interrupt request, you might want to skip this register save/restore sequence by directly attaching your ISR to the corresponding vector table entry using **intVecSet( )**. Note that this technique is only applicable to an interrupt service with NO VxWorks system call. For example it is not allowed to use **semGive( )** or **logMsg( )** in the interrupt service routine which is directly attached to vector table by **intVecSet( )**. To facilitate the direct usage of **intVecSet( )** by user, a special entry point to exit an interrupt context is provided within the SH77XX version of VxWorks kernel. This entry point is located at address (vbr + intRte1W), here the intRte1W is a global symbol for the vbr offset of the entry point in 16 bit length. This entry point **intRte1** assumes that the current register bank is 0 (SR.RB == 0), and r1 and r0 are still saved on the interrupt stack, and it also requires 0x70000000 in r0. Then **intRte1** properly cleans up the interrupt stack and executes *rte* instruction to return to the previous interrupt or task context. The following code is an example of **intRte1**usage.

Here the corresponding **intPrioTable[]** entry is assumed to be 0x400000X0, namely MD=1, RB=0, BL=0 at the beginning of **usrIsr1**.

```
      .text
      .align  2
      .global _usrIsr1
      .type   _usrIsr1,@function
      .extern _usrRtn
      .extern intRte1W
                                /* intPrioTable[] sets SR to 0x400000X0 */
_usrIsr1:
      mov.l   r0,@-sp           /* must save r0 first (BANK0) */
      mov.l   r1,@-sp           /* must save r1 second (BANK0) */
      mov.l   r2,@-sp           /* save rest of volatile registers (BANK0) */
      mov.l   r3,@-sp
      mov.l   r4,@-sp
      mov.l   r5,@-sp
      mov.l   r6,@-sp
      mov.l   r7,@-sp
      sts.l   pr,@-sp
      sts.l   mach,@-sp
      sts.l   macl,@-sp
      mov.l   UsrRtn,r0
      jsr     @r0               /* call user's C routine */
      nop                       /* (delay slot) */
      lds.l   @sp+,macl         /* restore volatile registers (BANK0) */
      lds.l   @sp+,mach
      lds.l   @sp+,pr
      mov.l   @sp+,r7
      mov.l   @sp+,r6
      mov.l   @sp+,r5
      mov.l   @sp+,r4
      mov.l   @sp+,r3
      mov.l   @sp+,r2
                                /* intRte1 restores r1 and r0 */
      mov.l   IntRte1W,r1
      mov.w   @r1,r0
      stc     vbr,r1
      add     r0,r1
      mov.l   IntRteSR,r0       /* r0: 0x70000000 */
      jmp     @r1               /* let intRte1 clean up stack, then rte */
      nop                       /* (delay slot) */
              .align  2
UsrRtn:       .long   _usrRtn           /* user's C routine */
IntRteSR:     .long   0x70000000        /* MD=1, RB=1, BL=1 */
IntRte1W:     .long   intRte1W
```

The **intRte1** sets r0 to status register (SR: 0x70000000), to safely restore SPC/SSR and to clean up the interrupt stack. Note that TLB mis-hit exception immediately reboots CPU while SR.BL=1. To avoid this fatal condition, VxWorks loads the **intRte1** code and the interrupt stack to a physical address space (P1) where no TLB mis-hit happens.

Furthermore, there is another special entry point called **intRte2** at an address (vbr + intRte2W). The **intRte2** assumes that SR is already set to 0x70000000 (MD: 1, RB: 1, BL: 1), then it does not restore r1 and r0. While SR value is 0x70000000, you may use r0,r1,r2,r3 in BANK1 as volatile registers. The rest of BANK1 registers (r4,r5,r6,r7) are non-volatile, so if you need to use them then you have to preserve their original values by saving/restoring them on the interrupt stack. So, if you need the ultimate interrupt response time, you may set the corresponding **intPrioTable[]** entry to **NULL** and manage your interrupt service only with r0,r1,r2,r3 in BANK1 as shown in the next sample code:

```
      .text
      .global  _usrIsr2
      .type    _usrIsr2,@function
      .extern  _usrIntCnt     /* interrupt counter */
      .extern  intRte2W
      .align   2
                              /* MD=1, RB=1, BL=1, since SR is not */
                              /* substituted from intPrioTable[].  */
_usrIsr2:
      mov.l   UsrIntAck,r1
      mov     #0x1,r0
      mov.b   r0,@r1          /* acknowledge interrupt */
      mov.l   UsrIntCnt,r1
      mov.l   X1FFFFFFF,r2
      mov.l   X80000000,r3
      and     r2,r1
      or      r3,r1           /* r1: _usrIntCnt address in P1 */
      mov.l   @r1,r0
      add     #1,r0
      mov.l   r0,@r1          /* increment counter */
      mov.l   IntRte2W,r1
      and     r2,r1
      or      r3,r1           /* r1: intRte2W address in P1 */
      mov.w   @r1,r0
      stc     vbr,r1
      add     r1,r0
      jmp     @r0             /* let intRte2 clean up stack, then rte */
      nop                     /* (delay slot) */
              .align  2
UsrIntAck:    .long   0xa0001234      /* interrupt acknowledge register */
UsrIntCnt:    .long   _usrIntCnt
IntRte2W:     .long   intRte2W
X1FFFFFFF:    .long   0x1fffffff
```

```
X80000000:    .long   0x80000000
```

Note that the entire interrupt service is executed under SR.BL=1 in this sample code. It means that any access to virtual address space may reboot CPU, since TLB mis-hit exception is blocked. Therefore **usrIsr2** has to access **usrIntCnt** and **intRte2W** from P1 region. Also **usrIsr2** itself has to be executed on P1 region, and it can be done by relocating the address of **usrIsr2** to P1 as shown below:

```
IMPORT void usrIsr2 (void);
intVecSet (vector, (FUNCPTR)(((UINT32) usrIsr2 & 0x1fffffff) | 0x80000000));
```

In conclusion, you have to guarantee that the entire ISR does not access to any virtual address space if you set the corresponding **intPrioTable[]** entry to **NULL**.

NOTE SIMNT This routine does nothing.

RETURNS N/A

SEE ALSO **intArchLib**, **intVecBaseSet( )**, **intVecGet( )**

I

---

intVecSet2( )

NAME **intVecSet2( )** – set a CPU vector, gate type(int/trap), and selector (x86)

SYNOPSIS

```
void intVecSet2
    (
    FUNCPTR * vector,         /* vector offset */
    FUNCPTR   function,       /* address to place in vector */
    int       idtGate,        /* IDT_TRAP_GATE or IDT_INT_GATE */
    int       idtSelector     /* sysCsExc or sysCsInt */
    )
```

DESCRIPTION This routine attaches an exception handler to a specified vector, with the type of the gate and the selector of the gate. The vector is specified as an offset into the CPU's vector table. This vector table starts, by default, at address 0. However, the vector table may be set to start at any address with **intVecBaseSet( )**. The vector table is set up in **usrInit( )**.

RETURNS N/A

SEE ALSO **intArchLib**, **intVecBaseSet( )**, **intVecGet( )**, **intVecSet( )**, **intVecGet2( )**

intVecTableWriteProtect( )

NAME

intVecTableWriteProtect( ) – write-protect exception vector table (68K, x86, ARM, SimSolaris, SimNT)

SYNOPSIS

```
STATUS intVecTableWriteProtect (void)
```

DESCRIPTION

If the unbundled Memory Management Unit (MMU) support package (VxVMI) is present, this routine write-protects the exception vector table to protect it from being accidentally corrupted.

Note that other data structures contained in the page will also be write-protected. In the default VxWorks configuration, the exception vector table is located at location 0 in memory. Write-protecting this affects the backplane anchor, boot configuration information, and potentially the text segment (assuming the default text location of 0x1000.) All code that manipulates these structures has been modified to write-enable memory for the duration of the operation. If you select a different address for the exception vector table, be sure it resides in a page separate from other writable data structures.

NOTE: This routine always returns **ERROR** on simulators.

RETURNS

OK, or **ERROR** if memory cannot be write-protected.

ERRNO

S_intLib_VEC_TABLE_WP_UNAVAILABLE

SEE ALSO

intArchLib

ioctl( )

NAME

ioctl( ) – perform an I/O control function

SYNOPSIS

```
int ioctl
    (
    int fd,                    /* file descriptor */
    int function,              /* function code */
    int arg                    /* arbitrary argument */
    )
```

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**, **rt11FsLib**, 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.

SEE ALSO **ioLib**, **tyLib**, **pipeDrv**, **nfsDrv**, **dosFsLib**, **rt11FsLib**, **rawFsLib**, *VxWorks Programmer's Guide: I/O System, Local File Systems*

I

ioDefPathGet( )

NAME **ioDefPathGet( )** – get the current default path

SYNOPSIS

```
void ioDefPathGet
    (
    char * pathname            /* where to return the name */
    )
```

DESCRIPTION This routine copies the name of the current default path to *pathname*. The parameter *pathname* should be **MAX_FILENAME_LENGTH** characters long.

RETURNS N/A

SEE ALSO **ioLib**, **ioDefPathSet( )**, **chdir( )**, **getcwd( )**

ioDefPathSet( )

NAME **ioDefPathSet( )** – set the current default path

SYNOPSIS

```
STATUS ioDefPathSet
    (
    char * name              /* name of the new default device and 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.

SEE ALSO **ioLib**, **ioDefPathGet( )**, **chdir( )**, **getcwd( )**

ioGlobalStdGet( )

NAME **ioGlobalStdGet( )** – get the file descriptor for global standard input/output/error

SYNOPSIS

```
int ioGlobalStdGet
    (
    int stdFd                /* std input (0), output (1), or error (2) */
    )
```

DESCRIPTION This routine returns the current underlying file descriptor for global standard input, output, and error.

RETURNS The underlying global file descriptor, or **ERROR** if *stdFd* is not 0, 1, or 2.

SEE ALSO **ioLib**, **ioGlobalStdSet( )**, **ioTaskStdGet( )**

ioGlobalStdSet( )

NAME **ioGlobalStdSet( )** – set the file descriptor for global standard input/output/error

SYNOPSIS

```
void ioGlobalStdSet
    (
    int stdFd,                /* std input (0), output (1), or error (2) */
    int newFd                 /* new underlying file descriptor */
    )
```

DESCRIPTION This routine changes the assignment of a specified global standard file descriptor *stdFd* (0, 1, or, 2) to the specified underlying file descriptor *newFd*. *newFd* should be a file descriptor open to the desired device or file. All tasks will use this new assignment when doing I/O to *stdFd*, unless they have specified a task-specific standard file descriptor (see **ioTaskStdSet( )**). If *stdFd* is not 0, 1, or 2, this routine has no effect.

RETURNS N/A

SEE ALSO **ioLib**, **ioGlobalStdGet( )**, **ioTaskStdSet( )**

ioHelp( )

NAME **ioHelp( )** – print a synopsis of I/O utility functions

SYNOPSIS

```
void ioHelp (void)
```

DESCRIPTION This function prints out synopsis for the I/O and File System utility functions.

RETURNS N/A

SEE ALSO **usrFsLib**, *VxWorks Programmer's Guide: Target Shell*

I

iosDevAdd( )

NAME **iosDevAdd( )** – add a device to the I/O system

SYNOPSIS

```
STATUS iosDevAdd
    (
    DEV_HDR * pDevHdr, /* pointer to device's structure */
    char *    name,    /* name of device */
    int       drvnum   /* # of servicing driver, ret'd by iosDrvInstall() */
    )
```

DESCRIPTION This routine adds a device to the I/O system device list, making the device available for subsequent **open( )** and **creat( )** calls.

The parameter *pDevHdr* is a pointer to a device header, **DEV_HDR** (defined in **iosLib.h**), which is used as the node in the device list. Usually this is the first item in a larger device structure for the specific device type. The parameters *name* and *drvnum* are entered in *pDevHdr*.

RETURNS **OK**, or **ERROR** if there is already a device with the specified name.

SEE ALSO **iosLib**

iosDevDelete( )

NAME **iosDevDelete( )** – delete a device from the I/O system

SYNOPSIS

```
void iosDevDelete
    (
    DEV_HDR * pDevHdr         /* pointer to device's structure */
    )
```

DESCRIPTION This routine deletes a device from the I/O system device list, making it unavailable to subsequent **open( )** or **creat( )** calls. No interaction with the driver occurs, and any file descriptors open on the device or pending operations are unaffected.

If the device was never added to the device list, unpredictable results may occur.

RETURNS N/A

SEE ALSO **iosLib**

iosDevFind( )

NAME **iosDevFind( )** – find an I/O device in the device list

SYNOPSIS

```
DEV_HDR *iosDevFind
    (
    char * name,                /* name of the device */
    char * *pNameTail           /* where to put ptr to tail of name */
    )
```

DESCRIPTION This routine searches the device list for a device whose name matches the first portion of *name*. If a device is found, **iosDevFind( )** sets the character pointer pointed to by *pNameTail* to point to the first character in *name*, following the portion which matched the device name. It then returns a pointer to the device. If the routine fails, it returns a pointer to the default device (that is, the device where the current working directory is mounted) and sets *pNameTail* to point to the beginning of *name*. If there is no default device, **iosDevFind( )** returns **NULL**.

RETURNS A pointer to the device header, or **NULL** if the device is not found.

SEE ALSO **iosLib**

iosDevShow( )

NAME **iosDevShow( )** – display the list of devices in the system

SYNOPSIS

```
void iosDevShow (void)
```

DESCRIPTION This routine displays a list of all devices in the device list.

RETURNS N/A

SEE ALSO **iosShow**, **devs( )**, *VxWorks Programmer's Guide: I/O System*, **windsh**, *Tornado User's Guide: Shell*

iosDrvInstall( )

NAME **iosDrvInstall( )** – install an I/O driver

SYNOPSIS

```
int iosDrvInstall
    (
    FUNCPTR pCreate,          /* pointer to driver create function */
    FUNCPTR pDelete,          /* pointer to driver delete function */
    FUNCPTR pOpen,            /* pointer to driver open function */
    FUNCPTR pClose,           /* pointer to driver close function */
    FUNCPTR pRead,            /* pointer to driver read function */
    FUNCPTR pWrite,           /* pointer to driver write function */
    FUNCPTR pIoctl            /* pointer to driver ioctl function */
    )
```

DESCRIPTION This routine should be called once by each I/O driver. It hooks up the various I/O service calls to the driver service routines, assigns the driver a number, and adds the driver to the driver table.

RETURNS The driver number of the new driver, or **ERROR** if there is no room for the driver.

SEE ALSO **iosLib**

iosDrvRemove( )

NAME **iosDrvRemove( )** – remove an I/O driver

SYNOPSIS

```
STATUS iosDrvRemove
    (
    int  drvnum,              /* no. of driver to remove, returned by */
                              /* iosDrvInstall() */
    BOOL forceClose           /* if TRUE, force closure of open files */
    )
```

DESCRIPTION This routine removes an I/O driver (added by **iosDrvInstall( )**) from the driver table.

RETURNS **OK**, or **ERROR** if the driver has open files.

SEE ALSO **iosLib**, **iosDrvInstall( )**

iosDrvShow( )

NAME **iosDrvShow( )** – display a list of system drivers

SYNOPSIS

```
void iosDrvShow (void)
```

DESCRIPTION This routine displays a list of all drivers in the driver list.

RETURNS N/A

SEE ALSO **iosShow**, *VxWorks Programmer's Guide: I/O System*, **windsh**, *Tornado User's Guide: Shell*

iosFdShow( )

NAME **iosFdShow( )** – display a list of file descriptor names in the system

SYNOPSIS

```
void iosFdShow (void)
```

DESCRIPTION This routine displays a list of all file descriptors in the system.

RETURNS N/A

SEE ALSO **iosShow**, **ioctl( )**, *VxWorks Programmer's Guide: I/O System*, **windsh**, *Tornado User's Guide: Shell*

iosFdValue( )

NAME **iosFdValue( )** – validate an open file descriptor and return the driver-specific value

SYNOPSIS

```
int iosFdValue
    (
    int fd                    /* file descriptor to check */
    )
```

DESCRIPTION This routine checks to see if a file descriptor is valid and returns the driver-specific value.

RETURNS The driver-specific value, or **ERROR** if the file descriptor is invalid.

SEE ALSO **iosLib**

iosInit( )

NAME **iosInit( )** – initialize the I/O system

SYNOPSIS

```
STATUS iosInit
    (
    int    max_drivers,       /* maximum number of drivers allowed */
    int    max_files,         /* max number of files allowed open at once */
    char * nullDevName        /* name of the null device (bit bucket) */
    )
```

DESCRIPTION This routine initializes the I/O system. It must be called before any other I/O system routine.

RETURNS **OK**, or **ERROR** if memory is insufficient.

SEE ALSO **iosLib**

iosShowInit( )

NAME **iosShowInit( )** – initialize the I/O system show facility

SYNOPSIS

```
void iosShowInit (void)
```

DESCRIPTION This routine links the I/O system show facility into the VxWorks system. It is called automatically when **INCLUDE_SHOW_ROUTINES** is defined in **configAll.h**.

RETURNS N/A

SEE ALSO **iosShow**

ioTaskStdGet( )

NAME **ioTaskStdGet( )** – get the file descriptor for task standard input/output/error

SYNOPSIS

```
int ioTaskStdGet
    (
    int taskId,            /* ID of desired task (0 = self) */
    int stdFd              /* std input (0), output (1), or error (2) */
    )
```

DESCRIPTION This routine returns the current underlying file descriptor for task-specific standard input, output, and error.

RETURNS The underlying file descriptor, or **ERROR** if *stdFd* is not 0, 1, or 2, or the routine is called at interrupt level.

SEE ALSO **ioLib**, **ioGlobalStdGet( )**, **ioTaskStdSet( )**

ioTaskStdSet( )

NAME **ioTaskStdSet( )** – set the file descriptor for task standard input/output/error

SYNOPSIS

```
void ioTaskStdSet
    (
    int taskId,            /* task whose std fd is to be set (0 = self) */
    int stdFd,             /* std input (0), output (1), or error (2) */
    int newFd              /* new underlying file descriptor */
    )
```

DESCRIPTION This routine changes the assignment of a specified task-specific standard file descriptor *stdFd* (0, 1, or, 2) to the specified underlying file descriptor *newFd*. *newFd* should be a file descriptor open to the desired device or file. The calling task will use this new assignment when doing I/O to *stdFd*, instead of the system-wide global assignment which is used by default. If *stdFd* is not 0, 1, or 2, this routine has no effect.

NOTE: This routine has no effect if it is called at interrupt level.

RETURNS N/A

SEE ALSO **ioLib**, **ioGlobalStdGet( )**, **ioTaskStdGet( )**

ipAttach( )

NAME

ipAttach( ) – a generic attach routine for the TCP/IP network stack

SYNOPSIS

```
int ipAttach
    (
    int    unit,                /* Unit number */
    char * pDevice              /* Device name (i.e. ln, ei etc.). */
    )
```

DESCRIPTION

This routine takes the unit number and device name of an END or NPT driver (*e.g.*, “ln0”, “ei0”, etc.) and attaches the IP protocol to the corresponding device. Following a successful attachment IP will begin receiving packets from the devices.

RETURNS

OK or **ERROR**

SEE ALSO

ipProto

ipDetach( )

NAME

ipDetach( ) – a generic detach routine for the TCP/IP network stack

SYNOPSIS

```
STATUS ipDetach
    (
    int    unit,                /* Unit number */
    char * pDevice              /* Device name (i.e. ln, ei etc.). */
    )
```

DESCRIPTION

This routine removes the TCP/IP stack from the MUX. If completed successfully, the IP protocol will no longer receive packets from the named END driver.

RETURNS

OK or **ERROR**

SEE ALSO

ipProto

ipFilterHookAdd( )

NAME

ipFilterHookAdd( ) – add a routine to receive all internet protocol packets

SYNOPSIS

```
STATUS ipFilterHookAdd
    (
    FUNCPTR ipFilterHook      /* routine to receive raw IP packets */
    )
```

DESCRIPTION

This routine adds a hook routine that will be called for every IP packet that is received.

The filter hook routine should be of the form:

```
BOOL ipFilterHook
    (
    struct ifnet *pIf,        /* interface that received the packet */
    struct mbuf  **pPtrMbuf,  /* pointer to pointer to an mbuf chain */
    struct ip    **pPtrIpHdr, /* pointer to pointer to IP header */
    int          ipHdrLen,    /* IP packet header length */
    )
```

The hook routine should return **TRUE** if it has handled the input packet. A returned value of **TRUE** effectively consumes the packet from the viewpoint of IP, which will never see the packet. As a result, when the filter hook returns **TRUE**, it must handle the freeing of any resources associated with the packet. For example, the filter hook routine would be responsible for freeing the packet's **mbuf** chain by calling **m_freem(*pPtrMbuf)**.

The filter hook routine should return **FALSE** if it has not handled the packet. In response to a **FALSE**, the network stack submits the packet for normal IP processing.

Within the packet's IP header (the filter hook can obtain a pointer to the IP header by de-referencing **pPtrIpHdr**), you will find that the values in the **ip_len** field, the **ip_id** field, and **ip_offset** field have been converted to the host byte order before the packet was handed to the filter hook.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call **ipFilterHookAdd( )** from within the kernel protection domain only, and the function referenced in the *ipFilterHook* parameter must reside in the kernel protection domain. This restriction does not apply to non-AE versions of VxWorks.

RETURNS

OK, always.

SEE ALSO

ipFilterLib

I

ipFilterHookDelete( )

NAME **ipFilterHookDelete( )** – delete a IP filter hook routine

SYNOPSIS

```
void ipFilterHookDelete (void)
```

DESCRIPTION This routine deletes an IP filter hook.

RETURNS N/A

SEE ALSO **ipFilterLib**

ipFilterLibInit( )

NAME **ipFilterLibInit( )** – initialize IP filter facility

SYNOPSIS

```
void ipFilterLibInit (void)
```

DESCRIPTION This routine links the IP filter facility into the VxWorks system. These routines are included automatically if **INCLUDE_IP_FILTER** is defined.

RETURNS N/A

SEE ALSO **ipFilterLib**

ipstatShow( )

NAME **ipstatShow( )** – display IP statistics

SYNOPSIS

```
void ipstatShow
    (
    BOOL zero                  /* TRUE = reset statistics to 0 */
    )
```

DESCRIPTION This routine displays detailed statistics for the IP protocol.

RETURNS N/A

SEE ALSO **netShow**

irint( )

NAME **irint( )** – convert a double-precision value to an integer

SYNOPSIS

```
int irint
    (
    double x  /* argument */
    )
```

DESCRIPTION This routine converts a double-precision value *x* to an integer using the selected IEEE rounding direction.

> **WARNING:** The rounding direction is not pre-selectable and is fixed for round-to-the-nearest.

INCLUDE FILES **math.h**

RETURNS The integer representation of *x*.

SEE ALSO **mathALib**

irintf( )

NAME

irintf( ) – convert a single-precision value to an integer

SYNOPSIS

```
int irintf
    (
    float x   /* argument */
    )
```

DESCRIPTION

This routine converts a single-precision value *x* to an integer using the selected IEEE rounding direction.

WARNING: The rounding direction is not pre-selectable and is fixed as round-to-the-nearest.

INCLUDE FILES

math.h

RETURNS

The integer representation of *x*.

SEE ALSO

mathALib

iround( )

NAME

iround( ) – round a number to the nearest integer

SYNOPSIS

```
int iround
    (
    double x    /* argument */
    )
```

DESCRIPTION

This routine rounds a double-precision value *x* to the nearest integer value.

NOTE: If *x* is spaced evenly between two integers, it returns the even integer.

INCLUDE FILES

math.h

RETURNS

The integer nearest to *x*.

SEE ALSO

mathALib

iroundf( )

NAME **iroundf( )** – round a number to the nearest integer

SYNOPSIS

```
int iroundf
    (
    float x   /* argument */
    )
```

DESCRIPTION This routine rounds a single-precision value *x* to the nearest integer value.

NOTE: If *x* is spaced evenly between two integers, the even integer is returned.

INCLUDE FILES **math.h**

RETURNS The integer nearest to *x*.

SEE ALSO **mathALib**

isalnum( )

NAME **isalnum( )** – test whether a character is alphanumeric (ANSI)

SYNOPSIS

```
int isalnum
    (
    int c                      /* character to test */
    )
```

DESCRIPTION This routine tests whether *c* is a character for which **isalpha( )** or **isdigit( )** returns true.

INCLUDE FILES **ctype.h**

RETURNS Non-zero if and only if *c* is alphanumeric.

SEE ALSO **ansiCtype**

isalpha( )

NAME

isalpha( ) – test whether a character is a letter (ANSI)

SYNOPSIS

```
int isalpha
    (
    int c                    /* character to test */
    )
```

DESCRIPTION

This routine tests whether *c* is a character for which **isupper( )** or **islower( )** returns true.

INCLUDE FILES

ctype.h

RETURNS

Non-zero if and only if *c* is a letter.

SEE ALSO

ansiCtype

isatty( )

NAME

isatty( ) – return whether the underlying driver is a tty device

SYNOPSIS

```
BOOL isatty
    (
    int fd                   /* file descriptor to check */
    )
```

DESCRIPTION

This routine simply invokes the **ioctl( )** function **FIOISATTY** on the specified file descriptor.

RETURNS

TRUE, or **FALSE** if the driver does not indicate a tty device.

SEE ALSO

ioLib

iscntrl( )

NAME **iscntrl( )** – test whether a character is a control character (ANSI)

SYNOPSIS

```
int iscntrl
    (
    int c                     /* character to test */
    )
```

DESCRIPTION This routine tests whether *c* is a control character.

INCLUDE FILES **ctype.h**

RETURNS Non-zero if and only if *c* is a control character.

SEE ALSO **ansiCtype**

isdigit( )

NAME **isdigit( )** – test whether a character is a decimal digit (ANSI)

SYNOPSIS

```
int isdigit
    (
    int c                     /* character to test */
    )
```

DESCRIPTION This routine tests whether *c* is a decimal-digit character.

INCLUDE FILES **ctype.h**

RETURNS Non-zero if and only if *c* is a decimal digit.

SEE ALSO **ansiCtype**

isgraph( )

NAME

isgraph( ) – test whether a character is a printing, non-white-space character (ANSI)

SYNOPSIS

```
int isgraph
    (
    int c                    /* character to test */
    )
```

DESCRIPTION

This routine returns true if *c* is a printing character, and not a character for which **isspace( )** returns true.

INCLUDE FILES

ctype.h

RETURNS

Non-zero if and only if *c* is a printable, non-white-space character.

SEE ALSO

ansiCtype, **isspace( )**

islower( )

NAME

islower( ) – test whether a character is a lower-case letter (ANSI)

SYNOPSIS

```
int islower
    (
    int c                    /* character to test */
    )
```

DESCRIPTION

This routine tests whether *c* is a lower-case letter.

INCLUDE FILES

ctype.h

RETURNS

Non-zero if and only if *c* is a lower-case letter.

SEE ALSO

ansiCtype

isprint( )

NAME **isprint( )** – test whether a character is printable, including the space character (ANSI)

SYNOPSIS

```
int isprint
    (
    int c                      /* character to test */
    )
```

DESCRIPTION This routine returns true if *c* is a printing character or the space character.

INCLUDE FILES **ctype.h**

RETURNS Non-zero if and only if *c* is printable, including the space character.

SEE ALSO **ansiCtype**

ispunct( )

NAME **ispunct( )** – test whether a character is punctuation (ANSI)

SYNOPSIS

```
int ispunct
    (
    int c                      /* character to test */
    )
```

DESCRIPTION This routine tests whether a character is punctuation, *i.e.*, a printing character for which neither **isspace( )** nor **isalnum( )** is true.

INCLUDE FILES **ctype.h**

RETURNS Non-zero if and only if *c* is a punctuation character.

SEE ALSO **ansiCtype**

isspace( )

NAME

isspace( ) – test whether a character is a white-space character (ANSI)

SYNOPSIS

```
int isspace
    (
    int c                     /* character to test */
    )
```

DESCRIPTION

This routine tests whether a character is a standard white-space characters, as follows:

space	' '
horizontal tab	\t
vertical tab	\v
carriage return	\r
new-line	\n
form-feed	\f

INCLUDE FILES

ctype.h

RETURNS

Non-zero if and only if *c* is a space, tab, carriage return, new-line, or form-feed character.

SEE ALSO

ansiCtype

isupper( )

NAME

isupper( ) – test whether a character is an upper-case letter (ANSI)

SYNOPSIS

```
int isupper
    (
    int c                     /* character to test */
    )
```

DESCRIPTION

This routine tests whether *c* is an upper-case letter.

INCLUDE FILES

ctype.h

RETURNS

Non-zero if and only if *c* is an upper-case letter.

SEE ALSO

ansiCtype

isxdigit( )

NAME **isxdigit( )** – test whether a character is a hexadecimal digit (ANSI)

SYNOPSIS

```
int isxdigit
    (
    int c                    /* character to test */
    )
```

DESCRIPTION This routine tests whether *c* is a hexadecimal-digit character.

INCLUDE FILES **ctype.h**

RETURNS Non-zero if and only if *c* is a hexadecimal digit.

SEE ALSO **ansiCtype**

kernelInit( )

NAME

kernelInit( ) – initialize the kernel

SYNOPSIS

```
void kernelInit
    (
    FUNCPTR  rootRtn,          /* user start-up routine */
    unsigned rootMemSize,      /* memory for TCB and root stack */
    char *   pMemPoolStart,    /* beginning of memory pool */
    char *   pMemPoolEnd,      /* end of memory pool */
    unsigned intStackSize,     /* interrupt stack size */
    int      lockOutLevel      /* interrupt lock-out level (1-7) */
    )
```

DESCRIPTION

This routine initializes and starts the kernel. It should be called only once. The parameter *rootRtn* specifies the entry point of the user's start-up code that subsequently initializes system facilities (*i.e.*, the I/O system, network). Typically, *rootRtn* is set to **usrRoot( )**.

Interrupts are enabled for the first time after **kernelInit( )** exits. VxWorks will not exceed the specified interrupt lock-out level during any of its brief uses of interrupt locking as a means of mutual exclusion.

The system memory partition is initialized by **kernelInit( )** with the size set by *pMemPoolStart* and *pMemPoolEnd*. Architectures that support a separate interrupt stack allocate a portion of memory for this purpose, of *intStackSize* bytes starting at *pMemPoolStart*.

NOTE: On SH77xx architectures, the interrupt stack is emulated by software, and it has to be located in a fixed physical address space (P1 or P2) if the on-chip MMU is enabled. If *pMemPoolStart* is in a logical address space (P0 or P3), the interrupt stack area is reserved on the same logical address space. The actual interrupt stack is relocated to a fixed physical space pointed by VBR.

RETURNS

N/A

SEE ALSO

kernelLib, **intLockLevelSet( )**

kernelTimeSlice( )

NAME **kernelTimeSlice( )** – enable round-robin selection

SYNOPSIS

```
STATUS kernelTimeSlice
    (
    int ticks                /* time-slice in ticks or 0 to disable */
                             /* round-robin */
    )
```

DESCRIPTION This routine enables round-robin selection among tasks of same priority and sets the system time-slice to *ticks*. Round-robin scheduling is disabled by default. A time-slice of zero ticks disables round-robin scheduling.

For more information about round-robin scheduling, see the manual entry for **kernelLib**.

RETURNS **OK**, always.

SEE ALSO **kernelLib**

kernelVersion( )

NAME **kernelVersion( )** – return the kernel revision string

SYNOPSIS

```
char *kernelVersion (void)
```

DESCRIPTION This routine returns a string which contains the current revision of the kernel. The string is of the form "WIND version x.y", where "x" corresponds to the kernel major revision, and "y" corresponds to the kernel minor revision.

RETURNS A pointer to a string of format "WIND version x.y".

SEE ALSO **kernelLib**

kill( )

NAME

kill( ) – send a signal to a task (POSIX)

SYNOPSIS

```
int kill
    (
    int tid,                    /* task to send signal to */
    int signo                   /* signal to send to task */
    )
```

DESCRIPTION

This routine sends a signal *signo* to the task specified by *tid*.

RETURNS

OK (0), or **ERROR** (-1) if the task ID or signal number is invalid.

ERRNO

EINVAL

SEE ALSO

sigLib

l( )

NAME

l( ) – disassemble and display a specified number of instructions

SYNOPSIS

```
void l
    (
    INSTR * addr,           /* address of first instruction to */
                            /* disassemble if 0, continue from the last */
                            /* instruction disassembled on the last call */
                            /* to l */
    int     count           /* number of instruction to disassemble if */
                            /* 0, use the same as the last call to l */
    )
```

DESCRIPTION

This routine disassembles a specified number of instructions and displays them on standard output. If the address of an instruction is entered in the system symbol table, the symbol will be displayed as a label for that instruction. Also, addresses in the opcode field of instructions will be displayed symbolically.

To execute, enter:

```
-> l [address [,count]]
```

If *address* is omitted or zero, disassembly continues from the previous address. If *count* is omitted or zero, the last specified count is used (initially 10). As with all values entered via the shell, the address may be typed symbolically.

RETURNS

N/A

SEE ALSO

dbgLib, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

labs( )

NAME

labs( ) – compute the absolute value of a **long** (ANSI)

SYNOPSIS

```
long labs
    (
    long i                  /* long for which to return absolute value */
    )
```

DESCRIPTION

This routine computes the absolute value of a specified **long**. If the result cannot be represented, the behavior is undefined. This routine is equivalent to **abs( )**, except that the argument and return value are all of type **long**.

INCLUDE FILES

stdlib.h

RETURNS

The absolute value of *i*.

SEE ALSO

ansiStdlib

ld( )

NAME

ld( ) – load an object module into memory

SYNOPSIS

```
MODULE_ID ld
    (
    int    syms,           /* -1, 0, or 1 */
    BOOL   noAbort,        /* TRUE = don't abort script on error */
    char * name            /* name of object module, NULL = standard input */
    )
```

DESCRIPTION

This command loads an object module from a file or from standard input. The object module must be in UNIX **a.out** format. External references in the module are resolved during loading. The *syms* parameter determines how symbols are loaded; possible values are:

0 - Add global symbols to the system symbol table.
1 - Add global and local symbols to the system symbol table.
-1 - Add no symbols to the system symbol table.

If there is an error during loading (*e.g.*, externals undefined, too many symbols, etc.), then **shellScriptAbort( )** is called to stop any script that this routine was called from. If *noAbort* is **TRUE**, errors are noted but ignored.

The normal way of using **ld( )** is to load all symbols (*syms* = 1) during debugging and to load only global symbols later.

The routine **ld( )** is a **shell command**. That is, it is designed to be used only in the shell, and not in code running on the target. In future releases, calling **ld( )** directly from code may not be supported.

COMMON SYMBOLS

On the target shell, for the **ld** command only, common symbol behavior is determined by the value of the global variable, **ldCommonMatchAll**. The reasoning for **ldCommonMatchAll** matches the purpose of the **windsh** environment variable, **LD_COMMON_MATCH_ALL** as explained below.

If **ldCommonMatchAll** is set to **TRUE** (equivalent to **windsh** "**LD_COMMON_MATCH_ALL**=on"), the loader tries to match a common symbol with an existing one. If a symbol with the same name is already defined, the loader takes its address. Otherwise, the loader creates a new entry. If set to **FALSE** (equivalent to **windsh** "**LD_COMMON_MATCH_ALL**=off"), the loader does not try to find an existing symbol. It creates an entry for each common symbol.

EXAMPLE

The following example loads the **a.out** file **module** from the default file device into memory, and adds any global symbols to the symbol table:

```
-> ld <module
```

This example loads **test.o** with all symbols:

```
-> ld 1,0,"test.o"
```

RETURNS

MODULE_ID, or **NULL** if there are too many symbols, the object file format is invalid, or there is an error reading the file.

SEE ALSO

usrLib, **loadLib**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

ldexp( )

NAME

ldexp( ) – multiply a number by an integral power of 2 (ANSI)

SYNOPSIS

```
double ldexp
    (
    double v,                /* a floating point number */
    int    xexp              /* exponent */
    )
```

DESCRIPTION

This routine multiplies a floating-point number by an integral power of 2. A range error may occur.

INCLUDE FILES

math.h

RETURNS The double-precision value of *v* times 2 to the power of *xexp*.

SEE ALSO **ansiMath**

ldiv( )

NAME **ldiv( )** – compute the quotient and remainder of the division (ANSI)

SYNOPSIS

```
ldiv_t ldiv
    (
    long numer,                 /* numerator */
    long denom                  /* denominator */
    )
```

DESCRIPTION This routine computes the quotient and remainder of *numer*/*denom*. This routine is similar to **div( )**, except that the arguments and the elements of the returned structure are all of type **long**.

This routine is not reentrant. For a reentrant version, see **ldiv_r( )**.

INCLUDE FILES **stdlib.h**

RETURNS A structure of type **ldiv_t**, containing both the quotient and the remainder.

SEE ALSO **ansiStdlib**

ldiv_r( )

NAME **ldiv_r( )** – compute a quotient and remainder (reentrant)

SYNOPSIS

```
void ldiv_r
    (
    long     numer,             /* numerator */
    long     denom,             /* denominator */
    ldiv_t * divStructPtr       /* ldiv_t structure */
    )
```

DESCRIPTION This routine computes the quotient and remainder of *numer*/*denom*. The quotient and remainder are stored in the **ldiv_t** structure **divStructPtr**.

This routine is the reentrant version of **ldiv( )**.

INCLUDE FILES **stdlib.h**

RETURNS N/A

SEE ALSO **ansiStdlib**

ledClose( )

NAME **ledClose( )** – discard the line-editor ID

SYNOPSIS

```
STATUS ledClose
    (
    int led_id                  /* ID returned by ledOpen */
    )
```

DESCRIPTION This routine frees resources allocated by **ledOpen( )**. The low-level input/output file descriptors are not closed.

RETURNS OK.

SEE ALSO **ledLib**, **ledOpen( )**

ledControl( )

NAME **ledControl( )** – change the line-editor ID parameters

SYNOPSIS

```
void ledControl
    (
    int led_id,                 /* ID returned by ledOpen */
    int inFd,                   /* new input fd (NONE = no change) */
    int outFd,                  /* new output fd (NONE = no change) */
    int histSize                /* new history list size (NONE = no */
                                /* change), (0 = display) */
    )
```

DESCRIPTION This routine changes the input/output file descriptor and the size of the history list.

RETURNS N/A

SEE ALSO **ledLib**

ledOpen( )

NAME **ledOpen( )** – create a new line-editor ID

SYNOPSIS

```
int ledOpen
    (
    int inFd,                   /* low-level device input fd */
    int outFd,                  /* low-level device output fd */
    int histSize                /* size of history list */
    )
```

DESCRIPTION This routine creates the ID that is used by **ledRead( )**, **ledClose( )**, and **ledControl( )**. Storage is allocated for up to *histSize* previously read lines.

RETURNS The line-editor ID, or **ERROR** if the routine runs out of memory.

SEE ALSO **ledLib**, **ledRead( )**, **ledClose( )**, **ledControl( )**

ledRead( )

NAME **ledRead( )** – read a line with line-editing

SYNOPSIS

```
int ledRead
    (
    int    led_id,              /* ID returned by ledOpen */
    char * string,              /* where to return line */
    int    maxBytes             /* maximum number of chars to read */
    )
```

DESCRIPTION This routine handles line-editing and history substitutions. If the low-level input file descriptor is not in **OPT_LINE** mode, only an ordinary **read( )** routine will be performed.

RETURNS The number of characters read, or EOF.

SEE ALSO **ledLib**

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 *      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**

INCLUDE FILES **aio.h**

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

listen( )

NAME

listen( ) – enable connections to a socket

SYNOPSIS

```
STATUS listen
    (
    int s,                    /* socket descriptor */
    int backlog               /* number of connections to queue */
    )
```

DESCRIPTION

This routine enables connections to a socket. It also specifies the maximum number of unaccepted connections that can be pending at one time (*backlog*). After enabling connections with **listen( )**, connections are actually accepted by **accept( )**.

RETURNS

OK, or **ERROR** if the socket is invalid or unable to listen.

SEE ALSO

sockLib

lkAddr( )

NAME

lkAddr( ) – list symbols whose values are near a specified value

SYNOPSIS

```
void lkAddr
    (
    unsigned int addr         /* address around which to look */
    )
```

DESCRIPTION

This command lists the symbols in the system symbol table that are near a specified value. The symbols that are displayed include:

– symbols whose values are immediately less than the specified value
– symbols with the specified value
– succeeding symbols, until at least 12 symbols have been displayed

This command also displays symbols that are local, *i.e.*, symbols found in the system symbol table only because their module was loaded by **ld( )**.

RETURNS

N/A

SEE ALSO

usrLib, **symLib**, **symEach( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

lkup( )

NAME

lkup( ) – list symbols

SYNOPSIS

```
void lkup
    (
    char * substr             /* substring to match */
    )
```

DESCRIPTION

This command lists all symbols in the system symbol table whose names contain the string *substr*. If *substr* is omitted or is 0, a short summary of symbol table statistics is printed. If *substr* is the empty string (""), all symbols in the table are listed.

This command also displays symbols that are local, *i.e.*, symbols found in the system symbol table only because their module was loaded by **ld( )**.

By default, **lkup( )** displays 22 symbols at a time. This can be changed by modifying the global variable **symLkupPgSz**. If this variable is set to 0, **lkup( )** displays all the symbols without interruption.

RETURNS

N/A

SEE ALSO

usrLib, **symLib**, **symEach( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

L

ll( )

NAME

ll( ) – generate a long listing of directory contents

SYNOPSIS

```
STATUS ll
    (
    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, TRUE, FALSE
```

dirName is a name of a directory or file, and may contain wildcards.

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 Tornado Shell (**windsh**), which has a built-in command of the same name that operates on the Host's I/O system.

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**.

SEE ALSO

usrFsLib, **dirList( )**

llr( )

NAME

llr( ) – do a long listing of directory and all its subdirectories contents

SYNOPSIS

```
STATUS llr
    (
    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, TRUE, TRUE
```

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**.

SEE ALSO

usrFsLib, **dirList( )**

loadModule( )

NAME

loadModule( ) – load an object module into memory

SYNOPSIS

```
MODULE_ID loadModule
    (
    int              fd,      /* fd of file to load */
    int              symFlag  /* symbols to add to table (LOAD_[NO
    ALL]_SYMBOLS) */ GLOBAL
    )
```

DESCRIPTION

This routine loads an object module from the specified file, and places the code, data, and BSS into memory allocated from the system memory pool.

This call is equivalent to **loadModuleAt( )** with **NULL** for the addresses of text, data, and BSS segments. For more details, see the manual entry for **loadModuleAt( )**.

RETURNS

MODULE_ID, or **NULL** if the routine cannot read the file, there is not enough memory, or the file format is illegal.

SEE ALSO

loadLib, **loadModuleAt( )**

loadModuleAt( )

NAME

loadModuleAt( ) – load an object module into memory

SYNOPSIS

```
MODULE_ID loadModuleAt
    (
    int    fd,              /* fd from which to read module */
    int    symFlag,         /* symbols to add to table (LOAD_[NO
    char * *ppText,         /* load text segment at addr. pointed to by */
                            /* this ptr, return load addr. via this ptr */
    char * *ppData,         /* load data segment at addr. pointed to by */
                            /* this pointer, return load addr. via this */
                            /* ptr */
    char * *ppBss           /* load BSS segment at addr. pointed to by */
                            /* this pointer, return load addr. via this */
                            /* ptr */
    )
```

DESCRIPTION

This routine reads an object module from *fd*, and loads the code, data, and BSS segments at the specified load addresses in memory set aside by the user using **malloc( )**, or in the system memory partition as described below. The module is properly relocated according to the relocation commands in the file. Unresolved externals will be linked to symbols found in the system symbol table. Symbols in the module being loaded can optionally be added to the system symbol table.

LINKING UNRESOLVED EXTERNALS

As the module is loaded, any unresolved external references are resolved by looking up the missing symbols in the system symbol table. If found, those references are correctly linked to the new module. If unresolved external references cannot be found in the system symbol table, then an error message ("undefined symbol: ...") is printed for the symbol, but the loading/linking continues. The partially resolved module is not removed, to enable the user to examine the module for debugging purposes. Care should be taken when executing code from the resulting module. Executing code which contains references to unresolved symbols may have unexpected results and may corrupt the system's memory.

Even though a module with unresolved symbols remains loaded after this routine returns, **NULL** will be returned to enable the caller to detect the failure programmatically. To unload the module, the caller may either call the unload routine with the module name, or look up the module using the module name and then unload the module using the returned **MODULE_ID**. See the library entries for **moduleLib** and **unldLib** for details. The name of the module is the name of the file loaded with the path removed.

ADDING SYMBOLS TO THE SYMBOL TABLE

The symbols defined in the module to be loaded may be optionally added to the system symbol table, depending on the value of *symFlag*:

LOAD_NO_SYMBOLS
: add no symbols to the system symbol table

LOAD_LOCAL_SYMBOLS
: add only local symbols to the system symbol table

LOAD_GLOBAL_SYMBOLS
: add only external symbols to the system symbol table

LOAD_ALL_SYMBOLS
: add both local and external symbols to the system symbol table

HIDDEN_MODULE
: do not display the module via **moduleShow( )**.

Obsolete symbols:

For backward compatibility with previous releases, the following symbols are also added to the symbol table to indicate the start of each segment: *filename*_text, *filename*_data, and *filename*_bss, where *filename* is the name associated with the fd. Note that these symbols are not available when the ELF format is used. Also they will disappear with the next VxWorks release. The **moduleLib** API should be used instead to get segment information.

RELOCATION

The relocation commands in the object module are used to relocate the text, data, and BSS segments of the module. The location of each segment can be specified explicitly, or left unspecified in which case memory will be allocated for the segment from the system memory partition. This is determined by the parameters *ppText*, *ppData*, and *ppBss*, each of which can have the following values:

NULL
: no load address is specified, none will be returned;

A pointer to **LD_NO_ADDRESS**
: no load address is specified, the return address is referenced by the pointer;

A pointer to an address
: the load address is specified.

The *ppText*, *ppData*, and *ppBss* parameters specify where to load the text, data, and bss sections respectively. Each of these parameters is a pointer to a pointer; for example, ***ppText* gives the address where the text segment is to begin.

For any of the three parameters, there are two ways to request that new memory be allocated, rather than specifying the section's starting address: you can either specify the parameter itself as **NULL**, or you can write the constant **LD_NO_ADDRESS** in place of an address. In the second case, **loadModuleAt( )** routine replaces the **LD_NO_ADDRESS** value with the address actually used for each section (that is, it records the address at **ppText*, **ppData*, or **ppBss*).

The double indirection not only permits reporting the addresses actually used, but also allows you to specify loading a segment at the beginning of memory, since the following cases can be distinguished:

(1) Allocate memory for a section (text in this example): *ppText* == **NULL**
(2) Begin a section at address zero (the text section, below): **ppText* == 0

Note that **loadModule( )** is equivalent to this routine if all three of the segment-address parameters are set to **NULL**.

COMMON

Some host compiler/linker combinations use another storage class internally called "common". In the C language, uninitialized global variables are eventually put in the bss segment. However, in partially linked object modules they are flagged internally as "common" and the static linker (host) resolves these and places them in bss as a final step in creating a fully linked object module. However, the target loader is most often used to load partially linked object modules. When the target loader encounters a variable labeled "common", its behavior depends on the following flags:

LOAD_COMMON_MATCH_NONE
: Allocate memory for the variable with **malloc( )** and enter the variable in the target symbol table (if specified) at that address. This is the default.

L

LOAD_COMMON_MATCH_USER
Search for the symbol in the target symbol table, excluding the vxWorks image symbols. If several symbols exist, then the order of matching is: (1) bss, (2) data. If no symbol is found, act like the default.

LOAD_COMMON_MATCH_ALL
Search for the symbol in the target symbol table, including the vxWorks image symbols. If several symbols exist, then the order of matching is: (1) bss, (2) data. If no symbol is found, act like the default.

Note that most UNIX loaders have an option that forces resolution of the common storage while leaving the module relocatable (for example, with typical BSD UNIX loaders, use options "-rd").

EXAMPLES

Load a module into allocated memory, but do not return segment addresses:

```
module_id = loadModuleAt (fd, LOAD_GLOBAL_SYMBOLS, NULL, NULL, NULL);
```

Load a module into allocated memory, and return segment addresses:

```
pText = pData = pBss = LD_NO_ADDRESS;
module_id = loadModuleAt (fd,LOAD_GLOBAL_SYMBOLS,&pText,&pData,&pBss);
```

Load a module to off-board memory at a specified address:

```
pText = 0x800000;                 /* address of text segment        */
pData = pBss = LD_NO_ADDRESS      /* other segments follow by default */
module_id = loadModuleAt (fd,LOAD_GLOBAL_SYMBOLS,&pText,&pData,&pBss);
```

RETURNS

MODULE_ID, or **NULL** if the file cannot be read, there is not enough memory, the file format is illegal, or there were unresolved symbols.

SEE ALSO

loadLib, *VxWorks Programmer's Guide: Basic OS*

localeconv( )

NAME

localeconv( ) – set the components of an object with type **lconv** (ANSI)

SYNOPSIS

```
struct lconv *localeconv (void)
```

DESCRIPTION

This routine sets the components of an object with type **struct lconv** with values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale.

The members of the structure with type **char *** are pointers to strings any of which (except **decimal_point**) can point to “” to indicate that the value is not available in the current locale or is of zero length. The members with type **char** are nonnegative numbers, any of which can be **CHAR_MAX** to indicate that the value is not available in the current locale. The members include the following:

char *decimal_point
: The decimal-point character used to format non-monetary quantities.

char *thousands_sep
: The character used to separate groups of digits before the decimal-point character in formatted non-monetary quantities.

char *grouping
: A string whose elements indicate the size of each group of digits in formatted non-monetary quantities.

char *int_curr_symbol
: The international currency symbol applicable to the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in ISO 4217:1987. The fourth character (immediately preceding the null character) is the character used to separate the international currency symbol from the monetary quantity.

char *currency_symbol
: The local currency symbol applicable to the current locale.

char *mon_decimal_point
: The decimal-point used to format monetary quantities.

char *mon_thousands_sep
: The separator for groups of digits before the decimal-point in formatted monetary quantities.

char *mon_grouping
: A string whose elements indicate the size of each group of digits in formatted monetary quantities.

char *positive_sign
: The string used to indicate a nonnegative-valued formatted monetary quantity.

char *negative_sign
: The string used to indicate a negative-valued formatted monetary quantity.

char int_frac_digits
: The number of fractional digits (those after the decimal-point) to be displayed in an internationally formatted monetary quantity.

char frac_digits
: The number of fractional digits (those after the decimal-point) to be displayed in a formatted monetary quantity.

char p_cs_precedes
Set to 1 or 0 if the **currency_symbol** respectively precedes or succeeds the value for a nonnegative formatted monetary quantity.

char p_sep_by_space
Set to 1 or 0 if the **currency_symbol** respectively is or is not separated by a space from the value for a nonnegative formatted monetary quantity.

char n_cs_precedes
Set to 1 or 0 if the **currency_symbol** respectively precedes or succeeds the value for a negative formatted monetary quantity.

char n_sep_by_space
Set to 1 or 0 if the **currency_symbol** respectively is or is not separated by a space from the value for a negative formatted monetary quantity.

char p_sign_posn
Set to a value indicating the positioning of the **positive_sign** for a nonnegative formatted monetary quantity.

char n_sign_posn
Set to a value indicating the positioning of the **negative_sign** for a negative formatted monetary quantity.

The elements of **grouping** and **mon_grouping** are interpreted according to the following:

CHAR_MAX
No further grouping is to be performed.

0
The previous element is to be repeatedly used for the remainder of the digits.

other
The integer value is the number of the digits that comprise the current group. The next element is examined to determined the size of the next group of digits before the current group.

The values of **p_sign_posn** and **n_sign_posn** are interpreted according to the following:

0
Parentheses surround the quantity and **currency_symbol**.

1
The sign string precedes the quantity and **currency_symbol**.

2
The sign string succeeds the quantity and **currency_symbol**.

3
The sign string immediately precedes the **currency_symbol**.

4
The sign string immediately succeeds the **currency_symbol**.

The implementation behaves as if no library function calls **localeconv( )**.

The **localeconv( )** routine returns a pointer to the filled-in object. The structure pointed to by the return value is not modified by the program, but may be overwritten by a subsequent call to **localeconv( )**. In addition, calls to **setlocale( )** with categories **LC_ALL**, **LC_MONETARY**, or **LC_NUMERIC** may overwrite the contents of the structure.

INCLUDE FILES **locale.h**, **limits.h**

RETURNS A pointer to the structure **lconv**.

SEE ALSO **ansiLocale**

localtime( )

NAME **localtime( )** – convert calendar time into broken-down time (ANSI)

SYNOPSIS

```
struct tm *localtime
    (
    const time_t * timer      /* calendar time in seconds */
    )
```

DESCRIPTION This routine converts the calendar time pointed to by *timer* into broken-down time, expressed as local time.

This routine is not reentrant. For a reentrant version, see **localtime_r( )**.

INCLUDE FILES **time.h**

RETURNS A pointer to a **tm** structure containing the local broken-down time.

SEE ALSO **ansiTime**

L

localtime_r( )

NAME

localtime_r( ) – convert calendar time into broken-down time (POSIX)

SYNOPSIS

```
int localtime_r
    (
    const time_t * timer,      /* calendar time in seconds */
    struct tm *    timeBuffer /* buffer for the broken-down time */
    )
```

DESCRIPTION

This routine converts the calendar time pointed to by *timer* into broken-down time, expressed as local time. The broken-down time is stored in *timeBuffer*.

This routine is the POSIX re-entrant version of **localtime( )**.

INCLUDE FILES

time.h

RETURNS

OK.

SEE ALSO

ansiTime

log( )

NAME

log( ) – compute a natural logarithm (ANSI)

SYNOPSIS

```
double log
    (
    double x                   /* value to compute the natural logarithm of */
    )
```

DESCRIPTION

This routine returns the natural logarithm of x in double precision (IEEE double, 53 bits).

A domain error occurs if the argument is negative. A range error may occur if the argument is zero.

INCLUDE FILES

math.h

RETURNS

The double-precision natural logarithm of x.

Special cases:
  If $x < 0$ (including -INF), it returns NaN with signal.
  If x is +INF, it returns x with no signal.

If *x* is 0, it returns -INF with signal.
If *x* is NaN it returns *x* with no signal.

SEE ALSO **ansiMath**, **mathALib**

log2( )

NAME **log2( )** – compute a base-2 logarithm

SYNOPSIS

```
double log2
    (
    double x  /* value to compute the base-two logarithm of */
    )
```

DESCRIPTION This routine returns the base-2 logarithm of *x* in double precision.

INCLUDE FILES **math.h**

RETURNS The double-precision base-2 logarithm of *x*.

SEE ALSO **mathALib**

log2f( )

NAME **log2f( )** – compute a base-2 logarithm

SYNOPSIS

```
float log2f
    (
    float x    /* value to compute the base-2 logarithm of */
    )
```

DESCRIPTION This routine returns the base-2 logarithm of *x* in single precision.

INCLUDE FILES **math.h**

RETURNS The single-precision base-2 logarithm of *x*.

SEE ALSO **mathALib**

log10( )

NAME

log10( ) – compute a base-10 logarithm (ANSI)

SYNOPSIS

```
double log10
    (
    double x                  /* value to compute the base-10 logarithm of */
    )
```

DESCRIPTION

This routine returns the base 10 logarithm of *x* in double precision (IEEE double, 53 bits).

A domain error occurs if the argument is negative. A range error may if the argument is zero.

INCLUDE FILES

math.h

RETURNS

The double-precision base-10 logarithm of *x*.

Special cases:
  If $x < 0$, **log10( )** returns NaN with signal.
  if *x* is +INF, it returns *x* with no signal.
  if *x* is 0, it returns -INF with signal.
  if *x* is NaN it returns *x* with no signal.

SEE ALSO

ansiMath, **mathALib**

log10f( )

NAME

log10f( ) – compute a base-10 logarithm (ANSI)

SYNOPSIS

```
float log10f
    (
    float x    /* value to compute the base-10 logarithm of */
    )
```

DESCRIPTION

This routine returns the base-10 logarithm of *x* in single precision.

INCLUDE FILES

math.h

RETURNS

The single-precision base-10 logarithm of *x*.

SEE ALSO

mathALib

logf( )

NAME **logf( )** – compute a natural logarithm (ANSI)

SYNOPSIS

```
float logf
    (
    float x   /* value to compute the natural logarithm of */
    )
```

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

INCLUDE FILES **math.h**

RETURNS The single-precision natural logarithm of *x*.

SEE ALSO **mathALib**

logFdAdd( )

NAME **logFdAdd( )** – add a logging file descriptor

SYNOPSIS

```
STATUS logFdAdd
    (
    int fd                    /* file descriptor for additional logging */
                              /* device */
    )
```

DESCRIPTION This routine adds to the log file descriptor list another file descriptor *fd* to which messages will be logged. The file descriptor must be a valid open file descriptor.

RETURNS **OK**, or **ERROR** if the allowable number of additional logging file descriptors (5) is exceeded.

SEE ALSO **logLib**, **logFdDelete( )**

logFdDelete( )

NAME

logFdDelete( ) – delete a logging file descriptor

SYNOPSIS

```
STATUS logFdDelete
    (
    int fd                      /* file descriptor to stop using as logging */
                                /* device */
    )
```

DESCRIPTION

This routine removes from the log file descriptor list a logging file descriptor added by **logFdAdd( )**. The file descriptor is not closed; but is no longer used by the logging facilities.

RETURNS

OK, or **ERROR** if the file descriptor was not added with **logFdAdd( )**.

SEE ALSO

logLib, **logFdAdd( )**

logFdSet( )

NAME

logFdSet( ) – set the primary logging file descriptor

SYNOPSIS

```
void logFdSet
    (
    int fd                      /* file descriptor to use as logging device */
    )
```

DESCRIPTION

This routine changes the file descriptor where messages from **logMsg( )** are written, allowing the log device to be changed from the default specified by **logInit( )**. It first removes the old file descriptor (if one had been previously set) from the log file descriptor list, then adds the new *fd*.

The old logging file descriptor is not closed or affected by this call; it is simply no longer used by the logging facilities.

RETURNS

N/A

SEE ALSO

logLib, **logFdAdd( )**, **logFdDelete( )**

loginDefaultEncrypt( )

NAME

loginDefaultEncrypt( ) – default password encryption routine

SYNOPSIS

```
STATUS loginDefaultEncrypt
    (
    char * in,                /* input string */
    char * out                /* encrypted string */
    )
```

DESCRIPTION

This routine provides default encryption for login passwords. It employs a simple encryption algorithm. It takes as arguments a string *in* and a pointer to a buffer *out*. The encrypted string is then stored in the buffer.

The input strings must be at least 8 characters and no more than 40 characters.

If a more sophisticated encryption algorithm is needed, this routine can be replaced, as long as the new encryption routine retains the same declarations as the default routine. The utility **vxencrypt** in **host/*hostOs*/bin** should also be replaced by a host version of *encryptionRoutine*. For more information, see the manual entry for **loginEncryptInstall( )**.

RETURNS

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

SEE ALSO

loginLib, **loginEncryptInstall( )**, **vxencrypt**

loginEncryptInstall( )

NAME

loginEncryptInstall( ) – install an encryption routine

SYNOPSIS

```
void loginEncryptInstall
    (
    FUNCPTR rtn,            /* function pointer to encryption routine */
    int     var             /* argument to the encryption routine (unused) */
    )
```

DESCRIPTION

This routine allows the user to install a custom encryption routine. The custom routine *rtn* must be of the following form:

```
STATUS encryptRoutine
       (
       char *password,              /* string to encrypt    */
```

L

```
    char *encryptedPassword       /* resulting encryption */
    )
```

When a custom encryption routine is installed, a host version of this routine must be written to replace the tool **vxencrypt( )** in **host/*hostOs*/bin**.

EXAMPLE The custom example above could be installed as follows:

```
#ifdef INCLUDE_SECURITY
    loginInit ();                              /* initialize login table   */
    shellLoginInstall (loginPrompt, NULL);     /* install shell security   */
    loginEncryptInstall (encryptRoutine, NULL); /* install encrypt routine */
#endif
```

RETURNS N/A

SEE ALSO **loginLib**, **loginDefaultEncrypt( )**, **vxencrypt**

loginInit( )

NAME **loginInit( )** – initialize the login table

SYNOPSIS

```
void loginInit (void)
```

DESCRIPTION This routine must be called to initialize the login data structure used by routines throughout this module. If the configuration macro **INCLUDE_SECURITY** is defined, it is called by **usrRoot( )** in **usrConfig.c**, before any other routines in this module.

RETURNS N/A

SEE ALSO **loginLib**

logInit( )

NAME
logInit( ) – initialize message logging library

SYNOPSIS
```
STATUS logInit
    (
    int fd,             /* file descriptor to use as logging device */
    int maxMsgs         /* max. number of messages allowed in log queue */
    )
```

DESCRIPTION
This routine specifies the file descriptor to be used as the logging device and the number of messages that can be in the logging queue. If more than *maxMsgs* are in the queue, they will be discarded. A message is printed to indicate lost messages.

This routine spawns **logTask( )**, the task-level portion of error logging.

This routine must be called before any other routine in **logLib**. This is done by the root task, **usrRoot( )**, in **usrConfig.c**.

RETURNS
OK, or **ERROR** if a message queue could not be created or **logTask( )** could not be spawned.

SEE ALSO
logLib

loginPrompt( )

NAME
loginPrompt( ) – display a login prompt and validate a user entry

SYNOPSIS
```
STATUS loginPrompt
    (
    char * userName         /* user name, ask if NULL or not provided */
    )
```

DESCRIPTION
This routine displays a login prompt and validates a user entry. If both user name and password match with an entry in the login table, the user is then given access to the VxWorks system. Otherwise, it prompts the user again.

All control characters are disabled during authentication except CTRL-D, which will terminate the remote login session.

RETURNS **OK** if the name and password are valid, or **ERROR** if there is an EOF or the routine times out.

SEE ALSO **loginLib**

loginStringSet( )

NAME **loginStringSet( )** – change the login string

SYNOPSIS

```
void loginStringSet
    (
    char * newString          /* string to become new login prompt */
    )
```

DESCRIPTION This routine changes the login prompt string to *newString*. The maximum string length is 80 characters.

RETURNS N/A

SEE ALSO **loginLib**

loginUserAdd( )

NAME **loginUserAdd( )** – add a user to the login table

SYNOPSIS

```
STATUS loginUserAdd
    (
    char name[MAX_LOGIN_NAME_LEN+1], /* user name */
    char passwd[80]                  /* user password */
    )
```

DESCRIPTION This routine adds a user name and password entry to the login table. Note that what is saved in the login table is the user name and the address of *passwd*, not the actual password.

The length of user names should not exceed **MAX_LOGIN_NAME_LEN**, while the length of passwords depends on the encryption routine used. For the default encryption routine, passwords should be at least 8 characters long and no more than 40 characters.

The procedure for adding a new user to login table is as follows:

(1) Generate the encrypted password by invoking **vxencrypt** in **host/*hostOs*/bin**.

(2) Add a user by invoking **loginUserAdd( )** in the VxWorks shell with the user name and the encrypted password.

The password of a user can be changed by first deleting the user entry, then adding the user entry again with the new encrypted password.

EXAMPLE

```
-> loginUserAdd "peter", "RRdRd9Qbyz"
value = 0 = 0x0
-> loginUserAdd "robin", "bSzyydqbSb"
value = 0 = 0x0
-> loginUserShow
  User Name
  =========
  peter
  robin
value = 0 = 0x0
->
```

RETURNS **OK**, or **ERROR** if the user name has already been entered.

SEE ALSO **loginLib**, **vxencrypt**

loginUserDelete( )

NAME **loginUserDelete( )** – delete a user entry from the login table

SYNOPSIS

```
STATUS loginUserDelete
    (
    char * name,              /* user name */
    char * passwd             /* user password */
    )
```

DESCRIPTION This routine deletes an entry in the login table. Both the user name and password must be specified to remove an entry from the login table.

RETURNS **OK**, or **ERROR** if the specified user or password is incorrect.

SEE ALSO **loginLib**

L

loginUserShow( )

NAME
loginUserShow( ) – display the user login table

SYNOPSIS

```
void loginUserShow (void)
```

DESCRIPTION
This routine displays valid user names.

EXAMPLE

```
-> loginUserShow ()
  User Name
  =========
  peter
  robin
value = 0 = 0x0
```

RETURNS
N/A

SEE ALSO
loginLib

loginUserVerify( )

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

SYNOPSIS

```
STATUS loginUserVerify
    (
    char * name,              /* name of user */
    char * passwd             /* password of user */
    )
```

DESCRIPTION
This routine verifies a user entry in the login table.

RETURNS
OK, or **ERROR** if the user name or password is not found.

SEE ALSO
loginLib

logMsg( )

NAME

logMsg( ) – log a formatted error message

SYNOPSIS

```
int logMsg
    (
    char * fmt,             /* format string for print */
    int    arg1,            /* first of six required args for fmt */
    int    arg2,
    int    arg3,
    int    arg4,
    int    arg5,
    int    arg6
    )
```

DESCRIPTION

This routine logs a specified message via the logging task. This routine's syntax is similar to **printf( )** -- a format string is followed by arguments to format. However, **logMsg( )** takes a **char** * rather than a **const char** * and requires a fixed number of arguments (6).

The task ID of the caller is prepended to the specified message.

SPECIAL CONSIDERATIONS

Because **logMsg( )** does not actually perform the output directly to the logging streams, but instead queues the message to the logging task, **logMsg( )** can be called from interrupt service routines.

However, since the arguments are interpreted by the **logTask( )** at the time of actual logging, instead of at the moment when **logMsg( )** is called, arguments to **logMsg( )** should not be pointers to volatile entities (*e.g.*, dynamic strings on the caller stack).

logMsg( ) checks to see whether or not it is running in interrupt context. If it is, it will not block. However, if invoked from a task, it can cause the task to block.

For more detailed information about the use of **logMsg( )**, see the manual entry for **logLib**.

EXAMPLE

If the following code were executed by task 20:

```
{
name = "GRONK";
num = 123;
logMsg ("ERROR - name = %s, num = %d.\n", name, num, 0, 0, 0, 0);
}
```

the following error message would appear on the system log:

```
0x180400 (t20): ERROR - name = GRONK, num = 123.
```

L

RETURNS The number of bytes written to the log queue, or EOF if the routine is unable to write a message.

SEE ALSO **logLib**, **printf( )**, **logTask( )**

logout( )

NAME **logout( )** – log out of the VxWorks system

SYNOPSIS

```
void logout (void)
```

DESCRIPTION This command logs out of the VxWorks shell. If a remote login is active (via **rlogin** or **telnet**), it is stopped, and standard I/O is restored to the console.

SEE ALSO **usrLib**, **rlogin( )**, **telnet( )**, **shellLogout( )**, *VxWorks Programmer's Guide: Target Shell*

logTask( )

NAME **logTask( )** – message-logging support task

SYNOPSIS

```
void logTask (void)
```

DESCRIPTION This routine prints the messages logged with **logMsg( )**. It waits on a message queue and prints the messages as they arrive on the file descriptor specified by **logInit( )** (or a subsequent call to **logFdSet( )** or **logFdAdd( )**).

This task is spawned by **logInit( )**.

RETURNS N/A

SEE ALSO **logLib**, **logMsg( )**

longjmp( )

NAME

longjmp( ) – perform non-local goto by restoring saved environment (ANSI)

SYNOPSIS

```
void longjmp
    (
    jmp_buf env,
    int     val
    )
```

DESCRIPTION

This routine restores the environment saved by the most recent invocation of **setjmp( )** 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.

If there was no corresponding **setjmp( )** call, or if the call containing the corresponding **setjmp( )** has already returned, the behavior of **longjmp( )** is unpredictable.

All accessible objects in memory retain their values as of the time **longjmp( )** was called, with one exception: local objects on the C stack that are not declared **volatile**, and have been changed between the **setjmp( )** invocation and the **longjmp( )** call, have unpredictable values.

The **longjmp( )** function executes correctly in contexts of signal handlers and any of their associated functions (but not from interrupt handlers).

WARNING: Do not use **longjmp( )** or **setjmp( )** from an ISR.

RETURNS

This routine does not return to its caller. Instead, it causes **setjmp( )** to return *val*, unless *val* is 0; in that case **setjmp( )** returns 1.

SEE ALSO

ansiSetjmp, **setjmp( )**

ls( )

NAME

ls( ) – generate a brief listing of a directory

SYNOPSIS

```
STATUS ls
    (
    char * dirName,          /* name of dir to list */
    BOOL   doLong            /* switch on details */
    )
```


L

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 Tornado 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**.

SEE ALSO

usrFsLib, **dirList( )**

lseek( )

NAME

lseek( ) – set a file read/write pointer

SYNOPSIS

```
int lseek
    (
    int  fd,                  /* file descriptor */
    long 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**.

SEE ALSO

ioLib

lsr( )

NAME **lsr( )** – list the contents of a directory and any of its subdirectories

SYNOPSIS

```
STATUS lsr
    (
    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**.

SEE ALSO **usrFsLib**, **dirList( )**

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

SEE ALSO **lstLib**

lstConcat( )

NAME

lstConcat( ) – concatenate two lists

SYNOPSIS

```
void lstConcat
    (
    LIST * pDstList,           /* destination list */
    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

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.

SEE ALSO

lstLib

lstDelete( )

NAME **lstDelete( )** – delete a specified node from a list

SYNOPSIS

```
void lstDelete
    (
    LIST * pList,              /* pointer to list descriptor */
    NODE * pNode               /* pointer to node to be deleted */
    )
```

DESCRIPTION This routine deletes a specified node from a specified list.

RETURNS N/A

SEE ALSO **lstLib**

L

lstExtract( )

NAME **lstExtract( )** – extract a sublist from a list

SYNOPSIS

```
void lstExtract
    (
    LIST * pSrcList,           /* pointer to source list */
    NODE * pStartNode,         /* first node in sublist to be extracted */
    NODE * pEndNode,           /* last node in sublist to be extracted */
    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

SEE ALSO **lstLib**

lstFind( )

NAME **lstFind( )** – find a node in a list

SYNOPSIS

```
int lstFind
    (
    LIST * pList,             /* list in which to search */
    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.

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.

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

SEE ALSO **lstLib**, **free( )**

L

lstGet( )

NAME **lstGet( )** – delete and return the first node from a list

SYNOPSIS

```
NODE *lstGet
    (
    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.

SEE ALSO **lstLib**

lstInit( )

NAME **lstInit( )** – initialize a list descriptor

SYNOPSIS

```
void lstInit
    (
    LIST * pList              /* ptr to list descriptor to be initialized */
    )
```

DESCRIPTION This routine initializes a specified list to an empty list.

RETURNS N/A

SEE ALSO **lstLib**

lstInsert( )

NAME **lstInsert( )** – insert a node in a list after a specified node

SYNOPSIS

```
void lstInsert
    (
    LIST * pList,             /* pointer to list descriptor */
    NODE * pPrev,             /* pointer to node after which to insert */
    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

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.

SEE ALSO **lstLib**

lstLibInit( )

NAME **lstLibInit( )** – initializes **lstLib** module

SYNOPSIS

```
void lstLibInit (void)
```

DESCRIPTION This routine pulls **lstLib** into the vxWorks image.

RETURNS N/A

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.

SEE ALSO

lstLib

lstNStep( )

NAME

lstNStep( ) – find a list node *nStep* steps away from a specified node

SYNOPSIS

```
NODE *lstNStep
    (
    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.

SEE ALSO

lstLib

lstNth( )

NAME **lstNth( )** – find the Nth node in a list

SYNOPSIS

```
NODE *lstNth
    (
    LIST * pList,            /* pointer to list descriptor */
    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.

SEE ALSO **lstLib**

L

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.

SEE ALSO **lstLib**

m( )

NAME

m( ) – modify memory

SYNOPSIS

```
void m
    (
    void * adrs,              /* address to change */
    int    width              /* width of unit to be modified (1, 2, 4, 8) */
    )
```

DESCRIPTION

This command prompts the user for modifications to memory in byte, short word, or long word specified by *width*, starting at the specified address. It prints each address and the current contents of that address, in turn. If *adrs* or *width* is zero or absent, it defaults to the previous value. The user can respond in one of several ways:

RETURN
: Do not change this address, but continue, prompting at the next address.

number
: Set the content of this address to *number*.

. (dot)
: Do not change this address, and quit.

EOF
: Do not change this address, and quit.

All numbers entered and displayed are in hexadecimal.

RETURNS

N/A

SEE ALSO

usrLib, **mRegs( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

m2Delete( )

NAME

m2Delete( ) – delete all the MIB-II library groups

SYNOPSIS

```
STATUS m2Delete (void)
```

DESCRIPTION

This routine cleans up the state associated with the MIB-II library.

RETURNS

OK (always).

SEE ALSO

m2Lib, **m2SysDelete( )**, **m2TcpDelete( )**, **m2UdpDelete( )**, **m2IcmpDelete( )**, **m2IfDelete( )**, **m2IpDelete( )**

m2IcmpDelete( )

NAME **m2IcmpDelete( )** – delete all resources used to access the ICMP group

SYNOPSIS

```
STATUS m2IcmpDelete (void)
```

DESCRIPTION This routine frees all the resources allocated at the time the ICMP group was initialized. The ICMP group should not be accessed after this routine has been called.

RETURNS **OK**, always.

SEE ALSO **m2IcmpLib**, **m2IcmpInit( )**, **m2IcmpGroupInfoGet( )**

m2IcmpGroupInfoGet( )

NAME **m2IcmpGroupInfoGet( )** – get the MIB-II ICMP-group global variables

SYNOPSIS

```
STATUS m2IcmpGroupInfoGet
    (
    M2_ICMP * pIcmpInfo       /* pointer to the ICMP group structure */
    )
```

DESCRIPTION This routine fills in the ICMP structure at *pIcmpInfo* with the MIB-II ICMP scalar variables.

RETURNS **OK**, or **ERROR** if the input parameter *pIcmpInfo* is invalid.

ERRNO **S_m2Lib_INVALID_PARAMETER**

SEE ALSO **m2IcmpLib**, **m2IcmpInit( )**, **m2IcmpDelete( )**

m2IcmpInit( )

NAME **m2IcmpInit( )** – initialize MIB-II ICMP-group access

SYNOPSIS

```
STATUS m2IcmpInit (void)
```

DESCRIPTION This routine allocates the resources needed to allow access to the MIB-II ICMP-group variables. This routine must be called before any ICMP variables can be accessed.

RETURNS **OK**, always.

SEE ALSO **m2IcmpLib**, **m2IcmpGroupInfoGet( )**, **m2IcmpDelete( )**

m2If8023PacketCount( )

NAME **m2If8023PacketCount( )** – increment the packet counters for an 802.3 device

SYNOPSIS

```
STATUS m2If8023PacketCount
    (
    M2_ID * pId,              /* The pointer to the device M2_ID object */
    UINT    ctrl,             /* Update In or Out counters */
    UCHAR * pPkt,             /* The incoming/outgoing packet */
    ULONG   pktLen            /* Length of the packet */
    )
```

DESCRIPTION This function is used to update basic interface counters for a packet. The *ctrl* argument specifies whether the packet is being sent or just received (**M2_PACKET_IN** or **M2_PACKET_OUT**). This function only works for 802.3 devices as it understand the Ethernet packet format. The following counters are updated:

- ifInOctets
- ifInUcastPkts
- ifInNUcastPkts
- ifOutOctets
- ifOutUcastPkts
- ifOutNUcastPkts
- ifInMulticastPkts
- ifInBroadcastPkts

- ifOutMulticastPkts
- ifOutBroadcastPkts
- ifHCInOctets
- ifHCInUcastPkts
- ifHCOutOctets
- ifHCOutUcastPkts
- ifHCInMulticastPkts
- ifHCInBroadcastPkts
- ifHCOutMulticastPkts
- ifHCOutBroadcastPkts
- ifCounterDiscontinuityTime

This function should be called right after the **netMblkToBufCopy( )** function has been completed. The first 6 bytes in the resulting buffer must contain the destination MAC address and the second 6 bytes of the buffer must contain the source MAC address.

The type of MAC address (*i.e.*, broadcast, multicast, or unicast) is determined by the following:

broadcast address: ff:ff:ff:ff:ff:ff
multicast address: first bit is set
unicast address: any other address not matching the above

RETURNS

ERROR, if the **M2_ID** is **NULL**, or the ctrl is invalid; **OK**, if the counters were updated.

SEE ALSO

m2IfLib

m2IfAlloc( )

NAME

m2IfAlloc( ) – allocate the structure for the interface table

SYNOPSIS

```
M2_ID * m2IfAlloc
    (
    ULONG   ifType,           /* If type of the interface */
    UCHAR * pEnetAddr,        /* Physical address of interface */
    ULONG   addrLen,          /* Address length */
    ULONG   mtuSize,          /* MTU of interface */
    ULONG   speed,            /* Speed of the interface */
    char *  pName,            /* Name of the device */
    int     unit              /* Unit number of the device */
    )
```

DESCRIPTION
This routine is called by the driver during initialization of the interface. The memory for the interface table is allocated here. We also set the default update routines in the **M2_ID** struct. These fields can later be overloaded using the installed routines in the **M2_ID**. Once this function returns, it is the driver's responsibility to set the pMib2Tbl pointer in the END object to the new **M2_ID**.

When this call returns, the calling routine must set the **END_MIB_2233** bit of the flags field in the END object.

RETURNS
Pointer to the **M2_ID** structure that was allocated.

SEE ALSO
m2IfLib

m2IfCommonValsGet( )

NAME
m2IfCommonValsGet( ) – get the common values

SYNOPSIS
```
void m2IfCommonValsGet
    (
    M2_DATA *    pM2Data,      /* The requested struct */
    M2_IFINDEX * pIfIndexEntry /* The ifindex node */
    )
```

DESCRIPTION
This function updates the requested struct with all the data that is independent of the driver ioctl. This information can be obtained from the ifnet structures.

RETURNS
n/a

SEE ALSO
m2IfLib

m2IfCounterUpdate( )

NAME **m2IfCounterUpdate( )** – increment interface counters

SYNOPSIS

```
STATUS m2IfCounterUpdate
    (
    M2_ID * pId,              /* The pointer to the device M2_ID object */
    UINT    ctrId,            /* Counter to update */
    ULONG   value             /* Amount to update the counter by */
    )
```

DESCRIPTION This function is used to directly update an interface counter. The counter is specified by *ctrId* and the amount to increment it is specified by value. If the counter would roll over then the **ifCounterDiscontinuityTime** is updated with the current system uptime.

RETURNS **ERROR** if the **M2_ID** is **NULL**, **OK** if the counter was updated.

SEE ALSO **m2IfLib**

m2IfCtrUpdateRtnInstall( )

NAME **m2IfCtrUpdateRtnInstall( )** – install an interface counter update routine

SYNOPSIS

```
STATUS m2IfCtrUpdateRtnInstall
    (
    M2_ID *           pId,
    M2_CTR_UPDATE_RTN pRtn
    )
```

DESCRIPTION This function installs a routine in the **M2_ID**. This routine is able to update a single specified interface counter.

RETURNS **ERROR** if the **M2_ID** is **NULL**, **OK** if the routine was installed.

SEE ALSO **m2IfLib**

m2IfDefaultValsGet( )

NAME **m2IfDefaultValsGet( )** – get the default values for the counters

SYNOPSIS

```
void m2IfDefaultValsGet
    (
    M2_DATA *    pM2Data,       /* The requested entry */
    M2_IFINDEX * pIfIndexEntry /* The ifindex node */
    )
```

DESCRIPTION This function fills the given struct with the default values as specified in the RFC. We will enter this routine only if the ioctl to the driver fails.

RETURNS n/a

SEE ALSO **m2IfLib**

m2IfDelete( )

NAME **m2IfDelete( )** – delete all resources used to access the interface group

SYNOPSIS

```
STATUS m2IfDelete (void)
```

DESCRIPTION This routine frees all the resources allocated at the time the group was initialized. The interface group should not be accessed after this routine has been called.

RETURNS **OK**, always.

SEE ALSO **m2IfLib**, **m2IfInit( )**, **m2IfGroupInfoGet( )**, **m2IfTblEntryGet( )**, **m2IfTblEntrySet( )**

m2IfFree( )

NAME **m2IfFree( )** – free an interface data structure

SYNOPSIS

```
STATUS m2IfFree
    (
    M2_ID * pId                /* pointer to the driver's M2_ID object */
    )
```

DESCRIPTION This routine frees the given **M2_ID**. Note if the driver is not an RFC 2233 driver then the **M2_ID** is **NULL** and this function simply returns.

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

SEE ALSO **m2IfLib**

m2IfGenericPacketCount( )

NAME **m2IfGenericPacketCount( )** – increment the interface packet counters

SYNOPSIS

```
STATUS m2IfGenericPacketCount
    (
    M2_ID * pId,               /* The pointer to the device M2_ID object */
    UINT    ctrl,              /* Update In or Out counters */
    UCHAR * pPkt,              /* The incoming/outgoing packet */
    ULONG   pktLen             /* Length of the packet */
    )
```

DESCRIPTION This function updates the basic interface counters for a packet. It knows nothing of the underlying media. Thus, so only the **ifInOctets**, **ifHCInOctets**, **ifOutOctets**, **ifHCOutOctets**, and **ifCounterDiscontinuityTime** variables are incremented. The *ctrl* argument specifies whether the packet is being sent or just received (**M2_PACKET_IN** or **M2_PACKET_OUT**).

RETURNS **ERROR** if the **M2_ID** is **NULL**, **OK** if the counters were updated.

SEE ALSO **m2IfLib**

m2IfGroupInfoGet( )

NAME

m2IfGroupInfoGet( ) – get the MIB-II interface-group scalar variables

SYNOPSIS

```
STATUS m2IfGroupInfoGet
    (
    M2_INTERFACE * pIfInfo    /* pointer to interface group structure */
    )
```

DESCRIPTION

This routine fills the interface-group structure at *pIfInfo* with the values of MIB-II interface-group global variables.

RETURNS

OK, or **ERROR** if *pIfInfo* is not a valid pointer.

ERRNO

S_m2Lib_INVALID_PARAMETER

SEE ALSO

m2IfLib, **m2IfInit( )**, **m2IfTblEntryGet( )**, **m2IfTblEntrySet( )**, **m2IfDelete( )**

m2IfInit( )

NAME

m2IfInit( ) – initialize MIB-II interface-group routines

SYNOPSIS

```
STATUS m2IfInit
    (
    FUNCPTR pTrapRtn,         /* pointer to user trap generator */
    void *  pTrapArg          /* pointer to user trap generator argument */
    )
```

DESCRIPTION

This routine allocates the resources needed to allow access to the MIB-II interface-group variables. This routine must be called before any interface variables can be accessed. The input parameter *pTrapRtn* is an optional pointer to a user-supplied SNMP trap generator. The input parameter *pTrapArg* is an optional argument to the trap generator. Only one trap generator is supported.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call **m2IfInit( )** from within the kernel protection domain only, and the data referenced in the *pTrapRtn* and *pTrapArg* parameters must reside in the kernel protection domain. This restriction does not apply to non-AE versions of VxWorks.

RETURNS **OK**, if successful; **ERROR**, if an error occurred.

ERRNO **S_m2Lib_CANT_CREATE_IF_SEM**

SEE ALSO **m2IfLib**, **m2IfGroupInfoGet( )**, **m2IfTblEntryGet( )**, **m2IfTblEntrySet( )**, **m2IfDelete( )**

m2IfPktCountRtnInstall( )

NAME **m2IfPktCountRtnInstall( )** – install an interface packet counter routine

SYNOPSIS

```
STATUS m2IfPktCountRtnInstall
    (
    M2_ID *          pId,
    M2_PKT_COUNT_RTN pRtn
    )
```

DESCRIPTION This function installs a routine in the **M2_ID**. This routine is a packet counter which is able to update all the interface counters.

RETURNS **ERROR** if the **M2_ID** is **NULL**, **OK** if the routine was installed.

SEE ALSO **m2IfLib**

m2IfRcvAddrEntryGet( )

NAME **m2IfRcvAddrEntryGet( )** – get the rcvAddress table entries for a given address

SYNOPSIS

```
STATUS m2IfRcvAddrEntryGet
    (
    int                search,      /* exact search or next search */
    int *              pIndex,      /* pointer to the ifIndex */
    M2_IFRCVADDRTBL * pIfReqEntry /* struct for the values */
    )
```

DESCRIPTION This function returns the exact or the next value in the ifRcvAddressTable based on the value of the search parameter. In order to identify the appropriate entry, this function needs two identifiers - the ifIndex of the interface and the physical address for which the status or the type is being requested. For a **M2_EXACT_VALUE** search, this function returns the status and the type of the physical address in the instance. For a **M2_NEXT_VALUE**

search, it returns the type and status of the lexicographic successor of the physical address seen in the instance.

RETURNS

OK, or **ERROR** if the input parameter is not specified, an interface is no longer valid, or the interface index is incorrect.

ERRNO

S_m2Lib_INVALID_PARAMETER
S_m2Lib_ENTRY_NOT_FOUND
S_m2Lib_IF_CNFG_CHANGED

SEE ALSO

m2IfLib

m2IfRcvAddrEntrySet( )

NAME

m2IfRcvAddrEntrySet( ) – modify the entries of the rcvAddressTable

SYNOPSIS

```
STATUS m2IfRcvAddrEntrySet
    (
    int                 varToSet,   /* entries that need to be modified */
    int                 index,      /* search type */
    M2_IFRCVADDRTBL * pIfReqEntry /* struct containing the new values */
    )
```

DESCRIPTION

This function modifies the status and type fields of a given receive address associated with a given interface. *varToSet* identifies the fields for which the change is being requested. We can also add multicast addresses by creating a new row in the table. The physical address is stripped from the instance value of the SNMP request. This routine does not allow the deletion of a unicast address. Neither does it allow the unicast address to be modified or created.

RETURNS

OK, or **ERROR** if the input parameter is not specified, an interface is no longer valid, the interface index is incorrect, or the **ioctl( )** command to the interface fails.

ERRNO

S_m2Lib_INVALID_PARAMETER
S_m2Lib_ENTRY_NOT_FOUND
S_m2Lib_IF_CNFG_CHANGED

SEE ALSO

m2IfLib, **m2IfInit( )**, **m2IfGroupInfoGet( )**, **m2IfTblEntryGet( )**, **m2IfDelete( )**

m2IfStackEntryGet( )

NAME **m2IfStackEntryGet( )** – get a MIB-II interface-group table entry

SYNOPSIS

```
STATUS m2IfStackEntryGet
    (
    int              search,     /* M2_EXACT_VALUE or M2_NEXT_VALUE */
    int *            pHighIndex, /* the higher layer's ifIndex */
    M2_IFSTACKTBL * pIfReqEntry /* pointer to the requested entry */
    )
```

DESCRIPTION This routine maps the given high and low indexes to the interfaces in the AVL tree. Using the *high* and *low* indexes, we retrieve the nodes in question and walk through their linked lists to get to the right relation. Once we get to the correct node, we can return the values based on the **M2_EXACT_VALUE** and the **M2_NEXT_VALUE** searches.

RETURNS **OK**, or **ERROR** if the input parameter is not specified, or a match is not found.

ERRNO **S_m2Lib_INVALID_PARAMETER**
S_m2Lib_ENTRY_NOT_FOUND

SEE ALSO **m2IfLib**

m2IfStackEntrySet( )

NAME **m2IfStackEntrySet( )** – modify the status of a relationship

SYNOPSIS

```
STATUS m2IfStackEntrySet
    (
    int              highIndex,  /* The higher layer's ifIndex */
    M2_IFSTACKTBL * pIfReqEntry /* The requested entry */
    )
```

DESCRIPTION This routine selects the interfaces specified in the input parameters *pIfReqEntry* and *highIndex* and sets the interface's status to the requested state.

RETURNS **OK**, or **ERROR** if the input parameter is not specified, an interface is no longer valid, or the interface index is incorrect.

M

ERRNO **S_m2Lib_INVALID_PARAMETER**
S_m2Lib_ENTRY_NOT_FOUND
S_m2Lib_IF_CNFG_CHANGED

SEE ALSO **m2IfLib**

m2IfStackTblUpdate( )

NAME **m2IfStackTblUpdate( )** – update the relationship between the sub-layers

SYNOPSIS

```
STATUS m2IfStackTblUpdate
    (
    UINT lowerIndex,          /* The ifIndex of the lower sub-layer */
    UINT higherIndex,         /* The ifIndex of the higher sub-layer */
    int  action               /* insert or remove */
    )
```

DESCRIPTION This function must be called to setup the relationship between the ifIndex values for each sub-layer. This information is required to support the ifStackTable for RFC 2233. Using this data, we can easily determine which sub-layer runs on top of which other.

action is either **M2_STACK_TABLE_INSERT** or **M2_STACK_TABLE_REMOVE**.

Each AVL node keeps a linked list of all the layers that are directly beneath it. Thus by walking through the AVL nodes in an orderly way, we can understand the relationships between all the interfaces.

RETURNS **OK** upon successful addition
ERROR otherwise.

SEE ALSO **m2IfLib**

m2IfTableUpdate( )

NAME **m2IfTableUpdate( )** – insert or remove an entry in the ifTable

SYNOPSIS

```
STATUS m2IfTableUpdate
    (
    struct ifNet * pIfNet,
    UINT           status,    /* attaching or detaching */
    int (* if_ioctl) (struct socket*,u_long,caddr_t),
              /* protocol-specific ioctl or null for default (ethernet) */
    STATUS (* addr_get) (struct ifnet* , M2_IFINDEX* )
              /* func to grab the interface's addrs, null */
              /* for default (ethernet) */
    )
```

DESCRIPTION This routine is called by **if_attach** and **if_detach** to insert/remove an entry from the local **m2IfLib** ifTable. The status can be either **M2_IF_TABLE_INSERT** or **M2_IF_TABLE_REMOVE**. The ifIndex that is searched for in the AVL tree is specified in given the ifnet struct. ***if_ioctl*** is a function pointer to change the flags on the interface. *addr_get* is a function pointer to add the interface's addresses to ifRcvAddressTable. Ethernet interfaces can use **NULL** for both function pointers, other interfaces will need to pass an appropriate function.

RETURNS **ERROR** if entry does not exist, **OK** if the entry was deleted

SEE ALSO **m2IfLib**

m2IfTblEntryGet( )

NAME **m2IfTblEntryGet( )** – get a MIB-II interface-group table entry

SYNOPSIS

```
STATUS m2IfTblEntryGet
    (
    int    search,             /* M2_EXACT_VALUE or M2_NEXT_VALUE */
    void * pIfReqEntry         /* pointer to requested interface entry */
    )
```

DESCRIPTION This routine maps the MIB-II interface index to the system's internal interface index. The internal representation is in the form of a balanced AVL tree indexed by ifIndex of the interface. The *search* parameter is set to either **M2_EXACT_VALUE** or **M2_NEXT_VALUE**; for a discussion of its use, see the manual entry for **m2Lib**. The interface table values are returned in a structure of type **M2_DATA**, which is passed as the second argument to this routine.

RETURNS **OK**, or **ERROR** if the input parameter is not specified, or a match is not found.

ERRNO **S_m2Lib_INVALID_PARAMETER**
S_m2Lib_ENTRY_NOT_FOUND

SEE ALSO **m2IfLib**, **m2Lib**, **m2IfInit( )**, **m2IfGroupInfoGet( )**, **m2IfTblEntrySet( )**, **m2IfDelete( )**

m2IfTblEntrySet( )

NAME **m2IfTblEntrySet( )** – set the state of a MIB-II interface entry to UP or DOWN

SYNOPSIS

```
STATUS m2IfTblEntrySet
    (
    void * pIfReqEntry        /* pointer to requested entry to change */
    )
```

DESCRIPTION This routine selects the interface specified in the input parameter *pIfReqEntry* and sets the interface parameters to the requested state. It is the responsibility of the calling routine to set the interface index, and to make sure that the state specified in the **ifAdminStatus** field of the structure at *pIfTblEntry* is a valid MIB-II state, up(1) or down(2).

The fields that can be modified by this routine are the following: ifAdminStatus, ifAlias, ifLinkUpDownTrapEnable and ifName.

RETURNS **OK**, or **ERROR** if the input parameter is not specified, an interface is no longer valid, the interface index is incorrect, or the **ioctl( )** command to the interface fails.

ERRNO **S_m2Lib_INVALID_PARAMETER**
S_m2Lib_ENTRY_NOT_FOUND
S_m2Lib_IF_CNFG_CHANGED

SEE ALSO **m2IfLib**, **m2IfInit( )**, **m2IfGroupInfoGet( )**, **m2IfTblEntryGet( )**, **m2IfDelete( )**

m2IfVariableUpdate( )

NAME

m2IfVariableUpdate( ) – update the contents of an interface non-counter object

SYNOPSIS

```
STATUS m2IfVariableUpdate
    (
    M2_ID * pId,              /* The pointer to the device M2_ID object */
    UINT    varId,            /* Variable to update */
    caddr_t pData             /* Data to use */
    )
```

DESCRIPTION

This function is used to update an interface variable. The variable is specified by varId and the data to use is specified by pData. Note that different variable expect different types of data. Here is a list of the variables and the type of data expected. Therefore, *pData* will be cast to the type listed below for each variable.

Variable	Cast to Type
ifDescr	char *
ifType	UINT
ifMtu	ULONG
ifSpeed	ULONG
ifPhysAddress	**M2_PHYADDR** *
ifAdminStatus	ULONG
ifOperStatus	ULONG
ifLastChange	ULONG
ifOutQLen	ULONG
ifSpecific	**M2_OBJECTID** *
ifName	char *
ifLinkUpDownTrapEnable	UINT
ifHighSpeed	ULONG
ifPromiscuousMode	UINT
ifConnectorPresent	UINT
ifAlias	char *

RETURNS

ERROR, if the **M2_ID** is **NULL**; **OK**, if the variable was updated.

SEE ALSO

m2IfLib

m2IfVarUpdateRtnInstall( )

NAME

m2IfVarUpdateRtnInstall( ) – install an interface variable update routine

SYNOPSIS

```
STATUS m2IfVarUpdateRtnInstall
    (
    M2_ID *            pId,
    M2_VAR_UPDATE_RTN pRtn
    )
```

DESCRIPTION

This function installs a routine in the **M2_ID**. This routine is able to update a single specified interface variable.

RETURNS

ERROR if the **M2_ID** is **NULL**, **OK** if the routine was installed.

SEE ALSO

m2IfLib

m2Init( )

NAME

m2Init( ) – initialize the SNMP MIB-2 library

SYNOPSIS

```
STATUS m2Init
    (
    char *         pMib2SysDescr,    /* sysDescr */
    char *         pMib2SysContact,  /* sysContact */
    char *         pMib2SysLocation, /* sysLocation */
    M2_OBJECTID *  pMib2SysObjectId, /* sysObjectID */
    FUNCPTR        pTrapRtn,         /* link up/down -trap routine */
    void *         pTrapArg,         /* trap routine arg */
    int            maxRouteTableSize /* max size of routing table */
    )
```

DESCRIPTION

This routine initializes the MIB-2 library by calling the initialization routines for each MIB-2 group. The parameters *pMib2SysDescrpMib2SysContact*, *pMib2SysLocation*, and *pMib2SysObjectId* are passed directly to **m2SysInit( )**; *pTrapRtn* and *pTrapArg* are passed directly to **m2IfInit( )**; and *maxRouteTableSize* is passed to **m2IpInit( )**.

RETURNS

OK if successful, otherwise **ERROR**.

SEE ALSO

m2Lib, **m2SysInit( )**, **m2TcpInit( )**, **m2UdpInit( )**, **m2IcmpInit( )**, **m2IfInit( )**, **m2IpInit( )**

m2IpAddrTblEntryGet( )

NAME

m2IpAddrTblEntryGet( ) – get an IP MIB-II address entry

SYNOPSIS

```
STATUS m2IpAddrTblEntryGet
    (
    int             search,         /* M2_EXACT_VALUE or M2_NEXT_VALUE */
    M2_IPADDRTBL * pIpAddrTblEntry /* ptr to requested IP address entry */
    )
```

DESCRIPTION

This routine traverses the IP address table and does an **M2_EXACT_VALUE** or a **M2_NEXT_VALUE** search based on the *search* parameter. The calling routine is responsible for supplying a valid MIB-II entry index in the input structure *pIpAddrTblEntry*. The index is the local IP address. The first entry in the table is retrieved by doing a NEXT search with the index field set to zero.

RETURNS

OK, **ERROR** if the input parameter is not specified, or a match is not found.

ERRNO

S_m2Lib_INVALID_PARAMETER
S_m2Lib_ENTRY_NOT_FOUND

SEE ALSO

m2IpLib, **m2Lib**, **m2IpInit( )**, **m2IpGroupInfoGet( )**, **m2IpGroupInfoSet( )**, **m2IpAtransTblEntrySet( )**, **m2IpRouteTblEntryGet( )**, **m2IpRouteTblEntrySet( )**, **m2IpDelete( )**

m2IpAtransTblEntryGet( )

NAME

m2IpAtransTblEntryGet( ) – get a MIB-II ARP table entry

SYNOPSIS

```
STATUS m2IpAtransTblEntryGet
    (
    int               search,       /* M2_EXACT_VALUE or M2_NEXT_VALUE */
    M2_IPATRANSTBL * pReqIpAtEntry /* ptr to the requested ARP entry */
    )
```

DESCRIPTION

This routine traverses the ARP table and does an **M2_EXACT_VALUE** or a **M2_NEXT_VALUE** search based on the *search* parameter. The calling routine is responsible for supplying a valid MIB-II entry index in the input structure *pReqIpatEntry*. The index is made up of the network interface index and the IP address corresponding to the physical address. The first entry in the table is retrieved by doing a NEXT search with the index fields set to zero.

RETURNS
OK, **ERROR** if the input parameter is not specified, or a match is not found.

ERRNO
S_m2Lib_INVALID_PARAMETER
S_m2Lib_ENTRY_NOT_FOUND

SEE ALSO
m2IpLib, **m2Lib**, **m2IpInit( )**, **m2IpGroupInfoGet( )**, **m2IpGroupInfoSet( )**, **m2IpAtransTblEntrySet( )**, **m2IpRouteTblEntryGet( )**, **m2IpRouteTblEntrySet( )**, **m2IpDelete( )**

m2IpAtransTblEntrySet( )

NAME
m2IpAtransTblEntrySet( ) – add, modify, or delete a MIB-II ARP entry

SYNOPSIS
```
STATUS m2IpAtransTblEntrySet
    (
    M2_IPATRANSTBL * pReqIpAtEntry /* pointer to MIB-II ARP entry */
    )
```

DESCRIPTION
This routine traverses the ARP table for the entry specified in the parameter *pReqIpAtEntry*. An ARP entry can be added, modified, or deleted. A MIB-II entry index is specified by the destination IP address and the physical media address. A new ARP entry can be added by specifying all the fields in the parameter *pReqIpAtEntry*. An entry can be modified by specifying the MIB-II index and the field that is to be modified. An entry is deleted by specifying the index and setting the type field in the input parameter *pReqIpAtEntry* to the MIB-II value "invalid" (2).

RETURNS
OK, or **ERROR** if the input parameter is not specified, the physical address is not specified for an add/modify request, or the **ioctl( )** request to the ARP module fails.

ERRNO
S_m2Lib_INVALID_PARAMETER
S_m2Lib_ARP_PHYSADDR_NOT_SPECIFIED

SEE ALSO
m2IpLib, **m2IpInit( )**, **m2IpGroupInfoGet( )**, **m2IpGroupInfoSet( )**, **m2IpAddrTblEntryGet( )**, **m2IpRouteTblEntryGet( )**, **m2IpRouteTblEntrySet( )**, **m2IpDelete( )**

m2IpDelete( )

NAME **m2IpDelete( )** – delete all resources used to access the IP group

SYNOPSIS

```
STATUS m2IpDelete (void)
```

DESCRIPTION This routine frees all the resources allocated when the IP group was initialized. The IP group should not be accessed after this routine has been called.

RETURNS **OK**, always.

SEE ALSO **m2IpLib**, **m2IpInit( )**, **m2IpGroupInfoGet( )**, **m2IpGroupInfoSet( )**, **m2IpAddrTblEntryGet( )**, **m2IpAtransTblEntrySet( )**, **m2IpRouteTblEntryGet( )**, **m2IpRouteTblEntrySet( )**

m2IpGroupInfoGet( )

NAME **m2IpGroupInfoGet( )** – get the MIB-II IP-group scalar variables

SYNOPSIS

```
STATUS m2IpGroupInfoGet
    (
    M2_IP * pIpInfo    /* pointer to IP MIB-II global group variables */
    )
```

DESCRIPTION This routine fills in the IP structure at *pIpInfo* with the values of MIB-II IP global variables.

RETURNS **OK**, or **ERROR** if *pIpInfo* is not a valid pointer.

ERRNO **S_m2Lib_INVALID_PARAMETER**

SEE ALSO **m2IpLib**, **m2IpInit( )**, **m2IpGroupInfoSet( )**, **m2IpAddrTblEntryGet( )**, **m2IpAtransTblEntrySet( )**, **m2IpRouteTblEntryGet( )**, **m2IpRouteTblEntrySet( )**, **m2IpDelete( )**

m2IpGroupInfoSet( )

NAME

m2IpGroupInfoSet( ) – set MIB-II IP-group variables to new values

SYNOPSIS

```
STATUS m2IpGroupInfoSet
    (
    unsigned int varToSet,  /* bit field used to set variables */
    M2_IP *      pIpInfo    /* ptr to the MIB-II IP group global variables */
    )
```

DESCRIPTION

This routine sets one or more variables in the IP group, as specified in the input structure *pIpInfo* and the bit field parameter *varToSet*.

RETURNS

OK, or **ERROR** if *pIpInfo* is not a valid pointer, or *varToSet* has an invalid bit field.

ERRNO

S_m2Lib_INVALID_PARAMETER
S_m2Lib_INVALID_VAR_TO_SET

SEE ALSO

m2IpLib, **m2IpInit( )**, **m2IpGroupInfoGet( )**, **m2IpAddrTblEntryGet( )**, **m2IpAtransTblEntrySet( )**, **m2IpRouteTblEntryGet( )**, **m2IpRouteTblEntrySet( )**, **m2IpDelete( )**

m2IpInit( )

NAME

m2IpInit( ) – initialize MIB-II IP-group access

SYNOPSIS

```
STATUS m2IpInit
    (
    int maxRouteTableSize     /* max size of routing table */
    )
```

DESCRIPTION

This routine allocates the resources needed to allow access to the MIB-II IP variables. This routine must be called before any IP variables can be accessed. The parameter *maxRouteTableSize* is used to increase the default size of the MIB-II route table cache.

RETURNS

OK, or **ERROR** if the route table or the route semaphore cannot be allocated.

ERRNO

S_m2Lib_CANT_CREATE_ROUTE_SEM

SEE ALSO **m2IpLib**, **m2IpGroupInfoGet( )**, **m2IpGroupInfoSet( )**, **m2IpAddrTblEntryGet( )**, **m2IpAtransTblEntrySet( )**, **m2IpRouteTblEntryGet( )**, **m2IpRouteTblEntrySet( )**, **m2IpDelete( )**

m2IpRouteTblEntryGet( )

NAME **m2IpRouteTblEntryGet( )** – get a MIB-2 routing table entry

SYNOPSIS

```
STATUS m2IpRouteTblEntryGet
    (
    int               search,           /* M2_EXACT_VALUE or M2_NEXT_VALUE */
    M2_IPROUTETBL * pIpRouteTblEntry /* route table entry */
    )
```

DESCRIPTION This routine retrieves MIB-II information about an entry in the network routing table and returns it in the caller-supplied structure *pIpRouteTblEntry*.

The routine compares routing table entries to the address specified by the **ipRouteDest** member of the *pIpRouteTblEntry* structure, and retrieves an entry chosen by the *search* type (**M2_EXACT_VALUE** or **M2_NEXT_VALUE**, as described in the manual entry for **m2Lib**).

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

ERRNO **S_m2Lib_INVALID_PARAMETER**
S_m2Lib_ENTRY_NOT_FOUND

SEE ALSO **m2IpLib**, **m2Lib**, **m2IpInit( )**, **m2IpGroupInfoGet( )**, **m2IpGroupInfoSet( )**, **m2IpAddrTblEntryGet( )**, **m2IpRouteTblEntryGet( )**, **m2IpRouteTblEntrySet( )**, **m2IpDelete( )**

m2IpRouteTblEntrySet( )

NAME **m2IpRouteTblEntrySet( )** – set a MIB-II routing table entry

SYNOPSIS

```
STATUS m2IpRouteTblEntrySet
    (
    int               varToSet,         /* variable to set */
    M2_IPROUTETBL * pIpRouteTblEntry /* route table entry */
    )
```

DESCRIPTION
This routine adds, changes, or deletes a network routing table entry. The table entry to be modified is specified by the **ipRouteDest** and **ipRouteNextHop** members of the *pIpRouteTblEntry* structure.

The *varToSet* parameter is a bit-field mask that specifies which values in the route table entry are to be set.

If *varToSet* has the **M2_IP_ROUTE_TYPE** bit set and **ipRouteType** has the value of **M2_ROUTE_TYPE_INVALID**, then the routing table entry is deleted.

If *varToSet* has the either the **M2_IP_ROUTE_DEST**, **M2_IP_ROUTE_NEXT_HOP** and the **M2_IP_ROUTE_MASK** bits set, then a new route entry is added to the table.

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

SEE ALSO
m2IpLib, **m2IpInit( )**, **m2IpGroupInfoGet( )**, **m2IpGroupInfoSet( )**, **m2IpAddrTblEntryGet( )**, **m2IpRouteTblEntryGet( )**, **m2IpRouteTblEntrySet( )**, **m2IpDelete( )**

---

m2RipDelete( )

NAME
m2RipDelete( ) – delete the RIP MIB support

SYNOPSIS
```
STATUS m2RipDelete (void)
```

DESCRIPTION
This routine should be called after all **m2RipLib** calls are completed.

RETURNS
OK, always.

SEE ALSO
m2RipLib

m2RipGlobalCountersGet( )

NAME

m2RipGlobalCountersGet( ) – get MIB-II RIP-group global counters

SYNOPSIS

```
STATUS m2RipGlobalCountersGet
    (
    M2_RIP2_GLOBAL_GROUP* pRipGlobal
    )
```

DESCRIPTION

This routine fills in an **M2_RIP2_GLOBAL_GROUP** structure pointed to by *pRipGlobal* with the values of the MIB-II RIP-group global counters.

RETURNS

OK or **ERROR**.

ERRNO

S_m2Lib_INVALID_PARAMETER

SEE ALSO

m2RipLib, **m2RipInit( )**

m2RipIfConfEntryGet( )

NAME

m2RipIfConfEntryGet( ) – get MIB-II RIP-group interface entry

SYNOPSIS

```
STATUS m2RipIfConfEntryGet
    (
    int                   search,
    M2_RIP2_IFCONF_ENTRY* pRipIfConf
    )
```

DESCRIPTION

This routine retrieves the interface configuration for the interface serving the subnet of the IP address contained in the **M2_RIP2_IFCONF_ENTRY** structure passed to it. *pRipIfConf* is a pointer to an **M2_RIP2_IFCONF_ENTRY** structure which the routine will fill in upon successful completion.

This routine either returns an exact match if *search* is **M2_EXACT_VALUE**, or the next value greater than or equal to the value supplied if the *search* is **M2_NEXT_VALUE**.

RETURNS

OK, or **ERROR** if *pRipIfConf* was invalid or the interface was not found.

ERRNO

S_m2Lib_INVALID_PARAMETER
S_m2Lib_ENTRY_NOT_FOUND

SEE ALSO

m2RipLib, **m2RipInit( )**

m2RipIfConfEntrySet( )

NAME **m2RipIfConfEntrySet( )** – set MIB-II RIP-group interface entry

SYNOPSIS

```
STATUS m2RipIfConfEntrySet
    (
    unsigned int           varToSet,
    M2_RIP2_IFCONF_ENTRY* pRipIfConf
    )
```

DESCRIPTION This routine sets the interface configuration for the interface serving the subnet of the IP address contained in the **M2_RIP2_IFCONF_ENTRY** structure.

pRipIfConf is a pointer to an **M2_RIP2_IFCONF_ENTRY** structure which the routine places into the system based on the *varToSet* value.

RETURNS **OK**, or **ERROR** if *pRipIfConf* is invalid or the interface cannot be found.

ERRNO **S_m2Lib_INVALID_PARAMETER**
S_m2Lib_ENTRY_NOT_FOUND

SEE ALSO **m2RipLib**, **m2RipInit( )**

m2RipIfStatEntryGet( )

NAME **m2RipIfStatEntryGet( )** – get MIB-II RIP-group interface entry

SYNOPSIS

```
STATUS m2RipIfStatEntryGet
    (
    int                    search,
    M2_RIP2_IFSTAT_ENTRY* pRipIfStat
    )
```

DESCRIPTION This routine retrieves the interface statistics for the interface serving the subnet of the IP address contained in the **M2_RIP2_IFSTAT_ENTRY** structure. *pRipIfStat* is a pointer to an **M2_RIP2_IFSTAT_ENTRY** structure which the routine will fill in upon successful completion.

This routine either returns an exact match if *search* is **M2_EXACT_VALUE**, or the next value greater than or equal to the value supplied if the *search* is **M2_NEXT_VALUE**.

RETURNS **OK**, or **ERROR** if either *pRipIfStat* is invalid or an exact match failed.

ERRNO **S_m2Lib_INVALID_PARAMETER**
S_m2Lib_ENTRY_NOT_FOUND

SEE ALSO **m2RipLib**, **m2RipInit( )**

m2RipInit( )

NAME **m2RipInit( )** – initialize the RIP MIB support

SYNOPSIS

```
STATUS m2RipInit (void)
```

DESCRIPTION This routine sets up the RIP MIB and should be called before any other **m2RipLib** routine.

RETURNS **OK**, always.

SEE ALSO **m2RipLib**

m2SysDelete( )

NAME **m2SysDelete( )** – delete resources used to access the MIB-II system group

SYNOPSIS

```
STATUS m2SysDelete (void)
```

DESCRIPTION This routine frees all the resources allocated at the time the group was initialized. Do not access the system group after calling this routine.

RETURNS **OK**, always.

SEE ALSO **m2SysLib**, **m2SysInit( )**, **m2SysGroupInfoGet( )**, **m2SysGroupInfoSet( )**.

m2SysGroupInfoGet( )

NAME **m2SysGroupInfoGet( )** – get system-group MIB-II variables

SYNOPSIS

```
STATUS m2SysGroupInfoGet
    (
    M2_SYSTEM * pSysInfo      /* pointer to MIB-II system group structure */
    )
```

DESCRIPTION This routine fills in the structure at *pSysInfo* with the values of MIB-II system-group variables.

RETURNS **OK**, or **ERROR** if *pSysInfo* is not a valid pointer.

ERRNO **S_m2Lib_INVALID_PARAMETER**

SEE ALSO **m2SysLib**, **m2SysInit( )**, **m2SysGroupInfoSet( )**, **m2SysDelete( )**

m2SysGroupInfoSet( )

NAME **m2SysGroupInfoSet( )** – set system-group MIB-II variables to new values

SYNOPSIS

```
STATUS m2SysGroupInfoSet
    (
    unsigned int varToSet,    /* bit field of variables to set */
    M2_SYSTEM *  pSysInfo     /* pointer to the system structure */
    )
```

DESCRIPTION This routine sets one or more variables in the system group as specified in the input structure at *pSysInfo* and the bit field parameter *varToSet*.

RETURNS **OK**, or **ERROR** if *pSysInfo* is not a valid pointer, or *varToSet* has an invalid bit field.

ERRNO **S_m2Lib_INVALID_PARAMETER**
S_m2Lib_INVALID_VAR_TO_SET

SEE ALSO **m2SysLib**, **m2SysInit( )**, **m2SysGroupInfoGet( )**, **m2SysDelete( )**

m2SysInit( )

NAME **m2SysInit( )** – initialize MIB-II system-group routines

SYNOPSIS

```
STATUS m2SysInit
    (
    char *        pMib2SysDescr,    /* pointer to MIB-2 sysDescr */
    char *        pMib2SysContact,  /* pointer to MIB-2 sysContact */
    char *        pMib2SysLocation, /* pointer to MIB-2 sysLocation */
    M2_OBJECTID * pObjectId         /* pointer to MIB-2 ObjectId */
    )
```

DESCRIPTION This routine allocates the resources needed to allow access to the system-group MIB-II variables. This routine must be called before any system-group variables can be accessed. The input parameters *pMib2SysDescr*, *pMib2SysContact*, *pMib2SysLocation*, and *pObjectId* are optional. The parameters *pMib2SysDescr*, *pObjectId* are read only, as specified by MIB-II, and can be set only by this routine.

RETURNS **OK**, always.

ERRNO **S_m2Lib_CANT_CREATE_SYS_SEM**

SEE ALSO **m2SysLib**, **m2SysGroupInfoGet( )**, **m2SysGroupInfoSet( )**, **m2SysDelete( )**

m2TcpConnEntryGet( )

NAME **m2TcpConnEntryGet( )** – get a MIB-II TCP connection table entry

SYNOPSIS

```
STATUS m2TcpConnEntryGet
    (
    int             search,           /* M2_EXACT_VALUE or M2_NEXT_VALUE */
    M2_TCPCONNTBL * pReqTcpConnEntry /* input = Index, Output = Entry */
    )
```

DESCRIPTION This routine traverses the TCP table of users and does an **M2_EXACT_VALUE** or a **M2_NEXT_VALUE** search based on the *search* parameter (see **m2Lib**). The calling routine is responsible for supplying a valid MIB-II entry index in the input structure *pReqTcpConnEntry*. The index is made up of the local IP address, the local port number, the remote IP address, and the remote port. The first entry in the table is retrieved by doing a **M2_NEXT_VALUE** search with the index fields set to zero.

RETURNS **OK**, or **ERROR** if the input parameter is not specified or a match is not found.

ERRNO **S_m2Lib_INVALID_PARAMETER**
S_m2Lib_ENTRY_NOT_FOUND

SEE ALSO **m2TcpLib**, **m2Lib**, **m2TcpInit( )**, **m2TcpGroupInfoGet( )**, **m2TcpConnEntrySet( )**, **m2TcpDelete( )**

---

m2TcpConnEntrySet( )

NAME **m2TcpConnEntrySet( )** – set a TCP connection to the closed state

SYNOPSIS

```
STATUS m2TcpConnEntrySet
    (
    M2_TCPCONNTBL * pReqTcpConnEntry /* pointer to TCP connection to close */
    )
```

DESCRIPTION This routine traverses the TCP connection table and searches for the connection specified by the input parameter *pReqTcpConnEntry*. The calling routine is responsible for providing a valid index as the input parameter *pReqTcpConnEntry*. The index is made up of the local IP address, the local port number, the remote IP address, and the remote port. This call can only succeed if the connection is in the MIB-II state "deleteTCB" (12). If a match is found, the socket associated with the TCP connection is closed.

RETURNS **OK**, or **ERROR** if the input parameter is invalid, the state of the connection specified at *pReqTcpConnEntry* is not "closed," the specified connection is not found, a socket is not associated with the connection, or the **close( )** call fails.

SEE ALSO **m2TcpLib**, **m2TcpInit( )**, **m2TcpGroupInfoGet( )**, **m2TcpConnEntryGet( )**, **m2TcpDelete( )**

---

m2TcpDelete( )

NAME **m2TcpDelete( )** – delete all resources used to access the TCP group

SYNOPSIS

```
STATUS m2TcpDelete (void)
```

DESCRIPTION This routine frees all the resources allocated at the time the group was initialized. The TCP group should not be accessed after this routine has been called.

RETURNS **OK**, always.

SEE ALSO **m2TcpLib**, **m2TcpInit( )**, **m2TcpGroupInfoGet( )**, **m2TcpConnEntryGet( )**, **m2TcpConnEntrySet( )**

---

m2TcpGroupInfoGet( )

NAME **m2TcpGroupInfoGet( )** – get MIB-II TCP-group scalar variables

SYNOPSIS

```
STATUS m2TcpGroupInfoGet
    (
    M2_TCPINFO * pTcpInfo     /* pointer to the TCP group structure */
    )
```

DESCRIPTION This routine fills in the TCP structure pointed to by *pTcpInfo* with the values of MIB-II TCP-group scalar variables.

RETURNS **OK**, or **ERROR** if *pTcpInfo* is not a valid pointer.

ERRNO **S_m2Lib_INVALID_PARAMETER**

SEE ALSO **m2TcpLib**, **m2TcpInit( )**, **m2TcpConnEntryGet( )**, **m2TcpConnEntrySet( )**, **m2TcpDelete( )**

---

m2TcpInit( )

NAME **m2TcpInit( )** – initialize MIB-II TCP-group access

SYNOPSIS

```
STATUS m2TcpInit (void)
```

DESCRIPTION This routine allocates the resources needed to allow access to the TCP MIB-II variables. This routine must be called before any TCP variables can be accessed.

RETURNS **OK**, always.

SEE ALSO **m2TcpLib**, **m2TcpGroupInfoGet( )**, **m2TcpConnEntryGet( )**, **m2TcpConnEntrySet( )**, **m2TcpDelete( )**

m2UdpDelete( )

NAME **m2UdpDelete( )** – delete all resources used to access the UDP group

SYNOPSIS

```
STATUS m2UdpDelete (void)
```

DESCRIPTION This routine frees all the resources allocated at the time the group was initialized. The UDP group should not be accessed after this routine has been called.

RETURNS **OK**, always.

SEE ALSO **m2UdpLib**, **m2UdpInit( )**, **m2UdpGroupInfoGet( )**, **m2UdpTblEntryGet( )**

m2UdpGroupInfoGet( )

NAME **m2UdpGroupInfoGet( )** – get MIB-II UDP-group scalar variables

SYNOPSIS

```
STATUS m2UdpGroupInfoGet
    (
    M2_UDP * pUdpInfo          /* pointer to the UDP group structure */
    )
```

DESCRIPTION This routine fills in the UDP structure at *pUdpInfo* with the MIB-II UDP scalar variables.

RETURNS **OK**, or **ERROR** if *pUdpInfo* is not a valid pointer.

ERRNO **S_m2Lib_INVALID_PARAMETER**

SEE ALSO **m2UdpLib**, **m2UdpInit( )**, **m2UdpTblEntryGet( )**, **m2UdpDelete( )**

m2UdpInit( )

NAME **m2UdpInit( )** – initialize MIB-II UDP-group access

SYNOPSIS

```
STATUS m2UdpInit (void)
```

DESCRIPTION This routine allocates the resources needed to allow access to the UDP MIB-II variables. This routine must be called before any UDP variables can be accessed.

RETURNS **OK**, always.

SEE ALSO **m2UdpLib**, **m2UdpGroupInfoGet( )**, **m2UdpTblEntryGet( )**, **m2UdpDelete( )**

m2UdpTblEntryGet( )

NAME **m2UdpTblEntryGet( )** – get a UDP MIB-II entry from the UDP list of listeners

SYNOPSIS

```
STATUS m2UdpTblEntryGet
    (
    int          search,       /* M2_EXACT_VALUE or M2_NEXT_VALUE */
    M2_UDPTBL * pUdpEntry      /* ptr to the requested entry with index */
    )
```

DESCRIPTION This routine traverses the UDP table of listeners and does an **M2_EXACT_VALUE** or a **M2_NEXT_VALUE** search based on the *search* parameter. The calling routine is responsible for supplying a valid MIB-II entry index in the input structure *pUdpEntry*. The index is made up of the IP address and the local port number. The first entry in the table is retrieved by doing a **M2_NEXT_VALUE** search with the index fields set to zero.

RETURNS **OK**, or **ERROR** if the input parameter is not specified or a match is not found.

ERRNO **S_m2Lib_INVALID_PARAMETER**
S_m2Lib_ENTRY_NOT_FOUND

SEE ALSO **m2UdpLib**, **m2Lib**, **m2UdpInit( )**, **m2UdpGroupInfoGet( )**, **m2UdpDelete( )**

M

mach( )

NAME **mach( )** – return the contents of system register **mach** (also **macl**, **pr**) (SH)

SYNOPSIS

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

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

Similar routines are provided for other system registers (**macl**, **pr**): **macl( )**, **pr( )**. Note that **pc( )** is provided by **usrLib.c**.

RETURNS The contents of register mach (or the requested system register).

SEE ALSO **dbgArchLib**, *VxWorks Programmer's Guide: Debugging*

malloc( )

NAME **malloc( )** – allocate a block of memory from the system memory partition (ANSI)

SYNOPSIS

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

DESCRIPTION This routine allocates a block of memory from the free list. 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.

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

mathHardInit( )

NAME **mathHardInit( )** – initialize hardware floating-point math support

SYNOPSIS

```
void mathHardInit ()
```

DESCRIPTION This routine places the addresses of the hardware high-level math functions (trigonometric functions, etc.) in a set of global variables. This allows the standard math functions (*e.g.*, **sin( )**, **pow( )**) to have a single entry point but to be dispatched to the hardware or software support routines, as specified.

This routine is called from **usrConfig.c** if **INCLUDE_HW_FP** is defined. This definition causes the linker to include the floating-point hardware support library.

Certain routines in the floating-point software emulation library do not have equivalent hardware support routines. (These are primarily routines that handle single-precision floating-point numbers.) If no emulation routine address has already been put in the global variable for this function, the address of a dummy routine that logs an error message is placed in the variable; if an emulation routine address is present (the emulation initialization, via **mathSoftInit( )**, must be done prior to hardware floating-point initialization), the emulation routine address is left alone. In this way, hardware routines will be used for all available functions, while emulation will be used for the missing functions.

RETURNS N/A

SEE ALSO **mathHardLib**, **mathSoftInit( )**

mathSoftInit( )

NAME **mathSoftInit( )** – initialize software floating-point math support

SYNOPSIS

```
void mathSoftInit (void)
```

DESCRIPTION This routine places the addresses of the emulated high-level math functions (trigonometric functions, etc.) in a set of global variables. This allows the standard math functions (*e.g.*, **sin( )**, **pow( )**) to have a single entry point but be dispatched to the hardware or software support routines, as specified.

This routine is called from **usrConfig.c** if **INCLUDE_SW_FP** is defined. This definition causes the linker to include the floating-point emulation library.

M

If the system is to use some combination of emulated as well as hardware coprocessor floating points, then this routine should be called before calling **mathHardInit( )**.

RETURNS N/A

SEE ALSO **mathSoftLib**, **mathHardInit( )**

mblen( )

NAME **mblen( )** – calculate the length of a multibyte character (Unimplemented) (ANSI)

SYNOPSIS

```
int mblen
    (
    const char * s,
    size_t       n
    )
```

DESCRIPTION This multibyte character function is unimplemented in VxWorks.

INCLUDE FILES **stdlib.h**

RETURNS **OK**, or **ERROR** if the parameters are invalid.

SEE ALSO **ansiStdlib**

mbstowcs( )

NAME **mbstowcs( )** – convert a series of multibyte char's to wide char's (Unimplemented) (ANSI)

SYNOPSIS

```
size_t mbstowcs
    (
    wchar_t *    pwcs,
    const char * s,
    size_t       n
    )
```

DESCRIPTION This multibyte character function is unimplemented in VxWorks.

INCLUDE FILES **stdlib.h**

RETURNS **OK**, or **ERROR** if the parameters are invalid.

SEE ALSO **ansiStdlib**

mbtowc( )

NAME

mbtowc( ) – convert a multibyte character to a wide character (Unimplemented) (ANSI)

SYNOPSIS

```
int mbtowc
    (
    wchar_t *    pwc,
    const char * s,
    size_t       n
    )
```

DESCRIPTION

This multibyte character function is unimplemented in VxWorks.

INCLUDE FILES

stdlib.h

RETURNS

OK, or **ERROR** if the parameters are invalid.

SEE ALSO

ansiStdlib

mbufShow( )

NAME

mbufShow( ) – report mbuf statistics

SYNOPSIS

```
void mbufShow (void)
```

DESCRIPTION

This routine displays the distribution of mbufs in the network.

RETURNS

N/A

SEE ALSO

netShow

memAddToPool( )

NAME

memAddToPool( ) – add memory to the system memory partition

SYNOPSIS

```
void memAddToPool
    (
    char *   pPool,           /* pointer to memory block */
    unsigned poolSize         /* block size in bytes */
    )
```

DESCRIPTION

This routine adds memory to the system memory partition, after the initial allocation of memory to the system memory partition.

RETURNS

N/A

SEE ALSO

memPartLib, **memPartAddToPool( )**

memalign( )

NAME

memalign( ) – allocate aligned memory

SYNOPSIS

```
void *memalign
    (
    unsigned alignment,       /* boundary to align to (power of 2) */
    unsigned size             /* number of bytes to allocate */
    )
```

DESCRIPTION

This routine allocates a buffer of size *size* from the system memory partition. 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.

SEE ALSO

memLib

memchr( )

NAME

memchr( ) – search a block of memory for a character (ANSI)

SYNOPSIS

```
void * memchr
    (
    const void * m,          /* block of memory */
    int          c,          /* character to search for */
    size_t       n           /* size of memory to search */
    )
```

DESCRIPTION

This routine searches for the first element of an array of **unsigned char**, beginning at the address *m* with size *n*, that equals *c* converted to an **unsigned char**.

INCLUDE FILES

string.h

RETURNS

If successful, it returns the address of the matching element; otherwise, it returns a null pointer.

SEE ALSO

ansiString

memcmp( )

NAME

memcmp( ) – compare two blocks of memory (ANSI)

SYNOPSIS

```
int memcmp
    (
    const void * s1,         /* array 1 */
    const void * s2,         /* array 2 */
    size_t       n           /* size of memory to compare */
    )
```

DESCRIPTION

This routine compares successive elements from two arrays of **unsigned char**, beginning at the addresses *s1* and *s2* (both of size *n*), until it finds elements that are not equal.

INCLUDE FILES

string.h

RETURNS

If all elements are equal, zero. If elements differ and the differing element from *s1* is greater than the element from *s2*, the routine returns a positive number; otherwise, it returns a negative number.

SEE ALSO

ansiString

memcpy( )

NAME
memcpy( ) – copy memory from one location to another (ANSI)

SYNOPSIS

```
void * memcpy
    (
    void *       destination, /* destination of copy */
    const void * source,      /* source of copy */
    size_t       size         /* size of memory to copy */
    )
```

DESCRIPTION
This routine copies *size* characters from the object pointed to by *source* into the object pointed to by *destination*. If copying takes place between objects that overlap, the behavior is undefined.

INCLUDE FILES
string.h

RETURNS
A pointer to *destination*.

SEE ALSO
ansiString

memDevCreate( )

NAME
memDevCreate( ) – create a memory device

SYNOPSIS

```
STATUS memDevCreate
    (
    char * name,              /* device name */
    char * base,              /* where to start in memory */
    int    length             /* number of bytes */
    )
```

DESCRIPTION
This routine creates a memory device containing a single file. Memory for the device is simply an absolute memory location beginning at *base*. The *length* parameter indicates the size of memory.

For example, to create the device **/mem/cpu0/**, a device for accessing the entire memory of the local processor, the proper call would be:

```
memDevCreate ("/mem/cpu0/", 0, sysMemTop())
```

The device is created with the specified name, start location, and size.

To open a file descriptor to the memory, use **open( )**. Specify a pseudo-file name of the byte offset desired, or open the "raw" file at the beginning and specify a position to seek to. For example, the following call to **open( )** allows memory to be read starting at decimal offset 1000.

```
-> fd = open ("/mem/cpu0/1000", O_RDONLY, 0)
```

Pseudo-file name offsets are scanned with "%d".

WARNING: The **FIOSEEK** operation overrides the offset given via the pseudo-file name at open time.

EXAMPLE

Consider a system configured with two CPUs in the backplane and a separate dual-ported memory board, each with 1 megabyte of memory. The first CPU is mapped at VMEbus address 0x00400000 (4 Meg.), the second at bus address 0x00800000 (8 Meg.), the dual-ported memory board at 0x00c00000 (12 Meg.). Three devices can be created on each CPU as follows. On processor 0:

```
-> memDevCreate ("/mem/local/", 0, sysMemTop())
...
-> memDevCreate ("/mem/cpu1/", 0x00800000, 0x00100000)
...
-> memDevCreate ("/mem/share/", 0x00c00000, 0x00100000)
```

On processor 1:

```
-> memDevCreate ("/mem/local/", 0, sysMemTop())
...
-> memDevCreate ("/mem/cpu0/", 0x00400000, 0x00100000)
...
-> memDevCreate ("/mem/share/", 0x00c00000, 0x00100000)
```

Processor 0 has a local disk. Data or an object module needs to be passed from processor 0 to processor 1. To accomplish this, processor 0 first calls:

```
-> copy </disk1/module.o >/mem/share/0
```

Processor 1 can then be given the load command:

```
-> ld </mem/share/0
```

RETURNS

OK, or **ERROR** if memory is insufficient or the I/O system cannot add the device.

ERRNO

S_ioLib_NO_DRIVER

SEE ALSO

memDrv

memDevCreateDir( )

NAME **memDevCreateDir( )** – create a memory device for multiple files

SYNOPSIS

```
STATUS memDevCreateDir
    (
    char *              name,    /* device name */
    MEM_DRV_DIRENTRY * files,    /* array of dir. entries - not copied */
    int                 numFiles /* number of entries */
    )
```

DESCRIPTION This routine creates a memory device for a collection of files organized into directories. The given array of directory entry records describes a number of files, some of which may be directories, represented by their own directory entry arrays. The structure may be arbitrarily deep. This effectively allows a file system to be created and installed in VxWorks, for essentially read-only use. The file system structure can be created on the host using the **memdrvbuild** utility.

Note that the array supplied is not copied; a reference to it is kept. This array should not be modified after being passed to **memDevCreateDir( )**.

RETURNS **OK**, or **ERROR** if memory is insufficient or the I/O system cannot add the device.

ERRNO **S_ioLib_NO_DRIVER**

SEE ALSO **memDrv**

memDevDelete( )

NAME **memDevDelete( )** – delete a memory device

SYNOPSIS

```
STATUS memDevDelete
    (
    char * name                /* device name */
    )
```

DESCRIPTION This routine deletes a memory device containing a single file or a collection of files. The device is deleted with it own name.

For example, to delete the device created by **memDevCreate (**"**/mem/cpu0/**", 0, **sysMemTop( ))**, the proper call would be:

```
memDevDelete ("/mem/cpu0/");
```

RETURNS **OK**, or **ERROR** if the device doesn't exist.

SEE ALSO **memDrv**

memDrv( )

NAME **memDrv( )** – install a memory driver

SYNOPSIS

```
STATUS memDrv (void)
```

DESCRIPTION This routine initializes the memory driver. It must be called first, before any other routine in the driver.

RETURNS **OK**, or **ERROR** if the I/O system cannot install the driver.

SEE ALSO **memDrv**

memFindMax( )

NAME **memFindMax( )** – find the largest free block in the system memory partition

SYNOPSIS

```
int memFindMax (void)
```

DESCRIPTION This routine searches for the largest block in the system memory partition free list and returns its size.

RETURNS The size, in bytes, of the largest available block.

SEE ALSO **memLib**, **memPartFindMax( )**

memmove( )

NAME

memmove( ) – copy memory from one location to another (ANSI)

SYNOPSIS

```
void * memmove
    (
    void *       destination, /* destination of copy */
    const void * source,      /* source of copy */
    size_t       size         /* size of memory to copy */
    )
```

DESCRIPTION

This routine copies *size* characters from the memory location *source* to the location *destination*. It ensures that the memory is not corrupted even if *source* and *destination* overlap.

INCLUDE FILES

string.h

RETURNS

A pointer to *destination*.

SEE ALSO

ansiString

memOptionsSet( )

NAME

memOptionsSet( ) – set the debug options for the system memory partition

SYNOPSIS

```
void memOptionsSet
    (
    unsigned options          /* options for system partition */
    )
```

DESCRIPTION

This routine sets the debug options for the system memory partition. Two kinds of errors are detected: attempts to allocate more memory than is available, and bad blocks found when memory is freed. In both cases, the following options can be selected for actions to be taken when the error is detected: (1) return the error status, (2) log an error message and return the error status, or (3) log an error message and suspend the calling task.

These options are discussed in detail in the library manual entry for **memLib**.

RETURNS

N/A

SEE ALSO

memLib, **memPartOptionsSet( )**

memPartAddToPool( )

NAME
memPartAddToPool( ) – add memory to a memory partition

SYNOPSIS

```
STATUS memPartAddToPool
    (
    PART_ID  partId,          /* partition to initialize */
    char *   pPool,           /* pointer to memory block */
    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.

RETURNS
OK or **ERROR**.

ERRNO
S_smObjLib_NOT_INITIALIZED, **S_memLib_INVALID_NBYTES**

SEE ALSO
memPartLib, **smMemLib**, **memPartCreate( )**

memPartAlignedAlloc( )

NAME
memPartAlignedAlloc( ) – allocate aligned memory from a partition

SYNOPSIS

```
void *memPartAlignedAlloc
    (
    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 insures 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.

SEE ALSO
memPartLib

memPartAlloc( )

NAME

memPartAlloc( ) – allocate a block of memory from a partition

SYNOPSIS

```
void *memPartAlloc
    (
    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

S_smObjLib_NOT_INITIALIZED

SEE ALSO

memPartLib, **smMemLib**, **memPartCreate( )**

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. It returns a partition ID, which can then 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.

NOTE: The descriptor for the new partition is allocated out of the system memory partition (*i.e.*, with **malloc( )**).

RETURNS

The partition ID, or **NULL** if there is insufficient memory in the system memory partition for a new partition descriptor.

SEE ALSO

memPartLib, **smMemLib**

memPartFindMax( )

NAME **memPartFindMax( )** – find the size of the largest available free block

SYNOPSIS

```
int memPartFindMax
    (
    PART_ID partId              /* partition ID */
    )
```

DESCRIPTION This routine searches for the largest block in the memory partition free list and returns its size.

RETURNS The size, in bytes, of the largest available block.

ERRNO **S_smObjLib_NOT_INITIALIZED**

SEE ALSO **memLib**, **smMemLib**

memPartFree( )

NAME **memPartFree( )** – free a block of memory in a partition

SYNOPSIS

```
STATUS memPartFree
    (
    PART_ID partId,             /* memory partition to add block to */
    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( )**.

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

ERRNO **S_smObjLib_NOT_INITIALIZED**

SEE ALSO **memPartLib**, **smMemLib**, **memPartAlloc( )**

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.

RETURNS **OK** if the structure has valid data, otherwise **ERROR**.

SEE ALSO **memShow**, **memShow( )**

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. In both cases, the error status is returned. There are four error-handling options that can be individually selected:

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).

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).

These options are discussed in detail in the library manual entry for **memLib**.

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

ERRNO **S_smObjLib_NOT_INITIALIZED**

SEE ALSO **memLib**, **smMemLib**

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( )**.

RETURNS A pointer to the new block of memory, or **NULL** if the call fails.

ERRNO **S_smObjLib_NOT_INITIALIZED**

SEE ALSO **memLib**, **smMemLib**

M

memPartShow( )

NAME
memPartShow( ) – show partition blocks and statistics

SYNOPSIS
```
STATUS memPartShow
    (
    PART_ID partId,           /* partition ID */
    int     type              /* 0 = statistics, 1 = statistics & list */
    )
```

DESCRIPTION
This routine displays statistics about the available and allocated memory in a specified memory partition. It shows the number of bytes, the number of blocks, and the average block size in both free and allocated memory, and also the maximum block size of free memory. It also shows the number of blocks currently allocated and the average allocated block size.

In addition, if *type* is 1, the routine displays a list of all the blocks in the free list of the specified partition.

RETURNS
OK or **ERROR**.

ERRNO
S_smObjLib_NOT_INITIALIZED

SEE ALSO
memShow, **memShow( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

memPartSmCreate( )

NAME
memPartSmCreate( ) – create a shared memory partition (VxMP Opt.)

SYNOPSIS
```
PART_ID memPartSmCreate
    (
    char *   pPool,           /* global address of shared memory area */
    unsigned poolSize         /* size in bytes */
    )
```

DESCRIPTION
This routine creates a shared memory partition that can be used by tasks on all CPUs in the system. It returns a partition ID which can then be passed to generic **memPartLib** routines to manage the partition (*i.e.*, to allocate and free memory blocks in the partition).

pPool is the global address of shared memory dedicated to the partition. The memory area pointed to by *pPool* must be in the same address space as the shared memory anchor and shared memory pool.

poolSize is the size in bytes of shared memory dedicated to the partition.

Before this routine can be called, the shared memory objects facility must be initialized (see **smMemLib**).

NOTE: The descriptor for the new partition is allocated out of an internal dedicated shared memory partition. The maximum number of partitions that can be created is **SM_OBJ_MAX_MEM_PART**.

Memory pool size is rounded down to a 16-byte boundary.

AVAILABILITY
This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS
The partition ID, or **NULL** if there is insufficient memory in the dedicated partition for a new partition descriptor.

ERRNO
S_memLib_NOT_ENOUGH_MEMORY
S_smObjLib_LOCK_TIMEOUT

SEE ALSO
smMemLib, **memLib**

---

memset( )

NAME
memset( ) – set a block of memory (ANSI)

SYNOPSIS
```
void * memset
    (
    void * m,                /* block of memory */
    int    c,                /* character to store */
    size_t size              /* size of memory */
    )
```

DESCRIPTION
This routine stores *c* converted to an **unsigned char** in each of the elements of the array of **unsigned char** beginning at *m*, with size *size*.

INCLUDE FILES
string.h

RETURNS
A pointer to *m*.

SEE ALSO
ansiString

M

memShow( )

NAME

memShow( ) – show system memory partition blocks and statistics

SYNOPSIS

```
void memShow
    (
    int type                    /* 1 = list all blocks in the free list */
    )
```

DESCRIPTION

This routine displays statistics about the available and allocated memory in the system memory partition. It shows the number of bytes, the number of blocks, and the average block size in both free and allocated memory, and also the maximum block size of free memory. It also shows the number of blocks currently allocated and the average allocated block size.

In addition, if *type* is 1, the routine displays a list of all the blocks in the free list of the system partition.

EXAMPLE

```
-> memShow 1
FREE LIST:
  num     addr      size
  --- ---------- ----------
    1   0x3fee18         16
    2   0x3b1434         20
    3    0x4d188    2909400
SUMMARY:
 status   bytes    blocks   avg block  max block
 ------ --------- -------- ---------- ----------
current
   free   2909436        3     969812   2909400
  alloc    969060    16102         60         -
cumulative
  alloc   1143340    16365         69         -
```

RETURNS

N/A

SEE ALSO

memShow, **memPartShow( )**, *VxWorks Programmer's Guide: Target Shell,* **windsh**, *Tornado User's Guide: Shell*

memShowInit( )

NAME **memShowInit( )** – initialize the memory partition show facility

SYNOPSIS

```
void memShowInit (void)
```

DESCRIPTION This routine links the memory partition show facility into the VxWorks system. These routines are included automatically when this show facility is configured into VxWorks using either of the following methods:

– If you use the configuration header files, define **INCLUDE_SHOW_ROUTINES** in **config.h**.

– If you use the Tornado project facility, select **INCLUDE_MEM_SHOW**.

RETURNS N/A

SEE ALSO **memShow**

M

mkdir( )

NAME **mkdir( )** – make a directory

SYNOPSIS

```
STATUS mkdir
    (
    const char * dirName      /* directory name */
    )
```

DESCRIPTION This command creates a new directory in a hierarchical file system. The *dirName* string specifies the name to be used for the new directory, and can be either a full or relative pathname.

This call is supported by the VxWorks NFS and dosFs file systems.

RETURNS **OK**, or **ERROR** if the directory cannot be created.

SEE ALSO **usrFsLib**, **rmdir( )**, *VxWorks Programmer's Guide: Target Shell*

mktime( )

NAME

mktime( ) – convert broken-down time into calendar time (ANSI)

SYNOPSIS

```
time_t mktime
    (
    struct tm * timeptr       /* pointer to broken-down structure */
    )
```

DESCRIPTION

This routine converts the broken-down time, expressed as local time, in the structure pointed to by *timeptr* into a calendar time value with the same encoding as that of the values returned by the **time( )** function. The original values of the **tm_wday** and **tm_yday** components of the **tm** structure are ignored, and the original values of the other components are not restricted to the ranges indicated in **time.h**. On successful completion, the values of **tm_wday** and **tm_yday** are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to the ranges indicated in **time.h**; the final value of **tm_mday** is not set until **tm_mon** and **tm_year** are determined.

INCLUDE FILES

time.h

RETURNS

The calendar time in seconds, or **ERROR** (-1) if calendar time cannot be calculated.

SEE ALSO

ansiTime

mlock( )

NAME

mlock( ) – lock specified pages into memory (POSIX)

SYNOPSIS

```
int mlock
    (
    const void * addr,
    size_t       len
    )
```

DESCRIPTION

This routine guarantees that the specified pages are memory resident. In VxWorks, the *addr* and *len* arguments are ignored, since all pages are memory resident.

RETURNS

0 (**OK**) always.

SEE ALSO

mmanPxLib

mlockall( )

NAME

mlockall( ) – lock all pages used by a process into memory (POSIX)

SYNOPSIS

```
int mlockall
    (
    int flags
    )
```

DESCRIPTION

This routine guarantees that all pages used by a process are memory resident. In VxWorks, the *flags* argument is ignored, since all pages are memory resident.

RETURNS

0 (**OK**) always.

ERRNO

N/A

SEE ALSO

mmanPxLib

mmuPhysToVirt( )

NAME

mmuPhysToVirt( ) – translate a physical address to a virtual address (ARM)

SYNOPSIS

```
void * mmuPhysToVirt
    (
    void * physAddr           /* physical address to be translated */
    )
```

DESCRIPTION

This function converts a physical address to a virtual address using the information contained within the **sysPhysMemDesc** structure of the BSP. This routine may be used both by the BSP MMU initialization and by the **vm(Base)Lib** code.

If the BSP has a default mapping where physical and virtual addresses are not identical, then it must provide routines to the cache and MMU architecture code to convert between physical and virtual addresses. If the mapping described within the **sysPhysMemDesc** structure is accurate, then the BSP may use this routine. If it is not accurate, then routines must be provided within the BSP that are accurate.

NOTE: This routine simply performs a linear search through the **sysPhysMemDesc** structure looking for the first entry with an address range that includes the given address. Typically, the performance of this should not be a problem, as this routine will generally be called to translate RAM addresses, and by convention, the RAM entries come first in the structure. If this becomes an issue, the routine could be changed so that a separate structure to **sysPhysMemDesc** is used, containing the information in a more quickly accessible form. In any case, if this is not satisfactory, the BSP can provide its own routines.

SEE ALSO **mmuMapLib**, **mmuVirtToPhys**

RETURNS the virtual address

mmuPro32LibInit( )

NAME

mmuPro32LibInit( ) – initialize module

SYNOPSIS

```
STATUS mmuPro32LibInit
    (
    int pageSize             /* system pageSize (must be 4KB or 4MB) */
    )
```

DESCRIPTION

Build a dummy translation table that will hold the page table entries for the global translation table. The MMU remains disabled upon completion.

RETURNS

OK if no error, **ERROR** otherwise

ERRNO

S_mmuLib_INVALID_PAGE_SIZE

SEE ALSO

mmuPro32Lib

mmuSh7700LibInit( )

NAME

mmuSh7700LibInit( ) – initialize module

SYNOPSIS

```
STATUS mmuSh7700LibInit
    (
    int pageSize
    )
```

DESCRIPTION

Build a dummy translation table that will hold the page table entries for the global translation table. The MMU remains disabled upon completion. Note that this routine is global so that it may be referenced in **usrConfig.c** to pull in the correct **mmuLib** for the specific architecture.

RETURNS

OK or **ERROR**

SEE ALSO

mmuSh7700Lib

mmuSh7750LibInit( )

NAME

mmuSh7750LibInit( ) – initialize module

SYNOPSIS

```
STATUS mmuSh7750LibInit
    (
    int pageSize
    )
```

DESCRIPTION

Build a dummy translation table that will hold the page table entries for the global translation table. The MMU remains disabled upon completion. Note that this routine is global so that it may be referenced in **usrConfig.c** to pull in the correct **mmuLib** for the specific architecture.

RETURNS

OK or **ERROR**

SEE ALSO

mmuSh7750Lib

mmuVirtToPhys( )

NAME

mmuVirtToPhys( ) – translate a virtual address to a physical address (ARM)

SYNOPSIS

```
void * mmuVirtToPhys
    (
    void * virtAddr           /* virtual address to be translated */
    )
```

DESCRIPTION

This function converts a virtual address to a physical address using the information contained within the **sysPhysMemDesc** structure of the BSP. This routine may be used both by the BSP MMU initialization and by the **vm(Base)Lib** code.

If the BSP has a default mapping where physical and virtual addresses are not identical, then it must provide routines to the cache and MMU architecture code to convert between physical and virtual addresses. If the mapping described within the **sysPhysMemDesc** structure is accurate, then the BSP may use this routine. If it is not accurate, then routines must be provided within the BSP that are accurate.

NOTE: This routine simply performs a linear search through the **sysPhysMemDesc** structure looking for the first entry with an address range that includes the given address. Typically, the performance of this should not be a problem, as this routine will generally be called to translate RAM addresses, and by convention, the RAM entries come first in the structure. If this becomes an issue, the routine could be changed so that a separate structure to **sysPhysMemDesc** is used, containing the information in a more quickly accessible form. In any case, if this is not satisfactory, the BSP can provide its own routines.

SEE ALSO **mmuMapLib**, **mmuPhysToVirt( )**

RETURNS the physical address

---

modf( )

NAME **modf( )** – separate a floating-point number into integer and fraction parts (ANSI)

SYNOPSIS

```
double modf
    (
    double   value,           /* value to split */
    double * pIntPart         /* where integer portion is stored */
    )
```

DESCRIPTION This routine stores the integer portion of *value* in *pIntPart* and returns the fractional portion. Both parts are double precision and will have the same sign as *value*.

INCLUDE FILES **math.h**

RETURNS The double-precision fractional portion of *value*.

SEE ALSO **ansiMath**, **frexp( )**, **ldexp( )**

---

moduleCheck( )

NAME **moduleCheck( )** – verify checksums on all modules

SYNOPSIS

```
STATUS moduleCheck
    (
    int options               /* validation options */
    )
```

M

DESCRIPTION

This routine verifies the checksums on the segments of all loaded modules. If any of the checksums are incorrect, a message is printed to the console, and the routine returns **ERROR**.

By default, only the text segment checksum is validated.

Bits in the *options* parameter may be set to control specific checks:

MODCHECK_TEXT
Validate the checksum for the TEXT segment (default).

MODCHECK_DATA
Validate the checksum for the DATA segment.

MODCHECK_BSS
Validate the checksum for the BSS segment.

MODCHECK_NOPRINT
Do not print a message (**moduleCheck( )** still returns **ERROR** on failure.)

See the definitions in **moduleLib.h**

RETURNS

OK, or **ERROR** if the checksum is invalid.

SEE ALSO

moduleLib

moduleCreate( )

NAME

moduleCreate( ) – create and initialize a module

SYNOPSIS

```
MODULE_ID moduleCreate
    (
    char * name,              /* module name */
    int    format,            /* object module format */
    int    flags              /* symFlag as passed to loader (see */
                              /* loadModuleAt()) */
    )
```

DESCRIPTION

This routine creates an object module descriptor.

The arguments specify the name of the object module file, the object module format, and an argument specifying which symbols to add to the symbol table. See the **loadModuleAt( )** description of *symFlag* for possible *flags* values.

Space for the new module is dynamically allocated.

RETURNS

MODULE_ID, or **NULL** if there is an error.

SEE ALSO **moduleLib**, **loadModuleAt( )**

moduleCreateHookAdd( )

NAME **moduleCreateHookAdd( )** – add a routine to be called when a module is added

SYNOPSIS

```
STATUS moduleCreateHookAdd
    (
    FUNCPTR moduleCreateHookRtn /* routine called when module is added */
    )
```

DESCRIPTION This routine adds a specified routine to a list of routines to be called when a module is created. The specified routine should be declared as follows:

```
    void moduleCreateHook
        (
        MODULE_ID  moduleId  /* the module ID */
        )
```

This routine is called after all fields of the module ID have been filled in.

NOTE: Modules do not have information about their object segments when they are created. This information is not available until after the entire load process has finished.

RETURNS **OK** or **ERROR**.

SEE ALSO **moduleLib**, **moduleCreateHookDelete( )**

moduleCreateHookDelete( )

NAME **moduleCreateHookDelete( )** – delete a previously added module create hook routine

SYNOPSIS

```
STATUS moduleCreateHookDelete
    (
    FUNCPTR moduleCreateHookRtn /* routine called when module is added */
    )
```

DESCRIPTION This routine removes a specified routine from the list of routines to be called at each **moduleCreate( )** call.

RETURNS **OK**, or **ERROR** if the routine is not in the table of module create hook routines.

SEE ALSO **moduleLib**, **moduleCreateHookAdd( )**

moduleDelete( )

NAME **moduleDelete( )** – delete module ID information (use **unld( )** to reclaim space)

SYNOPSIS

```
STATUS moduleDelete
    (
    MODULE_ID moduleId        /* module to delete */
    )
```

DESCRIPTION This routine deletes a module descriptor, freeing any space that was allocated for the use of the module ID.

This routine does not free space allocated for the object module itself -- this is done by **unld( )**.

RETURNS **OK** or **ERROR**.

SEE ALSO **moduleLib**

moduleFindByGroup( )

NAME **moduleFindByGroup( )** – find a module by group number

SYNOPSIS

```
MODULE_ID moduleFindByGroup
    (
    int groupNumber           /* group number to find */
    )
```

DESCRIPTION This routine searches for a module with a group number matching *groupNumber*.

RETURNS **MODULE_ID**, or **NULL** if no match is found.

SEE ALSO **moduleLib**

moduleFindByName( )

NAME

moduleFindByName( ) – find a module by name

SYNOPSIS

```
MODULE_ID moduleFindByName
    (
    char * moduleName         /* name of module to find */
    )
```

DESCRIPTION

This routine searches for a module with a name matching *moduleName*.

RETURNS

MODULE_ID, or **NULL** if no match is found.

SEE ALSO

moduleLib

moduleFindByNameAndPath( )

NAME

moduleFindByNameAndPath( ) – find a module by file name and path

SYNOPSIS

```
MODULE_ID moduleFindByNameAndPath
    (
    char * moduleName,        /* file name to find */
    char * pathName           /* path name to find */
    )
```

DESCRIPTION

This routine searches for a module with a name matching *moduleName* and path matching *pathName*.

RETURNS

MODULE_ID, or **NULL** if no match is found.

SEE ALSO

moduleLib

moduleFlagsGet( )

NAME

moduleFlagsGet( ) – get the flags associated with a module ID

SYNOPSIS

```
int moduleFlagsGet
    (
    MODULE_ID moduleId
    )
```

DESCRIPTION

This routine returns the flags associated with a module ID.

RETURNS

The flags associated with the module ID, or **NULL** if the module ID is invalid.

SEE ALSO

moduleLib

moduleIdListGet( )

NAME

moduleIdListGet( ) – get a list of loaded modules

SYNOPSIS

```
int moduleIdListGet
    (
    MODULE_ID * idList,       /* array of module IDs to be filled in */
    int         maxModules    /* max modules idList can accommodate */
    )
```

DESCRIPTION

This routine provides the calling task with a list of all loaded object modules. An unsorted list of module IDs for no more than *maxModules* modules is put into *idList*.

RETURNS

The number of modules put into the ID list, or **ERROR**.

SEE ALSO

moduleLib

M

moduleInfoGet( )

NAME

moduleInfoGet( ) – get information about an object module

SYNOPSIS

```
STATUS moduleInfoGet
    (
    MODULE_ID     moduleId,   /* module to return information about */
    MODULE_INFO * pModuleInfo /* pointer to module info struct */
    )
```

DESCRIPTION

This routine fills in a **MODULE_INFO** structure with information about the specified module.

RETURNS

OK or **ERROR**.

SEE ALSO

moduleLib

moduleNameGet( )

NAME **moduleNameGet( )** – get the name associated with a module ID

SYNOPSIS

```
char * moduleNameGet
    (
    MODULE_ID moduleId
    )
```

DESCRIPTION This routine returns a pointer to the name associated with a module ID.

RETURNS A pointer to the module name, or **NULL** if the module ID is invalid.

SEE ALSO **moduleLib**

moduleSegFirst( )

NAME **moduleSegFirst( )** – find the first segment in a module

SYNOPSIS

```
SEGMENT_ID moduleSegFirst
    (
    MODULE_ID moduleId         /* module to get segment from */
    )
```

DESCRIPTION This routine returns information about the first segment of a module descriptor.

RETURNS A pointer to the segment ID, or **NULL** if the segment list is empty.

SEE ALSO **moduleLib**, **moduleSegGet( )**

moduleSegGet( )

NAME **moduleSegGet( )** – get (delete and return) the first segment from a module

SYNOPSIS

```
SEGMENT_ID moduleSegGet
    (
    MODULE_ID moduleId        /* module to get segment from */
    )
```

DESCRIPTION This routine returns information about the first segment of a module descriptor, and then deletes the segment from the module.

RETURNS A pointer to the segment ID, or **NULL** if the segment list is empty.

SEE ALSO **moduleLib**, **moduleSegFirst( )**

moduleSegNext( )

NAME **moduleSegNext( )** – find the next segment in a module

SYNOPSIS

```
SEGMENT_ID moduleSegNext
    (
    SEGMENT_ID segmentId      /* segment whose successor is to be found */
    )
```

DESCRIPTION This routine returns the segment in the list immediately following *segmentId*.

RETURNS A pointer to the segment ID, or **NULL** if there is no next segment.

SEE ALSO **moduleLib**

moduleShow( )

NAME

moduleShow( ) – show the current status for all the loaded modules

SYNOPSIS

```
STATUS moduleShow
    (
    char * moduleNameOrId,    /* name or ID of the module to show */
    int    options            /* display options */
    )
```

DESCRIPTION

This routine displays a list of the currently loaded modules and some information about where the modules are loaded.

The specific information displayed depends on the format of the object modules. In the case of a.out and ECOFF object modules, **moduleShow( )** displays the start of the text, data, and BSS segments.

If **moduleShow( )** is called with no arguments, a summary list of all loaded modules is displayed. It can also be called with an argument, *moduleNameOrId*, which can be either the name of a loaded module or a module ID. If it is called with either of these, more information about the specified module will be displayed.

RETURNS

OK or **ERROR**.

SEE ALSO

moduleLib, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

mountdInit( )

NAME

mountdInit( ) – initialize the mount daemon

SYNOPSIS

```
STATUS mountdInit
    (
    int     priority,         /* priority of the mount daemon */
    int     stackSize,        /* stack size of the mount daemon */
    FUNCPTR authHook,         /* hook to run to authorize each request */
    int     nExports,         /* maximum number of exported file systems */
    int     options           /* currently unused - set to 0 */
    )
```

DESCRIPTION

This routine spawns a mount daemon if one does not already exist. Defaults for the *priority* and *stackSize* arguments are in the global variables **mountdPriorityDefault** and

mountdStackSizeDefault, and are initially set to **MOUNTD_PRIORITY_DEFAULT** and **MOUNTD_STACKSIZE_DEFAULT** respectively.

Normally, no authorization checking is performed by either mountd or nfsd. To add authorization checking, set *authHook* to point to a routine declared as follows:

```
nfsstat routine
    (
    int                progNum,      /* RPC program number */
    int                versNum,      /* RPC program version number */
    int                procNum,      /* RPC procedure number */
    struct sockaddr_in clientAddr,   /* address of the client */
    MOUNTD_ARGUMENT *  mountdArg     /* argument of the call */
    )
```

The *authHook* callback must return **OK** if the request is authorized, and any defined NFS error code (usually **NFSERR_ACCES**) if not.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call **mountdInit( )** from within the kernel protection domain only, and the function referenced in the *authHook* parameter must reside in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS **OK**, or **ERROR** if the mount daemon could not be correctly initialized.

SEE ALSO **mountLib**

mqPxLibInit( )

NAME **mqPxLibInit( )** – initialize the POSIX message queue library

SYNOPSIS

```
int mqPxLibInit
    (
    int hashSize              /* log2 of number of hash buckets */
    )
```

DESCRIPTION This routine initializes the POSIX message queue facility. If *hashSize* is 0, the default value is taken from **MQ_HASH_SIZE_DEFAULT**.

RETURNS **OK** or **ERROR**.

SEE ALSO **mqPxLib**

M

mqPxShowInit( )

NAME

mqPxShowInit( ) – initialize the POSIX message queue show facility

SYNOPSIS

```
STATUS mqPxShowInit (void)
```

DESCRIPTION

This routine links the POSIX message queue show routine into the VxWorks system. It is called automatically when this show facility is configured into VxWorks using either of the following methods:

– If you use the configuration header files, define

INCLUDE_SHOW_ROUTINES in **config.h**.

– If you use the Tornado project facility, select **INCLUDE_POSIX_MQ_SHOW**.

RETURNS

OK, or **ERROR** if an error occurs installing the file pointer show routine.

SEE ALSO

mqPxShow

mq_close( )

NAME

mq_close( ) – close a message queue (POSIX)

SYNOPSIS

```
int mq_close
    (
    mqd_t mqdes                /* message queue descriptor */
    )
```

DESCRIPTION

This routine is used to indicate that the calling task is finished with the specified message queue *mqdes*. The **mq_close( )** call deallocates any system resources allocated by the system for use by this task for its message queue. The behavior of a task that is blocked on either a **mq_send( )** or **mq_receive( )** is undefined when **mq_close( )** is called. The *mqdes* parameter will no longer be a valid message queue ID.

RETURNS

0 (**OK**) if the message queue is closed successfully, otherwise -1 (**ERROR**).

ERRNO

EBADF

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 created but with modifications made by subsequent calls to **mq_setattr( )**:

mq_flags
  May be modified by **mq_setattr( )**.

The following were set at message queue creation:

mq_maxmsg
  Maximum number of messages.

mq_msgsize
  Maximum message size.

mq_curmsgs
  The number of messages currently in the queue.

RETURNS

0 (**OK**) if message attributes can be determined, otherwise -1 (**ERROR**).

ERRNO

EBADF

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 is able to attach a single notification to each *mqdes* it has unless another task has already attached one.

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, **EBUSY**, **EINVAL**

SEE ALSO

mqPxLib, **mq_open( )**, **mq_send( )**

mq_open( )

NAME

mq_open( ) – open a message queue (POSIX)

SYNOPSIS

```
mqd_t mq_open
    (
    const char * mqName,      /* name of queue to open */
    int          oflags,      /* open flags */
                 ...          /* extra optional parameters */
    )
```

DESCRIPTION

This routine establishes a connection between a named message queue and the calling task. After a call to **mq_open( )**, the task can reference the message queue using the address returned by the call. The message queue remains usable until the queue is closed by a successful call to **mq_close( )**.

The *oflags* argument controls whether the message queue is created or merely accessed by the **mq_open( )** call. 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 remaining flags can be specified in *oflags*:

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 implementation-defined default message queue attributes. 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 and determines whether a **mq_send( )** or **mq_receive( )** will wait for resources or messages that are not currently available, or 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 **EEXIST**, **EINVAL**, **ENOENT**, **ENOSPC**

SEE ALSO **mqPxLib**, **mq_send( )**, **mq_receive( )**, **mq_close( )**, **mq_setattr( )**, **mq_getattr( )**, **mq_unlink( )**

mq_receive( )

NAME **mq_receive( )** – receive a message from a message queue (POSIX)

SYNOPSIS

```
ssize_t mq_receive
    (
    mqd_t  mqdes,              /* message queue descriptor */
    void * pMsg,               /* buffer to receive message */
    size_t msgLen,             /* size of buffer, in bytes */
    int *  pMsgPrio            /* if not NULL, priority of message */
    )
```

DESCRIPTION This routine receives the oldest of the highest priority message from the message queue specified by *mqdes*. If the size of the buffer in bytes, specified by the *msgLen* argument, is less than the **mq_msgsize** attribute of the message queue, **mq_receive( )** will fail and return an error. Otherwise, the selected message is removed from the queue and copied to *pMsg*.

If *pMsgPrio* is not **NULL**, the priority of the selected message will be stored in *pMsgPrio*.

If the message queue is empty and **O_NONBLOCK** is not set in the message queue's description, **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**, **EBADF**, **EMSGSIZE**, **EINTR**

SEE ALSO **mqPxLib**, **mq_send( )**

---

mq_send( )

NAME **mq_send( )** – send a message to a message queue (POSIX)

SYNOPSIS

```
int mq_send
    (
    mqd_t        mqdes,       /* message queue descriptor */
    const void * pMsg,        /* message to send */
    size_t       msgLen,      /* size of message, in bytes */
    int          msgPrio      /* priority of message */
    )
```

DESCRIPTION This routine adds the message *pMsg* to the message queue *mqdes*. The *msgLen* parameter specifies the length of the message in bytes pointed to by *pMsg*. The value of *pMsg* must be less than or equal to the **mq_msgsize** attribute of the message queue, or **mq_send( )** will fail.

If the message queue is not full, **mq_send( )** will behave as if the message is inserted into the message queue at the position indicated by the *msgPrio* argument. A message with a higher numeric value for *msgPrio* is inserted before messages with a lower value. The value of *msgPrio* must be less than or equal to 31.

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. The priority scheduling option is supported in the event that there is more than one task waiting on space becoming available. If the message queue is full and **O_NONBLOCK** is set in the message queue's description, the message is not queued, and **mq_send( )** returns an error.

USE BY INTERRUPT SERVICE ROUTINES

This routine can be called by interrupt service routines as well as by tasks. This is one of the primary means of communication between an interrupt service routine and a task. If **mq_send( )** is called from an interrupt service routine, it will behave as if the

O_NONBLOCK flag were set.

RETURNS 0 (**OK**), otherwise -1 (**ERROR**).

ERRNO **EAGAIN, EBADF, EINTR, EINVAL, EMSGSIZE**

SEE ALSO **mqPxLib**, **mq_receive( )**

mq_setattr( )

NAME **mq_setattr( )** – set message queue attributes (POSIX)

SYNOPSIS

```
int mq_setattr
    (
    mqd_t                    mqdes,      /* message queue descriptor */
    const struct mq_attr * pMqStat,    /* new attributes */
    struct mq_attr *         pOldMqStat /* old attributes */
    )
```

DESCRIPTION This routine sets attributes associated with the specified message queue *mqdes*.

The message queue attributes corresponding to the following members defined in the **mq_attr** structure are set to the specified values upon successful completion of the call:

mq_flags
: The value 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**

SEE ALSO **mqPxLib**, **mq_open( )**, **mq_send( )**, **mq_getattr( )**

mq_unlink( )

NAME **mq_unlink( )** – remove a message queue (POSIX)

SYNOPSIS

```
int mq_unlink
    (
    const char * mqName       /* name of message queue */
    )
```

DESCRIPTION This routine removes the message queue named by the pathname *mqName*. After a successful call to **mq_unlink( )**, a call to **mq_open( )** on the same message queue will fail if the flag **O_CREAT** is not set. If one or more tasks have the message queue open when **mq_unlink( )** is called, removal of the message queue is postponed until all references to the message queue have been closed.

RETURNS 0 (**OK**) if the message queue is unlinked successfully, otherwise -1 (**ERROR**).

ERRNO **ENOENT**

SEE ALSO **mqPxLib**, **mq_close( )**, **mq_open( )**

mRegs( )

NAME **mRegs( )** – modify registers

SYNOPSIS

```
STATUS mRegs
    (
    char * regName,           /* register name, NULL for all */
    int    taskNameOrId       /* task name or task ID, 0 = default task */
    )
```

DESCRIPTION This command modifies the specified register for the specified task. If *taskNameOrId* is omitted or zero, the last task referenced is assumed. If the specified register is not found, it prints out the valid register list and returns **ERROR**. If no register is specified, it sequentially prompts the user for new values for a task's registers. It displays each register and the current contents of that register, in turn. The user can respond in one of several ways:

RETURN
: Do not change this register, but continue, prompting at the next register.

number
    Set this register to *number*.

. (dot)
    Do not change this register, and quit.

EOF
    Do not change this register, and quit.

All numbers are entered and displayed in hexadecimal, except floating-point values, which may be entered in double precision.

RETURNS
OK, or **ERROR** if the task or register does not exist.

SEE ALSO
usrLib, **m( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

mRouteAdd( )

NAME
mRouteAdd( ) – add multiple routes to the same destination

SYNOPSIS
```
STATUS mRouteAdd
    (
    char * pDest,             /* destination addr in internet dot notation */
    char * pGate,             /* gateway address in internet dot notation */
    long   mask,              /* mask for destination */
    int    tos,               /* type of service */
    int    flags              /* route flags */
    )
```

DESCRIPTION
This routine is similar to **routeAdd( )**, except that you can use multiple **mRouteAdd( )** calls to add multiple routes to the same location. Use *pDest* to specify the destination, *pGate* to specify the gateway to that destination, *mask* to specify destination mask, and *tos* to specify the type of service. For *tos*, **netinet/ip.h** defines the following constants as valid values:

IPTOS_LOWDELAY
IPTOS_THROUGHPUT
IPTOS_RELIABILITY
IPTOS_MINCOST

Use *flags* to specify any flags you want to associate with this entry. The valid non-zero values are **RTF_HOST** and **RTF_CLONING** defined in **net/route.h**.

EXAMPLE

To add a route to the 90.0.0.0 network through 91.0.0.3:

```
-> mRouteAdd ("90.0.0.0", "91.0.0.3", 0xffffff00, 0, 0);
```

Using **mRouteAdd( )**, you could create multiple routes to the same destination. VxWorks would distinguish among these routes based on factors such as the netmask or the type of service. Thus, it is perfectly legal to say:

```
-> mRouteAdd ("90.0.0.0", "91.0.0.3", 0xffffff00, 0, 0);
-> mRouteAdd ("90.0.0.0", "91.0.0.254", 0xffff0000, 0, 0);
```

This adds two routes to the same network, "90.0.0.0", that go by two different gateways. The differentiating factor is the netmask.

This routine adds a route of type **M2_ipRouteProto_other**, which is a static route. This route will not be modified or deleted until a call to **mRouteDelete( )** removes it.

RETURNS

OK or **ERROR**.

SEE ALSO

routeLib, **mRouteEntryAdd( )**, **mRouteDelete( )**, **routeAdd( )**

---

mRouteDelete( )

M

NAME

mRouteDelete( ) – delete a route from the routing table

SYNOPSIS

```
STATUS mRouteDelete
    (
    char * pDest,              /* destination address */
    long   mask,               /* mask for destination */
    int    tos,                /* type of service */
    int    flags               /* either 0 or RTF_HOST */
    )
```

DESCRIPTION

This routine deletes a routing table entry as specified by the destination, *pDest*, the destination mask, *mask*, and type of service, *tos*. The *tos* values are as defined in the reference entry for **mRouteAdd( )**.

EXAMPLE

Consider the case of a route added in the following manner:

```
-> mRouteAdd ("90.0.0.0", "91.0.0.3", 0xffffff00, 0, 0);
```

To delete a route that was added in the above manner, call **mRouteDelete( )** as follows:

```
-> mRouteDelete("90.0.0.0", 0xffffff00, 0);
```

If the netmask and or type of service do not match, the route is not deleted.

The value of *flags* should be **RTF_HOST** for host routes, **RTF_CLONING** for routes which need to be cloned, and 0 in all other cases.

RETURNS **OK** or **ERROR**.

SEE ALSO **routeLib**, **mRouteAdd( )**

mRouteEntryAdd( )

NAME **mRouteEntryAdd( )** – add a protocol-specific route to the routing table

SYNOPSIS

```
STATUS mRouteEntryAdd
    (
    long destIp,            /* destination address, network order */
    long gateIp,            /* gateway address, network order */
    long mask,              /* mask for destination, network order */
    int  tos,               /* type of service */
    int  flags,             /* route flags */
    int  proto              /* routing protocol */
    )
```

DESCRIPTION For a single destination *destIp*, this routine can add additional routes *gateIp* to the routing table. The different routes are distinguished by the gateway, a destination mask *mask*, the type of service *tos* and associated flag values *flags*. Valid values for *flags* are 0, **RTF_HOST**, **RTF_CLONING** (defined in **net/route.h**). The *proto* parameter identifies the protocol that generated this route. Values for *proto* may be found in **m2Lib.h**. The *tos* parameter takes one of following values (defined in **netinet/ip.h**):

IPTOS_LOWDELAY
IPTOS_THROUGHPUT
IPTOS_RELIABILITY
IPTOS_MINCOST

RETURNS **OK** or **ERROR**.

SEE ALSO **routeLib**, **m2Lib.h**, **mRouteAdd( )**, **mRouteDelete( )**

mRouteEntryDelete( )

NAME **mRouteEntryDelete( )** – delete route from the routing table

SYNOPSIS

```
STATUS mRouteEntryDelete
    (
    long destIp,             /* destination address, network order */
    long gateIp,             /* gateway address, network order */
    long mask,               /* mask for destination, network order */
    int  tos,                /* type of service */
    int  flags,              /* route flags */
    int  proto               /* routing protocol */
    )
```

DESCRIPTION This routine deletes a protocol-specific route from the routing table. Specify the route using a destination *pDest*, a gateway *pGate*, a destination mask *mask*, the type of service *tos*, a *flags* value, and a *proto* value that identifies the routing protocol that added the route. The valid values for *flags* are 0 and **RTF_HOST** (defined in **net/route.h**). Values for *proto* may be found in **m2Lib.h** and *tos* is one of the following values defined in **netinet/ip.h**:

IPTOS_LOWDELA
IPTOS_THROUGHPU
IPTOS_RELIABILIT
IPTOS_MINCOST

An existing route is deleted only if it is owned by the protocol specified by *proto*.

RETURNS **OK** or **ERROR**.

SEE ALSO **routeLib**

mRouteShow( )

NAME **mRouteShow( )** – display all IP routes (verbose information)

SYNOPSIS

```
void mRouteShow (void)
```

DESCRIPTION This routine displays the list of destinations in the routing table along with the next-hop gateway and associated interface. It also displays the netmask for a route (to handle classless routes which use arbitrary values for that field) and the value which indicates the route's creator, as well as any type-of-service information.

M

When multiple routes exist to the same destination with the same netmask, the IP forwarding process only uses the first route entry with the lowest administrative weight. The remaining entries (listed as additional routes) use the same address and netmask. One of those entries will replace the primary route if it is deleted.

Some configuration is required when this routine is to be used remotely over the network, *e.g.*, through a **telnet** session or through the host shell using **WDB_COMM_NETWORK**. If more than 5 routes are expected in the table the parameter **RT_BUFFERED_DISPLAY** should be set to **TRUE** to prevent a possible deadlock. This requires a buffer whose size can be set with **RT_DISPLAY_MEMORY**. It will limit the number of routes that can be displayed (each route requires approx. 90 bytes).

RETURNS N/A

SEE ALSO **netShow**

msgQCreate( )

NAME **msgQCreate( )** – create and initialize a message queue

SYNOPSIS

```
MSG_Q_ID msgQCreate
    (
    int maxMsgs,              /* max messages that can be queued */
    int maxMsgLength,         /* max bytes in a message */
    int options               /* message queue options */
    )
```

DESCRIPTION This routine creates a message queue capable of holding up to *maxMsgs* messages, each up to *maxMsgLength* bytes long. The routine returns a message queue ID used to identify the created message queue in all subsequent calls to routines in this library. The queue can be created with the following options:

MSG_Q_FIFO (0x00)
queue pended tasks in FIFO order.

MSG_Q_PRIORITY (0x01)
queue pended tasks in priority order.

MSG_Q_EVENTSEND_ERR_NOTIFY (0x02)
When a message is sent, if a task is registered for events and the actual sending of events fails, a value of **ERROR** is returned and the **errno** is set accordingly. This option is off by default.

RETURNS **MSG_Q_ID**, or **NULL** if error.

ERRNO **S_memLib_NOT_ENOUGH_MEMORY**, **S_intLib_NOT_ISR_CALLABLE**

SEE ALSO **msgQLib**, **msgQSmLib**

msgQDelete( )

NAME

msgQDelete( ) – delete a message queue

SYNOPSIS

```
STATUS msgQDelete
    (
    MSG_Q_ID msgQId           /* message queue to delete */
    )
```

DESCRIPTION

This routine deletes a message queue. All tasks pending on either **msgQSend( )**, **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
Message queue ID is invalid

S_intLib_NOT_ISR_CALLABLE
Routine cannot be called from ISR

S_distLib_NO_OBJECT_DESTROY
Deleting a distributed message queue is not permitted

S_smObjLib_NO_OBJECT_DESTROY
Deleting a shared message queue is not permitted

SEE ALSO

msgQLib, **msgQSmLib**

M

msgQDistCreate( )

NAME

msgQDistCreate( ) – create a distributed message queue (VxFusion Opt.)

SYNOPSIS

```
MSG_Q_ID msgQDistCreate
    (
    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 distributed message queue capable of holding up to *maxMsgs* messages, each up to *maxMsgLength* bytes long. This routine returns a message queue ID used to identify the created message queue. The queue can be created with the following options:

MSG_Q_FIFO (0x00)
The queue pends tasks in FIFO order.

MSG_Q_PRIORITY (0x01)
The queue pends tasks in priority order. Remote tasks share the same priority level.

The global message queue identifier returned can be used directly by generic message queue handling routines in **msgQLib**, such as, **msgQSend( )**, **msgQReceive( )**, and **msgQNumMsgs( )**.

AVAILABILITY This routine is distributed as a component of the unbundled distributed message queues option, VxFusion.

RETURNS **MSG_Q_ID**, or **NULL** if there is an error.

ERRNO **S_memLib_NOT_ENOUGH_MEMORY**
If the routine is unable to allocate memory for message queues and message buffers.

S_intLib_NOT_ISR_CALLABLE
If the routine is called from an interrupt service routine.

S_msgQLib_INVALID_QUEUE_TYPE
If the type of queue is invalid.

S_msgQDistLib_INVALID_MSG_LENGTH
If the message is too long for the VxFusion network layer.

SEE ALSO **msgQDistLib**, **msgQLib**

msgQDistGrpAdd( )

NAME

msgQDistGrpAdd( ) – add a distributed message queue to a group (VxFusion Opt.)

SYNOPSIS

```
MSG_Q_ID msgQDistGrpAdd
    (
    char *       distGrpName, /* new or existing group name */
    MSG_Q_ID     msgQId,      /* message queue to add to the group */
    DIST_GRP_OPT options      /* group message queue options - UNUSED */
    )
```

DESCRIPTION

This routine adds the queue identified by the argument *msgQId* to a group with the ASCII name specified by the argument *distGrpName*.

Multicasting is based on distributed message queue groups. If the group does not exist, one is created. Any number of message queues from different nodes can be bound to a single group. In addition, a message queue can be added into any number of groups; **msgQDistGrpAdd( )** must be called for each group of which the message queue is to be a member.

The *options* parameter is presently unused and must be set to 0.

This routine returns a message queue ID, **MSG_Q_ID**, that can be used directly by **msgQDistSend( )** or by the generic **msgQSend( )** routine. Do not call the **msgQReceive( )** or **msgQNumMsgs( )** routines or their distributed counterparts, **msgQDistReceive( )** and **msgQDistNumMsgs( )**, with a group message queue ID.

As with **msgQDistCreate( )**, use **distNameAdd( )** to add the group message queue ID returned by this routine to the distributed name database so that the ID can be used by tasks on other nodes.

AVAILABILITY

This routine is distributed as a component of the unbundled distributed message queues option, VxFusion.

RETURNS

MSG_Q_ID, or **NULL** if there is an error.

ERRNO

S_msgQDistGrpLib_NAME_TOO_LONG
The name of the group is too long.

S_msgQDistGrpLib_INVALID_OPTION
The *options* parameter is invalid.

S_msgQDistGrpLib_DATABASE_FULL
The group database is full.

S_distLib_OBJ_ID_ERROR
The *msgQId* parameter is not a distributed message queue.

SEE ALSO

msgQDistGrpLib, **msgQLib**, **msgQDistLib**, **distNameLib**

msgQDistGrpDelete( )

NAME **msgQDistGrpDelete( )** – delete a distributed message queue from a group (VxFusion Opt.)

SYNOPSIS

```
STATUS msgQDistGrpDelete
    (
    char *   distGrpName,      /* group containing the queue to be deleted */
    MSG_Q_ID msgQId            /* ID of the message queue to delete */
    )
```

DESCRIPTION This routine deletes a distributed message queue from a group.

NOTE: For this release, it is not possible to remove a distributed message queue from a group.

AVAILABILITY This routine is distributed as a component of the unbundled distributed message queues option, VxFusion.

RETURNS **ERROR**, always.

ERRNO **S_distLib_NO_OBJECT_DESTROY**

SEE ALSO **msgQDistGrpLib**

msgQDistGrpShow( )

NAME **msgQDistGrpShow( )** – display all or one group with its members (VxFusion Opt.)

SYNOPSIS

```
STATUS msgQDistGrpShow
    (
    char * distGrpName     /* name of the group to display or NULL for all */
    )
```

DESCRIPTION This routine displays either all distributed message queue groups or a specified group in the group database. For each group displayed on the node, it lists only members added (using **msgQDistGrpAdd( )**) from the node executing the **msgQDistGrpShow( )** call.

If *distGrpName* is **NULL**, all groups and their locally added members are displayed. Otherwise, only the group specified by *distGrpName* and its locally added members are displayed.

NOTE: The concept of "locally added" is an important one. All nodes in the system can add groups to a message queue group. However, only those message queues (including remote distributed message queues) that were added to the group from the local node are displayed by this routine.

EXAMPLE

```
-> msgQDistGrpShow(0)
NAME OF GROUP         GROUP ID   STATE  MEMBER ID TYPE OF MEMBER
------------------- ---------- ------- ---------- ---------------------------
grp1                  0x3ff9e3  global   0x3ff98b distributed msg queue
                                         0x3ff9fb distributed msg queue
grp2                  0x3ff933  global   0x3ff89b distributed msg queue
                                         0x3ff8db distributed msg queue
                                         0x3ff94b distributed msg queue
```

AVAILABILITY

This routine is distributed as a component of the unbundled distributed message queues option, VxFusion.

RETURNS

OK, unless name not found.

ERRNO

S_msgQDistGrpLib_NO_MATCH
The group name was not found in the database.

SEE ALSO

msgQDistGrpShow

msgQDistNumMsgs( )

NAME

msgQDistNumMsgs( ) – get the number of messages in a distributed message queue (VxFusion Opt.)

SYNOPSIS

```
int msgQDistNumMsgs
    (
    MSG_Q_ID msgQId,          /* message queue to examine */
    int      overallTimeout   /* ticks to wait overall */
    )
```

DESCRIPTION

This routine returns the number of messages currently queued to a specified distributed message queue.

NOTE: When **msgQDistNumMsgs( )** is called through **msgQNumMsgs( )**, *overallTimeout* is set to **WAIT_FOREVER**. You cannot set *overallTimeout* to **NO_WAIT** (0) because the process of sending a message from the local node to the remote node always takes a finite amount of time.

AVAILABILITY This routine is distributed as a component of the unbundled distributed message queues option, VxFusion.

RETURNS The number of messages queued, or **ERROR** if the operation fails.

ERRNO

S_distLib_OBJ_ID_ERROR
The argument *msgQId* is invalid.

S_distLib_UNREACHABLE
Could not establish communications with the remote node.

S_msgQDistLib_INVALID_TIMEOUT
The argument *overallTimeout* is **NO_WAIT**.

SEE ALSO **msgQDistLib**, **msgQLib**

msgQDistReceive( )

NAME **msgQDistReceive( )** – receive a message from a distributed message queue (VxFusion Opt.)

SYNOPSIS

```
int msgQDistReceive
    (
    MSG_Q_ID msgQId,          /* message queue from which to receive */
    char *   buffer,          /* buffer to receive message */
    UINT     maxNBytes,       /* length of buffer */
    int      msgQTimeout,     /* ticks to wait at the message queue */
    int      overallTimeout   /* ticks to wait overall */
    )
```

DESCRIPTION This routine receives a message from the distributed message queue specified by *msgQId*. The received message is copied into the specified buffer, *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 argument *msgQTimeout* specifies the time in ticks to wait for the queuing of the message. The argument *overallTimeout* specifies the time in ticks to wait for both the sending and queuing of the message. While it is an error to set *overallTimeout* to **NO_WAIT** (0), **WAIT_FOREVER** (-1) is allowed for both *msgQTimeout* and *overallTimeout*.

Calling **msgQDistReceive( )** on a distributed message group returns an error.

NOTE: When **msgQDistReceive( )** is called through **msgQReceive( )**, *msgQTimeout* is set to *timeout* and *overallTimeout* to **WAIT_FOREVER**.

AVAILABILITY This routine is distributed as a component of the unbundled distributed message queues option, VxFusion.

RETURNS The number of bytes copied to *buffer*, or **ERROR**.

ERRNO

S_distLib_OBJ_ID_ERROR
The argument *msgQId* is invalid.

S_distLib_UNREACHABLE
Could not establish communications with the remote node.

S_msgQLib_INVALID_MSG_LENGTH
The argument *maxNBytes* is less than 0.

S_msgQDistLib_INVALID_TIMEOUT
The argument *overallTimeout* is **NO_WAIT**.

S_msgQDistLib_RMT_MEMORY_SHORTAGE
There is not enough memory on the remote node.

S_objLib_OBJ_UNAVAILABLE
The argument *msgQTimeout* is set to **NO_WAIT**, and no messages are available.

S_objLib_OBJ_TIMEOUT
No messages were received in *msgQTimeout* ticks.

S_msgQDistLib_OVERALL_TIMEOUT
There was no response from the remote side in *overallTimeout* ticks.

SEE ALSO **msgQDistLib**, **msgQLib**

msgQDistSend( )

NAME **msgQDistSend( )** – send a message to a distributed message queue (VxFusion Opt.)

SYNOPSIS

```
STATUS msgQDistSend
    (
    MSG_Q_ID msgQId,          /* message queue on which to send */
    char *   buffer,          /* message to send */
    UINT     nBytes,          /* length of message */
    int      msgQTimeout,     /* ticks to wait at message queue */
    int      overallTimeout,  /* ticks to wait overall */
    int      priority         /* priority */
    )
```

DESCRIPTION

This routine sends the message specified by *buffer* of length *nBytes* to the distributed message queue or group specified by *msgQId*.

The argument *msgQTimeout* specifies the time in ticks to wait for the queuing of the message. The argument *overallTimeout* specifies the time in ticks to wait for both the sending and queuing of the message. While it is an error to set *overallTimeout* to **NO_WAIT** (0), **WAIT_FOREVER** (-1) is allowed for both *msgQTimeout* and *overallTimeout*.

The *priority* parameter specifies the priority of the message being sent. It ranges between **DIST_MSG_PRI_0** (highest priority) and **DIST_MSG_PRI_7** (lowest priority). A priority of **MSG_PRI_URGENT** is mapped to **DIST_MSG_PRI_0**; **MSG_PRI_NORMAL** is mapped to **DIST_MSG_PRI_4**. Messages sent with high priorities (**DIST_MSG_PRI_0** to **DIST_MSG_PRI_3**) are put to the head of the list of queued messages. Lower priority messages (**DIST_MSG_PRI_4** to **DIST_MSG_PRI_7**) are placed at the queue's tail.

NOTE: When **msgQDistSend( )** is called through **msgQSend( )**, *msgQTimeout* is set to *timeout* and *overallTimeout* to **WAIT_FOREVER**.

AVAILABILITY

This routine is distributed as a component of the unbundled distributed message queues option, VxFusion.

RETURNS

OK, or **ERROR** if the operation fails.

ERRNO

S_distLib_OBJ_ID_ERROR
The argument *msgQId* is invalid.

S_distLib_UNREACHABLE
Could not establish communications with the remote node.

S_msgQDistLib_INVALID_PRIORITY
The argument *priority* is invalid.

S_msgQDistLib_INVALID_TIMEOUT
The argument *overallTimeout* is **NO_WAIT**.

S_msgQDistLib_RMT_MEMORY_SHORTAGE
There is not enough memory on the remote node.

S_objLib_OBJ_UNAVAILABLE
The argument *msgQTimeout* is set to **NO_WAIT**, and the queue is full.

S_objLib_OBJ_TIMEOUT
The queue is full for *msgQTimeout* ticks.

S_msgQLib_INVALID_MSG_LENGTH
The argument *nBytes* is larger than the *maxMsgLength* set for the message queue.

S_msgQDistLib_OVERALL_TIMEOUT
There was no response from the remote side in *overallTimeout* ticks.

SEE ALSO

msgQDistLib, **msgQLib**

msgQDistShowInit( )

NAME **msgQDistShowInit( )** – initialize the distributed message queue show package (VxFusion Opt.)

SYNOPSIS

```
void msgQDistShowInit (void)
```

DESCRIPTION This routine initializes the distributed message queue show package.

NOTE: This routine is called automatically when a target boots using a VxWorks image with VxFusion installed and show routines enabled.

AVAILABILITY This routine is distributed as a component of the unbundled distributed message queues option, VxFusion.

RETURNS N/A

SEE ALSO **msgQDistShow**

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. When a message becomes available but not wanted in that particular message queue, the events specified will be sent to the task registered by this function. A task can overwrite its own registration without first invoking **msgQEvStop( )** or specifying the **ALLOW_OVERWRITE** option.

The *options* parameter is used for 3 user options:

EVENTS_SEND_ONCE (0x1)
: tells the message queue to send the events one time only. Specify if the events are to be sent only once or every time a message arrives until **msgQEvStop( )** is called.

EVENTS_ALLOW_OVERWRITE (0x2)
: allows subsequent registrations to overwrite the current one. Specify if another task can register itself while the current task is still registered. If so, the current task registration is overwritten without any warning.

EVENTS_SEND_IF_FREE (0x4)
: tells the registration process to send events if a message is present on the message queue. Specify if events are to be sent right away in the case a message is waiting to be picked up.

If none of those three options is to be used, then the option

EVENTS_OPTIONS_NONE (0x0)
: has to be passed to the *options* parameter.

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
: Routine has been called from interrupt level.

S_eventLib_EVENTSEND_FAILED
: User chose to send events right away and that operation failed.

S_eventLib_ZERO_EVENTS
: User passed in a value of zero to the *events* parameter.

SEE ALSO

msgQEvLib, **eventLib**, **msgQLib**, **msgQEvStop( )**

msgQEvStop( )

NAME

msgQEvStop( ) – stop 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. It thus allows another task to register itself for event notification 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
Routine has been called from interrupt level.

S_eventLib_TASK_NOT_REGISTERED
Routine has not been called by registered task.

SEE ALSO **msgQEvLib**, **eventLib**, **msgQLib**, **msgQEvStart( )**

msgQInfoGet( )

M

NAME **msgQInfoGet( )** – get information about a message queue

SYNOPSIS

```
STATUS msgQInfoGet
    (
    MSG_Q_ID     msgQId,       /* message queue to query */
    MSG_Q_INFO * pInfo         /* where to return msg info */
    )
```

DESCRIPTION This routine gets information about the state and contents of a message queue. The parameter *pInfo* is a pointer to a structure of type **MSG_Q_INFO** defined in **msgQLib.h** as follows:

```
 typedef struct              /* MSG_Q_INFO */
    {
    int     numMsgs;         /* OUT: number of messages queued          */
    int     numTasks;        /* OUT: number of tasks waiting on msg q    */
    int     sendTimeouts;    /* OUT: count of send timeouts              */
    int     recvTimeouts;    /* OUT: count of receive timeouts           */
    int     options;         /* OUT: options with which msg q was created */
    int     maxMsgs;         /* OUT: max messages that can be queued     */
    int     maxMsgLength;    /* OUT: max byte length of each message     */
    int     taskIdListMax;   /* IN: max tasks to fill in taskIdList      */
    int *   taskIdList;      /* PTR: array of task IDs waiting on msg q  */
    int     msgListMax;      /* IN: max msgs to fill in msg lists        */
    char ** msgPtrList;      /* PTR: array of msg ptrs queued to msg q   */
```

```
    int *   msgLenList;       /* PTR: array of lengths of msgs             */
    } MSG_Q_INFO;
```

If a message queue is empty, there may be tasks blocked on receiving. If a message queue is full, there may be tasks blocked on sending. This can be determined as follows:

– If *numMsgs* is 0, then *numTasks* indicates the number of tasks blocked

on receiving.

– If *numMsgs* is equal to *maxMsgs*, then *numTasks* is the number of

tasks blocked on sending.

– If *numMsgs* is greater than 0 but less than *maxMsgs*, then *numTasks*

will be 0.

A list of pointers to the messages queued and their lengths can be obtained by setting *msgPtrList* and *msgLenList* to the addresses of arrays to receive the respective lists, and setting *msgListMax* to the maximum number of elements in those arrays. If either list pointer is **NULL**, no data will be returned for that array.

No more than *msgListMax* message pointers and lengths are returned, although *numMsgs* will always be returned with the actual number of messages queued.

For example, if the caller supplies a *msgPtrList* and *msgLenList*with room for 10 messages and sets *msgListMax* to 10, but there are 20 messages queued, then the pointers and lengths of the first 10 messages in the queue are returned in *msgPtrList* and *msgLenList*, but *numMsgs* will be returned with the value 20.

A list of the task IDs of tasks blocked on the message queue can be obtained by setting *taskIdList* to the address of an array to receive the list, and setting *taskIdListMax* to the maximum number of elements in that array. If *taskIdList* is **NULL**, then no task IDs are returned. No more than *taskIdListMax* task IDs are returned, although *numTasks* will always be returned with the actual number of tasks blocked.

For example, if the caller supplies a *taskIdList* with room for 10 task IDs and sets *taskIdListMax* to 10, but there are 20 tasks blocked on the message queue, then the IDs of the first 10 tasks in the blocked queue will be returned in *taskIdList*, but *numTasks* will be returned with the value 20.

Note that the tasks returned in *taskIdList* may be blocked for either send or receive. As noted above this can be determined by examining *numMsgs*.

The variables *sendTimeouts* and *recvTimeouts* are the counts of the number of times **msgQSend( )** and **msgQReceive( )** respectively returned with a timeout.

The variables *options*, *maxMsgs*, and *maxMsgLength* are the parameters with which the message queue was created.

WARNING: The information returned by this routine is not static and may be obsolete by the time it is examined. In particular, the lists of task IDs and/or message pointers may no longer be valid. However, the information is obtained atomically, thus it will be an accurate snapshot of the state of the message queue at the time of the call. This information is generally used for debugging purposes only.

WARNING: The current implementation of this routine locks out interrupts while obtaining the information. This can compromise the overall interrupt latency of the system. Generally this routine is used for debugging purposes only.

RETURNS **OK** or **ERROR**.

ERRNO **S_distLib_NOT_INITIALIZED**, **S_smObjLib_NOT_INITIALIZED**, **S_objLib_OBJ_ID_ERROR**

SEE ALSO **msgQShow**

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_distLib_NOT_INITIALIZED**, **S_smObjLib_NOT_INITIALIZED**, **S_objLib_OBJ_ID_ERROR**

SEE ALSO **msgQLib**, **msgQSmLib**

msgQReceive( )

NAME

msgQReceive( ) – receive a message from a message queue

SYNOPSIS

```
int msgQReceive
    (
    MSG_Q_ID msgQId,           /* message queue from which to receive */
    char *   buffer,           /* buffer to receive message */
    UINT     maxNBytes,        /* length of buffer */
    int      timeout           /* ticks to wait */
    )
```

DESCRIPTION

This routine receives a message from the message queue *msgQId*. The received message is copied into the specified *buffer*, which is *maxNBytes* in length. If the message is longer than *maxNBytes*, the remainder of the message is discarded (no error indication is returned).

The *timeout* parameter specifies the number of ticks to wait for a message to be sent to the queue, if no message is available when **msgQReceive( )** is called. The *timeout* parameter can also have the following special values:

NO_WAIT (0)
return immediately, whether a message has been received or not.

WAIT_FOREVER (-1)
never time out.

WARNING: This routine must not be called by interrupt service routines.

RETURNS

The number of bytes copied to *buffer*, or **ERROR**.

ERRNO

S_distLib_NOT_INITIALIZED, S_smObjLib_NOT_INITIALIZED, S_objLib_OBJ_ID_ERROR, S_objLib_OBJ_DELETED, S_objLib_OBJ_UNAVAILABLE, S_objLib_OBJ_TIMEOUT, S_msgQLib_INVALID_MSG_LENGTH, S_intLib_NOT_ISR_CALLABLE

SEE ALSO

msgQLib, **msgQSmLib**

msgQSend( )

NAME

msgQSend( ) – send a message to a message queue

SYNOPSIS

```
STATUS msgQSend
    (
    MSG_Q_ID msgQId,          /* message queue on which to send */
    char *   buffer,          /* message to send */
    UINT     nBytes,          /* length of message */
    int      timeout,         /* ticks to wait */
    int      priority         /* MSG_PRI_NORMAL or MSG_PRI_URGENT */
    )
```

DESCRIPTION

This routine sends the message in *buffer* of length *nBytes* to the message queue *msgQId*. If any tasks are already waiting to receive messages on the queue, the message is immediately delivered to the first waiting task. If no task is waiting to receive messages, the message is saved in the message queue and if a task has previously registered to receive events from the message queue, these events are sent in the context of this call. This may result in the unpending of the task waiting for the events. If the message queue fails to send events and if it was created using the **MSG_Q_EVENTSEND_ERR_NOTIFY** option, **ERROR** is returned even though the send operation was successful.

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.

USE BY INTERRUPT SERVICE ROUTINES

This routine can be called by interrupt service routines as well as by tasks. This is one of the primary means of communication between an interrupt service routine and a task. When called from an interrupt service routine, *timeout* must be **NO_WAIT**.

RETURNS

OK on success or **ERROR** otherwise.

ERRNO

S_distLib_NOT_INITIALIZED
Distributed objects message queue library (VxFusion) not initialized.

S_smObjLib_NOT_INITIALIZED
Shared memory message queue library (VxMP Opt.) not initialized.

S_objLib_OBJ_ID_ERROR
Invalid message queue ID.

S_objLib_OBJ_DELETED
Message queue deleted while calling task was pended.

S_objLib_OBJ_UNAVAILABLE
No free buffer space when **NO_WAIT** timeout specified.

S_objLib_OBJ_TIMEOUT
Timeout occurred while waiting for buffer space.

S_msgQLib_INVALID_MSG_LENGTH
Message length exceeds limit.

S_msgQLib_NON_ZERO_TIMEOUT_AT_INT_LEVEL
Called from ISR with non-zero timeout.

S_eventLib_EVENTSEND_FAILED
Message queue failed to send events to registered task. This **errno** value can only exist if the message queue was created with the **MSG_Q_EVENTSEND_ERR_NOTIFY** option.

SEE ALSO

msgQLib, **msgQSmLib**, **msgQEvStart( )**

msgQShow( )

NAME

msgQShow( ) – show information about a message queue

SYNOPSIS

```
STATUS msgQShow
    (
    MSG_Q_ID msgQId,          /* message queue to display */
    int      level            /* 0 = summary, 1 = details */
    )
```

DESCRIPTION

This routine displays the state and optionally the contents of a message queue.

A summary of the state of the message queue is displayed as follows:

```
Message Queue Id    : 0x3f8c20
Task Queuing        : FIFO
Message Byte Len    : 150
Messages Max        : 50
```

```
Messages Queued      : 0
Receivers Blocked    : 1
Send timeouts        : 0
Receive timeouts     : 0
Options              : 0x1       MSG_Q_FIFO
VxWorks Events
--------------
Registered Task      : 0x3f5c70 (t1)
Event(s) to Send     : 0x1
Options              : 0x7       EVENTS_SEND_ONCE
                                 EVENTS_ALLOW_OVERWRITE
                                 EVENTS_SEND_IF_FREE
```

If *level* is 1, then more detailed information will be displayed. If messages are queued, they will be displayed as follows:

```
Messages queued:
  #    address length value
  1 0x123eb204    4   0x00000001 0x12345678
```

If tasks are blocked on the queue, they will be displayed as follows:

```
Receivers blocked:
   NAME      TID    PRI DELAY
---------- -------- --- -----
tExcTask     3fd678   0   21
```

RETURNS **OK** or **ERROR**.

ERRNO **S_distLib_NOT_INITIALIZED**, **S_smObjLib_NOT_INITIALIZED**

SEE ALSO **msgQShow**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

msgQShowInit( )

NAME **msgQShowInit( )** – initialize the message queue show facility

SYNOPSIS

```
void msgQShowInit (void)
```

DESCRIPTION This routine links the message queue show facility into the VxWorks system. It is called automatically when the message queue show facility is configured into VxWorks using either of the following methods:

– If you use the configuration header files, define **INCLUDE_SHOW_ROUTINES** in **config.h**.

– If you use the Tornado project facility, select **INCLUDE_MSG_Q_SHOW**.

M

RETURNS N/A

SEE ALSO **msgQShow**

msgQSmCreate( )

NAME **msgQSmCreate( )** – create and initialize a shared memory message queue (VxMP Opt.)

SYNOPSIS

```
MSG_Q_ID msgQSmCreate
    (
    int maxMsgs,              /* max messages that can be queued */
    int maxMsgLength,         /* max bytes in a message */
    int options               /* message queue options */
    )
```

DESCRIPTION This routine creates a shared memory message queue capable of holding up to *maxMsgs* messages, each up to *maxMsgLength* bytes long. It returns a message queue ID used to identify the created message queue. The queue can only be created with the option **MSG_Q_FIFO** (0), thus queuing pended tasks in FIFO order.

The global message queue identifier returned can be used directly by generic message queue handling routines in **msgQLib** -- **msgQSend( )**, **msgQReceive( )**, and **msgQNumMsgs( )** -- and by the show routines **show( )** and **msgQShow( )**.

If there is insufficient memory to store the message queue structure in the shared memory message queue partition or if the shared memory system pool cannot handle the requested message queue size, shared memory message queue creation will fail with **errno** set to **S_memLib_NOT_ENOUGH_MEMORY**. This problem can be solved by incrementing the value of **SM_OBJ_MAX_MSG_Q** and/or the shared memory objects dedicated memory size **SM_OBJ_MEM_SIZE**.

Before this routine can be called, the shared memory objects facility must be initialized (see **msgQSmLib**).

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS **MSG_Q_ID**, or **NULL** if error.

ERRNO **S_memLib_NOT_ENOUGH_MEMORY**, **S_intLib_NOT_ISR_CALLABLE**, **S_msgQLib_INVALID_QUEUE_TYPE**, **S_smObjLib_LOCK_TIMEOUT**

SEE ALSO **msgQSmLib**, **smObjLib**, **msgQLib**, **msgQShow**

munlock( )

NAME

munlock( ) – unlock specified pages (POSIX)

SYNOPSIS

```
int munlock
    (
    const void * addr,
    size_t       len
    )
```

DESCRIPTION

This routine unlocks specified pages from being memory resident.

RETURNS

0 (**OK**) always.

ERRNO

N/A

SEE ALSO

mmanPxLib

M

munlockall( )

NAME

munlockall( ) – unlock all pages used by a process (POSIX)

SYNOPSIS

```
int munlockall (void)
```

DESCRIPTION

This routine unlocks all pages used by a process from being memory resident.

RETURNS

0 (**OK**) always.

ERRNO

N/A

SEE ALSO

mmanPxLib

muxAddressForm( )

NAME

muxAddressForm( ) – form a frame with a link-layer address

SYNOPSIS

```
M_BLK_ID muxAddressForm
    (
    void *   pCookie,         /* protocol/device binding from muxBind() */
    M_BLK_ID pMblk,           /* structure to contain packet */
    M_BLK_ID pSrcAddr,        /* structure containing source address */
    M_BLK_ID pDstAddr         /* structure containing destination address */
    )
```

DESCRIPTION

Use this routine to create a frame with an appropriate link-layer address. As input, this function expects the source address, the destination address, and the data you want to include in the frame. When control returns from the **muxAddressForm( )** call, the *pMblk* parameter references a frame ready for transmission. Internally, **muxAddressForm( )** either prepended the link-layer header to the data buffer supplied in *pMblk* (if there was enough room) or it allocated a new **mBlk-clBlk**-cluster and prepended the new **mBlk** to the **mBlk** chain supplied in *pMblk*.

NOTE: You should set the **pDstAddr.mBlkHdr.reserved** field to the network service type.

pCookie
: Expects the cookie returned from the **muxBind( )**. This cookie indicates the device to which the MUX has bound this protocol.

pMblk
: Expects a pointer to the **mBlk** structure that contains the packet.

pSrcAddr
: Expects a pointer to the **mBlk** that contains the source address.

pDstAddr
: Expects a pointer to the **mBlk** that contains the destination address.

NOTE: This routine is used only with ENDs, and is not needed for NPT drivers.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call **muxAddressForm( )** from within the kernel protection domain only, and the data referenced in the *pCookie* parameter must reside in the kernel protection domain. In addition, the returned **M_BLK_ID** is valid in the kernel protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

M_BLK_ID or **NULL**.

ERRNO S_muxLib_NO_DEVICE

SEE ALSO **muxLib**

muxAddrResFuncAdd( )

NAME **muxAddrResFuncAdd( )** – replace the default address resolution function

SYNOPSIS

```
STATUS muxAddrResFuncAdd
    (
    long    ifType,        /* Media interface type, typically from m2Lib.h */
    long    protocol,      /* Service type, for instance from RFC 1700 */
    FUNCPTR addrResFunc    /* Function to call. */
    )
```

DESCRIPTION Use this routine to register an address resolution function for an interface-type/protocol pair. You must call **muxAddrResFuncAdd( )** prior to calling the protocol's **protocolAttach( )** routine. If the driver registers itself as an Ethernet driver, you do not need to call this routine. VxWorks automatically assigns **arpresolve( )** to registered Ethernet devices. The **muxAddrResFuncAdd( )** functionality is intended for using the VxWorks network stack with non-Ethernet drivers that require address resolution.

ifType
Expects a media interface or network driver type, such as can be found in **m2Lib.h**. If using the END model, the *ifType* argument is restricted to the values in **m2Lib.h**. In the NPT model, this restriction does not apply.

protocol
Expects a network service or protocol type, such as can be found in RFC 1700. Look for the values under ETHER TYPES. For example, Internet IP would be identified as 2048 (0x800 hexadecimal). If using the END model, *protocol* is restricted to the values in RFC 1700. In the NPT model, this restriction does not apply.

addrResFunc
Expects a pointer to an address resolution function for this combination of driver type and service type. The prototype of your replacement address resolution function must match that of **arpresolve( )**:

```
int arpresolve
   (
   struct arpcom *   ac,
   struct rtentry *  rt,
   struct mbuf *     m,
   struct sockaddr * dst,
```

M

```
u_char *            desten
)
```

This function returns one upon success, which indicates that *desten* has been updated with the necessary data-link layer information and that the IP sublayer output function can transmit the packet.

This function returns zero if it cannot resolve the address immediately. In the default **arpresolve( )** implementation, resolving the address immediately means **arpresolve( )** was able to find the address in its table of results from previous ARP requests. Returning zero indicates that the table did not contain the information but that the packet has been stored and that an ARP request has been queued.

If the ARP request times out, the packet is dropped. If the ARP request completes successfully, processing that event updates the local ARP table and resubmits the packet to the IP sublayer's output function for transmission. This time, the **arpresolve( )** call will return one.

What is essential to note here is that **arpresolve( )** did not wait for the ARP request to complete before returning. If you replace the default **arpresolve( )** function, you must make sure your function returns as soon as possible and that it never blocks. Otherwise, you block the IP sublayer from transmitting other packets out through the interface for which this packet was queued. You must also make sure that your **arpresolve( )** function takes responsibility for the packet if it returns zero. Otherwise, the packet is dropped.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call **muxAddrResFuncAdd( )** from within the kernel protection domain only, and the data referenced in the *addrResFunc* parameter must reside in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS **OK**, or **ERROR**.

SEE ALSO **muxLib**

muxAddrResFuncDel( )

NAME **muxAddrResFuncDel( )** – delete an address resolution function

SYNOPSIS

```
STATUS muxAddrResFuncDel
    (
    long ifType,            /* ifType of function you want to delete */
    long protocol           /* protocol from which to delete the function */
    )
```

DESCRIPTION

This function deletes the address resolution function registered for the specified interface-protocol pair. If using the NPT architecture, the *ifType* and *protocol* arguments are not restricted to the **m2Lib.h** or RFC 1700 values.

ifType
: Expects a media interface or network driver type. For and END driver, use the values specified in **m2Lib.h**.

protocol
: Expects a network service or protocol type. For example, Internet IP would be identified as 2048 (0x800 hexadecimal). This value can be found in RFC 1700 under the heading, ETHER TYPES.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call **muxAddrResFuncDel( )** from within the kernel protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

OK or **ERROR**.

SEE ALSO

muxLib

M

muxAddrResFuncGet( )

NAME

muxAddrResFuncGet( ) – get the address resolution function for ifType/protocol

SYNOPSIS

```
FUNCPTR muxAddrResFuncGet
    (
    long ifType,              /* ifType from m2Lib.h */
    long protocol             /* protocol from RFC 1700 */
    )
```

DESCRIPTION

This routine gets a pointer to the registered address resolution function for the specified interface-protocol pair. If no such function exists, **muxAddResFuncGet( )** returns **NULL**.

ifType
: Expects a media interface or network driver type, such as those found in **m2Lib.h**. If using the END model, the *ifType* argument is restricted to the **m2Lib.h** values. In the NPT model, this restriction does not apply.

protocol
: Expects a network service or protocol type such as those found in RFC 1700. Look for the values under ETHER TYPES. For example, Internet IP would be identified as 2048 (0x800 hexadecimal). If using the END model, the *protocol* argument is restricted to the RFC 1700 values. In the NPT model, this restriction does not apply.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call **muxAddrResFuncGet( )** from within the kernel protection domain only. In addition, the returned FUNCPTR is valid in the kernel protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

FUNCPTR to the routine or **NULL**.

SEE ALSO

muxLib

muxBind( )

NAME

muxBind( ) – create a binding between a network service and an END

SYNOPSIS

```
void * muxBind
    (
    char * pName,              /* interface name, for example, ln, ei,... */
    int    unit,               /* unit number */
    BOOL (* stackRcvRtn) (void* , long, M_BLK_ID, LL_HDR_INFO * , void* ),
                               /* receive function to be called. */
    STATUS (* stackShutdownRtn) (void* , void* ),
                               /* routine to call to shutdown the stack */
    STATUS (* stackTxRestartRtn) (void* , void* ),
                               /* routine to tell the stack it can transmit */
    void (* stackErrorRtn) (END_OBJ* , END_ERR* , void* ),
                               /* routine to call on an error. */
    long   type,               /* protocol type from RFC1700 and many */
                               /* other sources (for example, 0x800 is IP) */
    char * pProtoName,         /* string name for protocol */
    void * pSpare              /* per protocol spare pointer */
    )
```

DESCRIPTION

A network service uses this routine to bind to an END specified by the *pName* and *unit* arguments (for example, ln and 0, ln and 1, or ei and 0).

NOTE: This routine should only be used to bind to drivers that use the old END driver callback function prototypes. NPT drivers, or END drivers that use the newer callback function prototypes, should use **muxTkBind( )** instead. See the *Network Protocol Toolkit Programmer's Guide* for more information on when to use **muxBind( )** and **muxTkBind( )**.

The *type* argument assigns a network service to one of several classes. Standard services receive the portion of incoming data associated with *type* values from RFC 1700. Only one service for each RFC 1700 type value may be bound to an END.

Services with *type* **MUX_PROTO_SNARF** provide a mechanism for bypassing the standard services for purposes such as firewalls. These services will get incoming packets before any of the standard services.

Promiscuous services with *type* **MUX_PROTO_PROMISC** receive any packets not consumed by the snarf or standard services.

The MUX allows multiple snarf and promiscuous services but does not coordinate between them. It simply delivers available packets to each service in FIFO order. Services that consume packets may prevent "downstream" services from receiving data if the desired packets overlap.

An output service (with *type* **MUX_PROTO_OUTPUT**) receives outgoing data before it is sent to the device. This service type allows two network services to communicate directly and provides a mechanism for loop-back testing. Only one output service is supported for each driver.

The MUX calls the registered *stackRcvRtn* whenever it receives a packet of the appropriate type. If that routine returns **TRUE**, the packet is not offered to any remaining services (or to the driver in the case of output services). A service (including an output service) may return **FALSE** to examine a packet without consuming it. See the description of a **stackRcvRtn( )** in the *Network Protocol Toolkit Programmer's Guide* for additional information about the expected behavior of that routine.

The *stackShutdownRtn* argument provides a function that the MUX can use to shut down the service. See the *Network Protocol Toolkit Programmer's Guide* for a description of how to write such a routine.

The *pProtoName* argument provides the name of the service as a character string. A service name is assigned internally if the argument is **NULL**.

The *pSpare* argument registers a pointer to data defined by the service. The MUX includes this argument in calls to the call back routines from this service.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call **muxBind( )** from within the kernel protection domain only, and the data referenced in the *stackRcvRtn*, *stackShutdownRtn*, *stackTxRestartRtn*, *stackErrorRtn* and *pSpare* parameters must reside in the kernel protection domain. In addition, the returned void pointer is valid in the kernel protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS A cookie identifying the binding between service and driver; or **NULL**, if an error occurred.

ERRNO **S_muxLib_NO_DEVICE**, **S_muxLib_ALREADY_BOUND**, **S_muxLib_ALLOC_FAILED**

SEE ALSO **muxLib**

muxDevExists( )

NAME

muxDevExists( ) – tests whether a device is already loaded into the MUX

SYNOPSIS

```
BOOL muxDevExists
    (
    char * pName,          /* string containing a device name (ln, ei, ...)*/
    int    unit            /* unit number */
    )
```

DESCRIPTION

This routine takes a string device name (for example, ln or ei) and a unit number. If this device is already known to the MUX, it returns **TRUE**. Otherwise, this routine returns **FALSE**.

pName
Expects a pointer to a string containing the device name

unit
Expects the unit number of the device

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call **muxDevExists( )** from within the kernel protection domain only, and the data referenced in the *pName* parameter must reside in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

TRUE, if the device exists; else **FALSE**.

SEE ALSO

muxLib

muxDevLoad( )

NAME

muxDevLoad( ) – load a driver into the MUX

SYNOPSIS

```
void * muxDevLoad
    (
    int    unit,              /* unit number of device */
    END_OBJ * (* endLoad) (char* , void* ),
                              /* load function of the driver */
    char * pInitString,       /* init string for this driver */
    BOOL   loaning,           /* we loan buffers */
    void * pBSP               /* for BSP group */
    )
```

DESCRIPTION

The **muxDevLoad( )** routine loads a network driver into the MUX. Internally, this routine calls the specified *endLoad* routine to initialize the software state of the device. After the device is initialized, you must call **muxDevStart( )** to start the device.

unit
Expects the unit number of the device.

endLoad
Expects a pointer to the network driver's **endLoad( )** or **nptLoad( )** entry point.

pInitString
Expects a pointer to an initialization string, typically a colon-delimited list of options. The **muxDevLoad( )** routine passes this along blindly to the *endLoad* function.

loaning
Currently unused.

pBSP
The MUX blindly passes this argument to the driver, which may or may not use it. Some BSPs use this parameter to pass in tables of functions that the diver can use to deal with the particulars of the BSP.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call **muxDevLoad( )** from within the kernel protection domain only, and the data referenced in the *endLoad* and *pBSP* parameters must reside in the kernel protection domain. In addition, the returned void pointer is valid in the kernel protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

A cookie representing the new device, or **NULL** if an error occurred.

ERRNO

S_muxLib_LOAD_FAILED

SEE ALSO

muxLib

muxDevStart( )

NAME

muxDevStart( ) – start a device by calling its start routine

SYNOPSIS

```
STATUS muxDevStart
    (
    void * pCookie          /* device identifier from muxDevLoad() routine */
    )
```

M

DESCRIPTION

This routine starts a device that has already been initialized and loaded into the MUX with **muxDevLoad( )**. **muxDevStart( )** activates the network interfaces for a device, and calls the device's **endStart( )** or **nptStart( )** routine, which registers the driver's interrupt service routine and does whatever else is needed to allow the device to handle receiving and transmitting. This call to **endStart( )** or **nptStart( )** puts the device into a running state.

pCookie
Expects the pointer returned as the function value of the **muxDevLoad( )** call for this device. This pointer identifies the device.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call **muxDevStart( )** from within the kernel protection domain only, and the data referenced in the *pCookie* parameter must reside in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

OK; **ENETDOWN**, if *pCookie* does not represent a valid device; or **ERROR**, if the start routine for the device fails.

ERRNO

S_muxLib_NO_DEVICE

SEE ALSO

muxLib

muxDevStop( )

NAME

muxDevStop( ) – stop a device by calling its stop routine

SYNOPSIS

```
STATUS muxDevStop
    (
    void * pCookie          /* device identifier from muxDevLoad() routine */
    )
```

DESCRIPTION

This routine stops the device specified in *pCookie*. **muxDevStop( )** calls the device's **endStop( )** or **nptStop( )** routine.

pCookie
Expects the cookie returned as the function value of the **muxDevLoad( )** call for this device. This cookie identifies the device.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call **muxDevStop( )** from within the kernel protection domain only, and the data referenced in the *pCookie* parameter must reside in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS **OK**; **ENETDOWN**, if *pCookie* does not represent a valid device; or **ERROR**, if the **endStop( )** or **nptStop( )** routine for the device fails.

ERRNO **S_muxLib_NO_DEVICE**

SEE ALSO **muxLib**

---

muxDevUnload( )

NAME **muxDevUnload( )** – unloads a device from the MUX

SYNOPSIS

```
STATUS muxDevUnload
    (
    char * pName,            /* a string containing the name of the */
                             /* device for example, ln or ei */
    int    unit              /* the unit number */
    )
```

DESCRIPTION This routine unloads a device from the MUX. This breaks any network connections that use the device. When this routine is called, each service bound to the device disconnects from it with the **stackShutdownRtn( )** routine that was registered by the service. The **stackShutdownRtn( )** should call **muxUnbind( )** to detach from the device. Then, **muxDevUnload( )** calls the device's **endUnload( )** or **nptUnload( )** routine.

pName
: Expects a pointer to a string containing the name of the device, for example **ln** or **ei**

unit
: Expects the unit number of the device indicated by *pName*

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call **muxDevUnLoad( )** from within the kernel protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS **OK**, on success; **ERROR**, if the specified device was not found or some other error occurred; or the error value returned by the driver's **unload( )** routine.

ERRNO **S_muxLib_UNLOAD_FAILED**, **S_muxLib_NO_DEVICE**

SEE ALSO **muxLib**

M

muxIoctl( )

NAME

muxIoctl( ) – send control information to the MUX or to a device

SYNOPSIS

```
STATUS muxIoctl
    (
    void *   pCookie,          /* service/device binding from */
                               /* muxBind()/muxTkBind() */
    int      cmd,              /* command to pass to ioctl */
    caddr_t  data              /* data need for command in cmd */
    )
```

DESCRIPTION

This routine gives the service access to the network driver's control functions. The MUX itself can implement some of the standard control functions, so not all commands necessarily pass down to the device. Otherwise, both command and data pass to the device without modification.

Typical uses of **muxIoctl( )** include commands to start, stop, or reset the network interface, or to add or configure MAC and network addresses.

pCookie
: Expects the cookie returned from **muxBind( )** or **muxTkBind( )**. This cookie indicates the device to which this service is bound.

cmd
: Expects a value indicating the control command you want to execute. For valid *cmd* values, see the description of the **endIoctl( )** and **nptIoctl( )** routines provided in the *Network Protocol Toolkit Programmer's Guide.*

data
: Expects the data or a pointer to the data needed to carry out the command specified in *cmd*.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call **muxIoctl( )** from within the kernel protection domain only, and the data referenced in the *pCookie* and *data* parameters must reside in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

OK; **ENETDOWN**, if *pCookie* does not represent a bound device; or **ERROR**, if the command fails.

ERRNO

S_muxLib_NO_DEVICE

SEE ALSO

muxLib

muxLibInit( )

NAME **muxLibInit( )** – initialize global state for the MUX

SYNOPSIS
```
STATUS muxLibInit (void)
```

DESCRIPTION This routine initializes all global states for the MUX.

RETURNS **OK** or **ERROR**.

SEE ALSO **muxLib**

muxLinkHeaderCreate( )

NAME **muxLinkHeaderCreate( )** – attach a link-level header to a packet

SYNOPSIS
```
M_BLK_ID muxLinkHeaderCreate
    (
    void *   pCookie,          /* protocol/device binding from muxBind() */
    M_BLK_ID pPacket,          /* structure containing frame contents */
    M_BLK_ID pSrcAddr,         /* structure containing source address */
    M_BLK_ID pDstAddr,         /* structure containing destination address */
    BOOL     bcastFlag         /* use broadcast destination (if available)? */
    )
```

DESCRIPTION This routine constructs a link-level header using the source address of the device indicated by the *pCookie* argument as returned from the **muxBind( )** routine.

The *pDstAddr* argument provides an **M_BLK_ID** buffer containing the link-level destination address. Alternatively, the *bcastFlag* argument, if **TRUE**, indicates that the routine should use the link-level broadcast address, if available for the device. Although other information contained in the *pDstAddr*argument must be accurate, the address data itself is ignored in that case.

The *pPacket* argument contains the contents of the resulting link-level frame. This routine prepends the new link-level header to the initial **mBlk** in that network packet if space is available or allocates a new **mBlk**-**clBlk**-cluster triplet and prepends it to the **mBlk** chain. When construction of the header is complete, it returns an **M_BLK_ID** that points to the initial **mBlk** in the assembled link-level frame.

RETURNS **M_BLK_ID** or **NULL**.

M

ERRNO
S_muxLib_INVALID_ARGS

SEE ALSO
muxLib

muxMCastAddrAdd( )

NAME
muxMCastAddrAdd( ) – add a multicast address to a device's multicast table

SYNOPSIS

```
STATUS muxMCastAddrAdd
    (
    void * pCookie,            /* binding instance from muxBind() or */
                               /* muxTkBind() */
    char * pAddress            /* address to add to the table */
    )
```

DESCRIPTION
This routine adds an address to the multicast table maintained by a device. This routine calls the driver's **endMCastAddrAdd( )** or **nptMCastAddrAdd( )** routine to accomplish this.

If the device does not support multicasting, **muxMCastAddrAdd( )** will return **ERROR** and **errno** will be set to **ENOTSUP** (assuming the driver has been written properly).

pCookie
: Expects the cookie returned from the **muxBind( )** or **muxTkBind( )** call. This cookie identifies the device to which the MUX has bound this service.

pAddress
: Expects a pointer to a character string containing the address you want to add.

VXWORKS AE PROTECTION DOMAINS
Under VxWorks AE, you can call **muxMCastAddrAdd( )** from within the kernel protection domain only, and the data referenced in the *pCookie* parameter must reside in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS
OK; **ENETDOWN**, if *pCookie* does not represent a valid device; or **ERROR**, if the device's **endMCastAddrAdd( )** function fails.

ERRNO
S_muxLib_NO_DEVICE

SEE ALSO
muxLib

muxMCastAddrDel( )

NAME

muxMCastAddrDel( ) – delete a multicast address from a device's multicast table

SYNOPSIS

```
STATUS muxMCastAddrDel
    (
    void * pCookie,          /* binding instance from muxBind() or */
                             /* muxTkBind() */
    char * pAddress          /* Address to delete from the table. */
    )
```

DESCRIPTION

This routine deletes an address from the multicast table maintained by a device by calling that device's **endMCastAddrDel( )** or **nptMCastAddrDel( )** routine.

If the device does not support multicasting, **muxMCastAddrAdd( )** will return **ERROR** and **errno** will be set to **ENOTSUP** (assuming the driver has been written properly).

pCookie
: Expects the cookie returned from **muxBind( )** or **muxTkBind( )** call. This cookie identifies the device to which the MUX bound this service.

pAddress
: Expects a pointer to a character string containing the address you want to delete.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call **muxMCastAddrDell( )** from within the kernel protection domain only, and the data referenced in the *pCookie* parameter must reside in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

OK; **ENETDOWN**, if *pCookie* does not represent a valid driver; or **ERROR**, if the driver's registered **endMCastAddrDel( )** or **nptMCastAddrDel( )** functions fail.

ERRNO

S_muxLib_NO_DEVICE

SEE ALSO

muxLib

muxMCastAddrGet( )

NAME

muxMCastAddrGet( ) – get the multicast address table from the MUX/Driver

SYNOPSIS

```
int muxMCastAddrGet
    (
    void *        pCookie,    /* binding instance from muxBind() or */
                              /* muxTkBind() */
    MULTI_TABLE * pTable      /* pointer to a table to be filled and */
                              /* returned. */
    )
```

DESCRIPTION

This routine writes the list of multicast addresses for a specified device into a buffer. To get this list, it calls the driver's own **endMCastAddrGet( )** or **nptMCastAddrGet( )** routine.

pCookie
: Expects the cookie returned from **muxBind( )** or **muxTkBind( )** call. This cookie indicates the device to which the MUX has bound this service.

pTable
: Expects a pointer to a **MULTI_TABLE** structure. You must have allocated this structure at some time before the call to **muxMCastAddrGet( )**. The **MULTI_TABLE** structure is defined in **end.h** as:

```
typedef struct multi_table
    {
    int     tableLen;  /* length of table in bytes */
    char *  pTable;    /* pointer to entries */
    } MULTI_TABLE;
```

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call **muxMCastAddrGet( )** from within the kernel protection domain only, and the data referenced in the *pCookie* parameter must reside in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

OK; **ENETDOWN**, if *pCookie* does not represent a valid driver; or **ERROR**, if the driver's registered **endMCastAddrGet( )** or **nptMCastAddrGet( )** routines fail.

ERRNO

S_muxLib_NO_DEVICE

SEE ALSO

muxLib

muxPacketAddrGet( )

NAME

muxPacketAddrGet( ) – get addressing information from a packet

SYNOPSIS

```
STATUS muxPacketAddrGet
    (
    void *   pCookie,          /* protocol/device binding from muxBind() */
    M_BLK_ID pMblk,            /* structure to contain packet */
    M_BLK_ID pSrcAddr,         /* structure containing source address */
    M_BLK_ID pDstAddr,         /* structure containing destination address */
    M_BLK_ID pESrcAddr,        /* structure containing the end source */
    M_BLK_ID pEDstAddr         /* structure containing the end destination */
    )
```

DESCRIPTION

The routine returns the immediate source, immediate destination, ultimate source, and ultimate destination addresses from the packet pointed to in the first **M_BLK_ID**. This routine makes no attempt to extract that information from the packet directly. Instead, it passes the packet to the driver call that knows how to interpret the packets it has received.

pCookie
Expects the cookie returned from the **muxBind( )** call. This cookie indicates the device to which the MUX bound this service.

pMblk
Expects an **M_BLK_ID** representing packet data from which the addressing information is to be extracted

pSrcAddr
Expects **NULL** or an **M_BLK_ID** which will hold the local source address extracted from the packet

pDstAddr
Expects **NULL** or an **M_BLK_ID** which will hold the local destination address extracted from the packet

pESrcAddr
Expects **NULL** or an **M_BLK_ID** which will hold the end source address extracted from the packet

pEDstAddr
Expects **NULL** or an **M_BLK_ID** which will hold the end destination address extracted from the packet

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call **muxPacketAddrGet( )** from within the kernel protection domain only, and the data referenced in the parameters must reside in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS OK or ERROR.

ERRNO S_muxLib_NO_DEVICE

SEE ALSO **muxLib**

muxPacketDataGet( )

NAME **muxPacketDataGet( )** – return the data from a packet

SYNOPSIS

```
STATUS muxPacketDataGet
    (
    void *        pCookie,      /* protocol/device binding from muxBind() */
    M_BLK_ID      pMblk,        /* returns the packet data */
    LL_HDR_INFO * pLinkHdrInfo  /* returns the packet header information */
    )
```

DESCRIPTION Any service bound to a driver may use this routine to extract the packet data and remove the link-level header information. This routine copies the header information from the packet referenced in *pMblk* into the **LL_HDR_INFO** structure referenced in *pLinkHdrInfo*.

pCookie
: Expects the cookie returned from the **muxBind( )** call. This cookie indicates the device to which the MUX bound this service.

pMblk
: Expects a pointer to an **mBlk** or **mBlk** cluster representing a packet containing the data to be returned

pLinkHdrInfo
: Expects a pointer to an **LL_HDR_INFO** structure into which the packet header information is copied from the incoming **mBlk**

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call **muxPacketDataGet( )** from within the kernel protection domain only, and the data referenced in the parameters must reside in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS **OK**; or **ERROR**, if the device type is not recognized.

ERRNO S_muxLib_NO_DEVICE

SEE ALSO **muxLib**

muxPollDevAdd( )

NAME **muxPollDevAdd( )** – adds a device to list polled by **tMuxPollTask**

SYNOPSIS

```
STATUS muxPollDevAdd
    (
    int    unit,              /* Device unit number */
    char * pName              /* Device name */
    )
```

DESCRIPTION This routine adds a device to list of devices polled by **tMuxPollTask**. It assumes that you have already called **muxPollStart( )** and that **tMuxPollTask** is still running.

NOTE: You cannot use a device for **WDB_COMM_END** type debugging while that device is on the **tMuxPollTask** poll list.

RETURNS **OK** or **ERROR**

SEE ALSO **muxLib**

muxPollDevDel( )

NAME **muxPollDevDel( )** – removes a device from the list polled by **tMuxPollTask**

SYNOPSIS

```
STATUS muxPollDevDel
    (
    int    unit,              /* Device unit number */
    char * pName              /* Device name */
    )
```

DESCRIPTION This routine removes a device from the list of devices polled by **tMuxPollTask**. If you remove the last device on the list, a call to **muxPollDevDel( )** also makes an internal call to **muxPollEnd( )**. This shuts down **tMuxPollTask** completely.

RETURNS **OK** or **ERROR**

SEE ALSO **muxLib**

muxPollDevStat( )

NAME **muxPollDevStat( )** – reports whether device is on list polled by **tMuxPollTask**

SYNOPSIS

```
BOOL muxPollDevStat
    (
    int    unit,              /* Device unit number */
    char * pName              /* Device name */
    )
```

DESCRIPTION This routine returns true or false depending on whether the specified device is on the list of devices polled by **tMuxPollTask**.

RETURNS **TRUE**, if it is; or **FALSE**.

SEE ALSO **muxLib**

muxPollEnd( )

NAME **muxPollEnd( )** – shuts down **tMuxPollTask** and returns devices to interrupt mode

SYNOPSIS

```
STATUS muxPollEnd ()
```

DESCRIPTION This routine shuts down **tMuxPollTask** and returns network devices to run in their interrupt-driven mode.

RETURNS **OK** or **ERROR**

SEE ALSO **muxLib**

muxPollReceive( )

NAME

muxPollReceive( ) – now **deprecated**, see **muxTkPollReceive( )**

SYNOPSIS

```
STATUS muxPollReceive
    (
    void *   pCookie,          /* binding instance from muxBind() */
    M_BLK_ID pNBuff            /* a vector of buffers passed to us */
    )
```

DESCRIPTION

NOTE: This routine has been deprecated in favor of **muxTkPollReceive( )**

Upper layers can call this routine to poll for a packet.

pCookie
: Expects the cookie that was returned from **muxBind( )**. This cookie indicates which driver to query for available data.

pNBuff
: Expects a pointer to a buffer chain into which to receive data.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call **muxPollReceive( )** from within the kernel protection domain only, and the data referenced in the *pCookie* and *pNBuff* parameters must reside in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

OK; **ENETDOWN**, if the *pCookie* argument does not represent a loaded driver; or an error value returned from the driver's registered **endPollReceive( )** function.

ERRNO

S_muxLib_NO_DEVICE

SEE ALSO

muxLib

M

muxPollSend( )

NAME

muxPollSend( ) – now **deprecated**, see **muxTkPollSend( )**

SYNOPSIS

```
STATUS muxPollSend
    (
    void *   pCookie,          /* binding instance from muxBind() */
    M_BLK_ID pNBuff            /* data to be sent */
    )
```

DESCRIPTION

This routine transmits a packet for the service specified by *pCookie*. You got this cookie from a previous bind call that bound the service to a particular interface. This **muxPollSend( )** call uses this bound interface to transmit the packet. The *pNBuff* argument is a buffer (**mBlk**) chain that contains the packet to be sent.

RETURNS

OK; **ENETDOWN**, if *pCookie* does not represent a valid device; **ERROR**, if the device type is not recognized; or an error value from the device's registered **endPollSend( )** routine.

ERRNO

S_muxLib_NO_DEVICE

SEE ALSO

muxLib

muxPollStart( )

NAME **muxPollStart( )** – initialize and start the MUX poll task

SYNOPSIS

```
STATUS muxPollStart
    (
    int numDev,              /* Maximum number of devices to support */
                             /* poll mode. */
    int priority,            /* tMuxPollTask priority, not to exceed */
                             /* tNetTask. */
    int delay                /* Delay, in ticks, at end of each polling */
                             /* cycle. */
    )
```

DESCRIPTION This routine initializes and starts the MUX poll task, **tMuxPollTask**. This task runs an infinite loop in which it polls each of the interfaces referenced on a list of network interfaces. To add or remove devices from this list, use **muxPollDevAdd( )** and **muxPollDevDel( )**. Removing all devices from the list automatically triggers a call to **muxPollEnd( )**, which shuts down **tMuxPollTask**.

Using the *priority* parameter, you assign the priority to **tMuxPollTask**. Valid values are between 0 and 255, inclusive. However, you must not set the priority of **tMuxPollTask** to exceed that of **tNetTask**. Otherwise, you risk shutting **tNetTask** out from getting processor time. To reset the **tMuxPollTask** priority after launch, use **muxTaskPrioritySet( )**.

Using the *delay* parameter, you can set up a delay at the end of each trip though the poll list. To reset the value of this delay after the launch of **tNetTask**, call **muxTaskDelaySet( )**.

To shut down **tMuxPollTask**, call **muxPollEnd( )**.

RETURNS **OK** or **ERROR**

SEE ALSO **muxLib**

M

muxSend( )

NAME

muxSend( ) – send a packet out on a network interface

SYNOPSIS

```
STATUS muxSend
    (
    void *   pCookie,          /* protocol/device binding from muxBind() */
    M_BLK_ID pNBuff            /* data to be sent */
    )
```

DESCRIPTION

This routine transmits a packet for the service specified by *pCookie*. You got this cookie from a previous bind call that bound the service to a particular interface. This **muxSend( )** call uses this bound interface to transmit the packet.

pCookie
Expects the cookie returned from **muxBind( )**. This cookie identifies a particular service-to-interface binding.

pNBuff
Expects a pointer to the buffer that contains the packet you want to transmit. Before you call **muxSend( )**, you need to put the addressing information at the head of the buffer. To do this, call **muxAddressForm( )**.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call **muxSend( )** from within the kernel protection domain only, and the data referenced in the *pCookie* and *pNBuff* parameters must reside in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

OK; **ENETDOWN**, if *pCookie* does not represent a valid binding; or **ERROR**, if the driver's **endSend( )** routine fails.

ERRNO

S_muxLib_NO_DEVICE

SEE ALSO

muxLib

muxShow( )

NAME

muxShow( ) – display configuration of devices registered with the MUX

SYNOPSIS

```
void muxShow
    (
    char * pDevName,          /* pointer to device name, or NULL for all */
    int    unit               /* unit number for a single device */
    )
```

DESCRIPTION

If the *pDevName* and *unit* arguments specify an existing device, this routine reports the name and type of each protocol bound to it. Otherwise, if *pDevName* is **NULL**, the routine displays the entire list of existing devices and their associated protocols.

pDevName
A string that contains the name of the device, or a null pointer to indicate "all devices."

unit
Specifies the unit number of the device (if *pDevName* is not a null pointer).

RETURNS

N/A

SEE ALSO

muxLib

muxTaskDelayGet( )

NAME

muxTaskDelayGet( ) – get the delay on the polling task

SYNOPSIS

```
STATUS muxTaskDelayGet
    (
    int* pDelay
    )
```

DESCRIPTION

This routine returns the amount of delay (in ticks) that is inserted between the polling runs of **tMuxPollTask**. This value is written to the location specified by *pDelay*.

RETURNS

OK; or **ERROR**, if **NULL** is passed in the *pDelay* variable.

SEE ALSO

muxLib

M

muxTaskDelaySet( )

NAME **muxTaskDelaySet( )** – set the inter-cycle delay on the polling task

SYNOPSIS

```
STATUS muxTaskDelaySet
    (
    int delay
    )
```

DESCRIPTION This routine sets up a delay (measured in ticks) that is inserted at the end of each run through the list of devices polled by **tMuxPollTask**.

RETURNS **OK**; or **ERROR**, if you specify a delay less than zero.

SEE ALSO **muxLib**

muxTaskPriorityGet( )

NAME **muxTaskPriorityGet( )** – get the priority of **tMuxPollTask**

SYNOPSIS

```
STATUS muxTaskPriorityGet
    (
    int* pPriority
    )
```

DESCRIPTION This routine returns the current priority of **tMuxPollTask**. This value is returned to the location specified by the *pPriority* parameter.

RETURNS **OK**; or **ERROR**, if **NULL** is passed in the *pPriority* parameter.

SEE ALSO **muxLib**

muxTaskPrioritySet( )

NAME **muxTaskPrioritySet( )** – reset the priority of **tMuxPollTask**

SYNOPSIS

```
STATUS muxTaskPrioritySet
    (
    int priority
    )
```

DESCRIPTION This routine resets the priority of a running **tMuxPollTask**. Valid task priorities are values between zero and 255 inclusive. However, do not set the priority of **tMuxPollTask** to exceed that of **tNetTask**. Otherwise, you will shut out **tNetTask** from getting any processor time.

RETURNS **OK**; or **ERROR**, if you specify a non-valid priority value.

SEE ALSO **muxLib**

M

muxTkBind( )

NAME **muxTkBind( )** – bind an NPT protocol to a driver

SYNOPSIS

```
void * muxTkBind
    (
    char * pName,             /* interface name, for example, ln, ei,... */
    int    unit,              /* unit number */
    BOOL (* stackRcvRtn) (void* ,long, M_BLK_ID, void * ),
                              /* receive function to be called. */
    STATUS (* stackShutdownRtn) (void * ),
                              /* routine to call to shutdown the stack */
    STATUS (* stackTxRestartRtn) (void * ),
                              /* routine to tell the stack it can transmit */
    void (* stackErrorRtn) (void* , END_ERR* ),
                              /* routine to call on an error. */
    long   type,              /* protocol type from RFC1700 and many */
                              /* other sources (for example, 0x800 is IP) */
    char * pProtoName,        /* string name for protocol */
    void * pNetCallbackId,    /* returned to network service sublayer */
                              /* during recv */
    void * pNetSvcInfo,       /* reference to netSrvInfo structure */
    void * pNetDrvInfo        /* reference to netDrvInfo structure */
    )
```

DESCRIPTION

A network protocol, network service, or service sublayer uses this routine to bind to a specific driver. This bind routine is valid both for END and NPT drivers, but the specified stack routine parameters must use the NPT function prototypes, and are somewhat different from those used with **muxBind( )**.

The driver is specified by the *pName* and *unit* arguments, (for example, ln and 0, ln and 1, or ei and 0).

pName
: Expects a pointer to a character string that contains the name of the device that this network service wants to use to send and receive packets.

unit
: Expects the unit number of the device of the type indicated by *pName*.

stackRcvRtn
: Expects a pointer to the function that the MUX will call when it wants to pass a packet up to the network service. For a description of how to write this routine, see the *WindNet TCP/IP Network Programmer's Guide*

stackShutdownRtn
: Expects a pointer to the function that the MUX will call to shutdown the network service. For a description of how to write such a routine, see the *WindNet TCP/IP Network Programmer's Guide*

stackTxRestartRtn
: Expects a pointer to the function that the MUX will call after packet transmission has been suspended, to tell the network service that it can continue transmitting packets. For a description of how to write this routine, see the *WindNet TCP/IP Network Programmer's Guide*

stackErrorRtn
: Expects a pointer to the function that the MUX will call to give errors to the network service. For a description of how to write this routine, see the section *WindNet TCP/IP Network Programmer's Guide*

type
: Expects a value that indicates the protocol type. The MUX uses this type to prioritize a network service as well as to modify its capabilities. For example, a network service of type **MUX_PROTO_SNARF** has the highest priority (see the description of protocol prioritizing provided in *WindNet TCP/IP Network Programmer's Guide.* Aside from **MUX_PROTO_SNARF** and **MUX_PROTO_PROMISC**, valid network service types include any of the values specified in RFC 1700, or can be user-defined.

The *stackRcvRtn* is called whenever the MUX has a packet of the specified type. If the type is **MUX_PROTO_PROMISC**, the protocol is considered promiscuous and will get all of the packets that have not been consumed by any other protocol. If the type is **MUX_PROTO_SNARF**, it will get all of the packets that the MUX sees.

If the type is **MUX_PROTO_OUTPUT**, this network service is an output protocol and all packets that are to be output on this device are first passed to *stackRcvRtn* routine rather than being sent to the device. This can be used by a network service that needs to send packets directly to another network service, or in a loop-back test. If the *stackRcvRtn* returns **OK**, the packet is consumed and as no longer available. The *stackRcvRtn* for an output protocol may return **ERROR** to indicate that it wants to look at the packet without consuming it.

pProtoName
Expects a pointer to a character string for the name of this network service. This string can be **NULL**, in which case a network service name is assigned internally.

pNetCallbackId
Expects a pointer to a structure defined by the protocol. This argument is passed up to the protocol as the first argument of all the callbacks. This argument corresponds to the *pSpare* argument in **muxBind( )**

pNetSvcInfo
Reference to an optional structure specifying network service layer information needed by the driver

pNetDrvInfo
Reference to an optional structure specifying network driver information needed by the network protocol, network service, or service sublayer

RETURNS A cookie that uniquely represents the binding instance, or **NULL** if the bind fails.

ERRNO **S_muxLib_NO_DEVICE**, **S_muxLib_END_BIND_FAILED**, **S_muxLib_NO_TK_DEVICE**, **S_muxLib_NOT_A_TK_DEVICE**, **S_muxLib_ALREADY_BOUND**, **S_muxLib_ALLOC_FAILED**

SEE ALSO **muxTkLib**, **muxBind( )**

muxTkCookieGet( )

NAME **muxTkCookieGet( )** – returns the cookie for a device

SYNOPSIS

```
void *muxTkCookieGet
    (
    char * pName,              /* Device Name */
    int    unit                /* Device Unit */
    )
```

DESCRIPTION This routine returns the cookie for a device.

RETURNS a cookie to the device or **NULL** if unsuccessful

SEE ALSO **muxTkLib**

muxTkDrvCheck( )

NAME **muxTkDrvCheck( )** – checks if the device is an NPT or an END interface

SYNOPSIS

```
int muxTkDrvCheck
    (
    char * pDevName           /* device name */
    )
```

DESCRIPTION This function returns 1 if the driver indicated by *pDevName* is of the Toolkit (NPT) paradigm, and 0 (zero) if it is an END. This routine is called by the network service sublayer so that it can discover the driver type before it binds to it via the MUX.

RETURNS 1 for an NPT driver, 0 for an END or other driver, or **ERROR** (-1) if no device is found with the given name

SEE ALSO **muxTkLib**, **muxTkBind( )**, **muxBind( )**

muxTkPollReceive( )

NAME **muxTkPollReceive( )** – poll for a packet from a NPT or END driver

SYNOPSIS

```
STATUS muxTkPollReceive
    (
    void *   pCookie,       /* cookie from muxTkBind routine */
    M_BLK_ID pNBuff,        /* a vector of buffers passed to us */
    void *   pSpare         /* a reference to spare data is returned here */
    )
```

DESCRIPTION This is the routine that an upper layer can call to poll for a packet. Any service type retrieved from the MAC frame is passed via the reserved member of the **M_BLK** header.

This API effectively replaces **muxPollReceive( )** for both END and NPT drivers.

For an NPT driver its **pollReceive( )** entry point is called based on the new prototype:

```
STATUS nptPollReceive
    (
    END_OBJ * pEND,           /* END object */
    M_BLK_ID  pPkt,           /* network packet buffer */
    long *    pNetSvc,        /* service type from MAC frame */
    long *    pNetOffset,     /* offset to network packet */
```

```
    void *    pSpareData      /* optional network service data */
    )
```

The **pollReceive( )** entry point for an END driver uses the original prototype:

```
STATUS endPollRcv
    (
    END_OBJ * pEND,    /* END object */
    M_BLK_ID  pPkt,    /* network packet buffer */
    )
```

An END driver must continue to provide the **packetDataGet( )** entry point

pCookie
Expects the cookie that was returned from **muxBind( )** or **muxTkBind( )**. This "cookie" identifies the driver.

pNBuff
Expects a pointer to a buffer chain into which incoming data will be put.

pSpareData
A pointer to any optional spare data provided by a NPT driver. Always **NULL** with an END driver.

M

RETURNS **OK**; **EAGAIN**, if no packet was available; **ENETDOWN**, if the **pCookie**does not represent a loaded driver; or an error value returned from the driver's registered **pollReceive( )** function.

ERRNO **S_muxLib_NO_DEVICE**

SEE ALSO **muxTkLib**

muxTkPollSend( )

NAME **muxTkPollSend( )** – send a packet out in polled mode to an END or NPT interface

SYNOPSIS

```
STATUS muxTkPollSend
    (
    void *   pCookie,          /* returned by muxTkBind()*/
    M_BLK_ID pNBuff,           /* data to be sent */
    char *   dstMacAddr,       /* destination MAC address */
    USHORT   netType,          /* network protocol that is calling us * is */
                               /* netType redundant? * */
    void *   pSpareData        /* spare data passed on each send */
    )
```

DESCRIPTION

This routine uses *pCookie* to find a specific network interface and use that driver's **pollSend( )** routine to transmit a packet.

This routine replaces the **muxPollSend( )** routine for both END and NPT drivers.

When using this routine, the driver does not need to call **muxAddressForm( )** to complete the packet, nor does it need to prepend an **mBlk** of type **MF_IFADDR** containing the destination address.

An NPT driver's **pollSend( )** entry point is called based on this prototype:

```
STATUS nptPollSend
    (
    END_OBJ * pEND,           /* END object */
    M_BLK_ID  pPkt,           /* network packet to transmit */
    char *    pDstAddr,       /* destination MAC address */
    long      netType         /* network service type */
    void *    pSpareData      /* optional network service data */
    )
```

The **pollSend( )** entry point for an END uses this prototype:

```
STATUS endPollSend
    (
    END_OBJ * pEND,   /* END object */
    M_BLK_ID  pPkt,   /* network packet to transmit */
    )
```

An END driver must provide the **addressForm( )** entry point to construct the appropriate link-level header. The *pDst* and *pSrc* **M_BLK** arguments to that routine supply the link-level addresses with the **mData** and **mLen** fields. The reserved field of the destination **M_BLK** contains the network service type. Both arguments *must* be treated as read-only.

pCookie
: Expects the cookie returned from **muxBind( )** or **muxTkBind( )**. This cookie identifies the device to which the MUX has bound this protocol.

pNBuff
: The network packet to be sent.

dstMacAddr
: Destination MAC address to which packet is to be sent

netType
: Network service type that will be used to identify the payload data in the MAC frame.

pSpareData
: Reference to any additional data the network service wants to pass to the driver during the send operation.

RETURNS OK, ENETDOWN if *pCookie* doesn't represent a valid device, or an error if the driver's **pollSend( )** routine fails.

ERRNO S_muxLib_NO_DEVICE

SEE ALSO **muxTkLib**

muxTkReceive( )

NAME **muxTkReceive( )** – receive a packet from a NPT driver

SYNOPSIS

```
STATUS muxTkReceive
    (
    void *   pCookie,         /* cookie passed in endLoad() call */
    M_BLK_ID pMblk,           /* a buffer passed to us. */
    long     netSvcOffset,    /* offset to network datagram in the packet */
    long     netSvcType,      /* network service type */
    BOOL     uniPromiscuous,  /* TRUE when driver is in promiscuous mode */
    void *   pSpareData       /* out of band data */
    )
```

DESCRIPTION This is the routine that the NPT driver calls to hand a packet to the MUX. This routine forwards the received **mBlk** chain to the network service sublayer by calling its registered **stackRcvRtn( )**.

Typically, a driver includes an interrupt handling routine to process received packets. It should keep processing to a minimum during interrupt context and then arrange for processing of the received packet within task context.

Once the frame has been validated, the driver should pass it to the MUX with the **receiveRtn** member of its **END_OBJ** structure. This routine has the same prototype as (and typically is) **muxTkReceive( )**.

Depending on the protocol type (for example, **MUX_PROTO_SNARF** or **MUX_PROTO_PROMISC**), this routine either forwards the received packet chain unmodified or it changes the data pointer in the **mBlk** to strip off the frame header before forwarding the packet.

pCookie
: Expects the **END_OBJ** pointer returned by the driver's **endLoad( )** or **nptLoad( )** function

pMblk
: Expects a pointer to the **mBlk** structure containing the packet that has been received

netSvcOffset
Expects an offset into the frame to the point where the data field (the network service layer header) begins

netSvcType
Expects the network service type of the service for which the packet is destined (typically this value can be found in the header of the received frame)

uniPromiscuous
Expects a boolean set to **TRUE** when driver is in promiscuous mode and receives a unicast or a multicast packet not intended for this device. When **TRUE** the packet is not handed over to network services other than those registered as SNARF or PROMISCUOUS.

pSpareData
Expects a pointer to any spare data the driver needs to pass up to the network service layer, or **NULL**

RETURNS **OK** or **ERROR**.

ERRNO **S_muxLib_NO_DEVICE**

SEE ALSO **muxTkLib**

---

muxTkSend( )

NAME **muxTkSend( )** – send a packet out on a Toolkit or END network interface

SYNOPSIS

```
STATUS muxTkSend
    (
    void *   pCookie,         /* returned by muxTkBind()*/
    M_BLK_ID pNBuff,          /* data to be sent */
    char *   dstMacAddr,      /* destination MAC address */
    USHORT   netType,         /* network protocol that is calling us * is */
                              /* netType redundant? * */
    void *   pSpareData       /* spare data passed on each send */
    )
```

DESCRIPTION This routine uses *pCookie* to find a specific network interface and uses that driver's send routine to transmit a packet.

The transmit entry point for an NPT driver uses the following prototype:

```
STATUS nptSend
    (
    END_OBJ * pEND,            /* END object */
    M_BLK_ID  pPkt,            /* network packet to transmit */
    char *    pDstAddr,        /* destination MAC address */
    int       netType          /* network service type */
    void *    pSpareData       /* optional network service data */
    )
```

The transmit entry point for an END driver the following prototype:

```
STATUS endSend
    (
    void *    pEND,    /* END object */
    M_BLK_ID  pPkt,    /* Network packet to transmit */
    )
```

An END driver must continue to provide the **addressForm( )** entry point to construct the appropriate link-level header. The *pDst* and *pSrc* **M_BLK** arguments to that routine supply the link-level addresses with the **mData** and **mLen** fields. The reserved field of the destination **M_BLK** contains the network service type. Both arguments *must* be treated as read-only.

To send a fully formed physical layer frame to a device using an NPT driver (which typically forms the frame itself), set the **M_L2HDR** flag in the **mBlk** header.

A driver may be written so that it returns the error **END_ERR_BLOCK** if the driver has insufficient resources to transmit data. The network service sublayer can use this feedback to establish a flow control mechanism by holding off on making any further calls to **muxTkSend( )** until the device is ready to restart transmission, at which time the device should call **muxTxRestart( )** which will call the service sublayer's **stackRestartRtn( )** that was registered for the interface at bind time.

pCookie
: Expects the cookie returned from **muxTkBind( )**. This Cookie identifies the device to which the MUX has bound this protocol.

pNBuff
: The network packet to be sent, formed into an **mBlk** chain.

dstMacAddr
: Destination MAC address to which packet is to be sent, determined perhaps by calling the address resolution function that was registered for this service/device interface.

netType
: Network service type of the sending service. This will be used to identify the payload type in the MAC frame.

pSpareData
Reference to any additional data the network service wants to pass to the driver during the send operation.

NOTE: A driver may return **END_ERR_BLOCK** if it is temporarily unable to complete the send, and then call **muxTxRestart( )** to indicate that it is again able to send data. If the driver has been written in this way, **muxTkSend( )** will pass the **ERR_END_BLOCK** back as its own return value and the service can wait for its **stackRestartRtn( )** callback routine to be called before trying the send operation again.

RETURNS **OK**; **ENETDOWN**, if *pCookie* doesn't represent a valid device; or an error, if the driver's **send( )** routine fails.

ERRNO **S_muxLib_NO_DEVICE**

SEE ALSO **muxTkLib**

muxUnbind( )

NAME **muxUnbind( )** – detach a network service from the specified device

SYNOPSIS

```
STATUS muxUnbind
    (
    void *   pCookie,      /* binding instance from muxBind() or */
                           /* muxTkBind() */
    long     type,         /* type passed to muxBind() or muxTkBind() call */
    FUNCPTR stackRcvRtn    /* pointer to stack receive routine */
    )
```

DESCRIPTION This routine disconnects a network service from the specified device. The *pCookie* argument indicates the service/device binding returned by the **muxBind( )** or **muxTkBind( )** routine. The *type* and *stackRcvRtn* arguments must also match the values given to the original **muxBind( )** or **muxTkBind( )** call.

NOTE: If **muxUnbind( )** returns **ERROR**, and **errno** is set to **EINVAL**, this indicates that the device is not bound to the service.

RETURNS **OK**; or **ERROR**, if **muxUnbind( )** fails.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call **muxUnBind( )** from within the kernel protection domain only, and the data referenced in the *stackRcvRtn* and *pCookie* parameters must reside in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

ERRNO
EINVAL, S_muxLib_NO_DEVICE

SEE ALSO
muxLib

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.

SEE ALSO
usrFsLib

M

nanosleep( )

NAME

nanosleep( ) – suspend the current task until the time interval elapses (POSIX)

SYNOPSIS

```
int nanosleep
    (
    const struct timespec * rqtp, /* time to delay */
    struct timespec *       rmtp  /* premature wakeup (NULL=no result) */
    )
```

DESCRIPTION

This routine suspends the current task for a specified time *rqtp*or until a signal or event notification is made.

The suspension may be longer than requested due to the rounding up of the request to the timer's resolution or to other scheduling activities (*e.g.*, a higher priority task intervenes).

The **timespec** structure is defined as follows:

```
struct timespec
    {
                                /* interval = tv_sec*10**9 + tv_nsec */
     time_t tv_sec;             /* seconds */
     long tv_nsec;              /* nanoseconds (0 - 1,000,000,000) */
     };
```

If *rmtp* is non-**NULL**, the **timespec** structure is updated to contain the amount of time remaining. If *rmtp* is **NULL**, the remaining time is not returned. The *rqtp* parameter is greater than 0 or less than or equal to 1,000,000,000.

RETURNS

0 (**OK**), or -1 (**ERROR**) if the routine is interrupted by a signal or an asynchronous event notification, or *rqtp* is invalid.

ERRNO

EINVAL, **EINTR**

SEE ALSO

timerLib, **sleep( )**, **taskDelay( )**

netBufLibInit( )

NAME **netBufLibInit( )** – initialize **netBufLib**

SYNOPSIS

```
STATUS netBufLibInit (void)
```

DESCRIPTION This routine executes during system startup if **INCLUDE_NETWORK** is defined when the image is built. It links the network buffer library into the image.

RETURNS **OK** or **ERROR**.

SEE ALSO **netBufLib**

netClBlkFree( )

NAME **netClBlkFree( )** – free a **clBlk**-cluster construct back to the memory pool

SYNOPSIS

```
void netClBlkFree
    (
    NET_POOL_ID pNetPool,     /* pointer to the net pool */
    CL_BLK_ID   pClBlk        /* pointer to the clBlk to free */
    )
```

DESCRIPTION This routine decrements the reference counter in the specified **clBlk**. If the reference count falls to zero, this routine frees both the **clBlk** and its associated cluster back to the specified memory pool.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS N/A

SEE ALSO **netBufLib**

netClBlkGet( )

NAME

netClBlkGet( ) – get a **clBlk**

SYNOPSIS

```
CL_BLK_ID netClBlkGet
    (
    NET_POOL_ID pNetPool,      /* pointer to the net pool */
    int         canWait        /* M_WAIT/M_DONTWAIT */
    )
```

DESCRIPTION

This routine gets a **clBlk** from the specified memory pool.

pNetPool
Expects a pointer to the pool from which you want a **clBlk**.

canWait
Expects either **M_WAIT** or **M_DONTWAIT**. If no **clBlk** is immediately available, the **M_WAIT** value allows this routine to repeat the allocation attempt after performing garbage collection. It omits these steps when the **M_DONTWAIT** value is used.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. Likewise, the returned ID is valid in the kernel protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

CL_BLK_ID or a **NULL** if no **clBlk** was available.

SEE ALSO

netBufLib

netClBlkJoin( )

NAME

netClBlkJoin( ) – join a cluster to a **clBlk** structure

SYNOPSIS

```
CL_BLK_ID netClBlkJoin
    (
    CL_BLK_ID pClBlk,          /* pointer to a cluster Blk */
    char *    pClBuf,          /* pointer to a cluster buffer */
    int       size,            /* size of the cluster buffer */
```

```
    FUNCPTR   pFreeRtn,       /* pointer to the free routine */
    int       arg1,           /* argument 1 of the free routine */
    int       arg2,           /* argument 2 of the free routine */
    int       arg3            /* argument 3 of the free routine */
    )
```

DESCRIPTION This routine joins the previously reserved cluster specified by *pClBuf* to the previously reserved **clBlk** structure specified by *pClBlk*. The *size* parameter passes in the size of the cluster referenced in *pClBuf*. The arguments *pFreeRtn*, *arg1*, *arg2*, *arg3* set the values of the **pCLFreeRtn**, **clFreeArg1**, **clFreeArg2**, and **clFreeArg1**, members of the specified **clBlk** structure.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. Likewise, the returned ID is valid in the kernel protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS **CL_BLK_ID** or **NULL**.

SEE ALSO **netBufLib**

N

netClFree( )

NAME **netClFree( )** – free a cluster back to the memory pool

SYNOPSIS

```
void netClFree
    (
    NET_POOL_ID pNetPool,     /* pointer to the net pool */
    UCHAR *     pClBuf        /* pointer to the cluster buffer */
    )
```

DESCRIPTION This routine returns the specified cluster buffer back to the specified memory pool.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS N/A

SEE ALSO **netBufLib**

netClPoolIdGet( )

NAME **netClPoolIdGet( )** – return a **CL_POOL_ID** for a specified buffer size

SYNOPSIS

```
CL_POOL_ID netClPoolIdGet
    (
    NET_POOL_ID pNetPool,     /* pointer to the net pool */
    int         bufSize,      /* size of the buffer */
    BOOL        bestFit       /* TRUE/FALSE */
    )
```

DESCRIPTION This routine returns a **CL_POOL_ID** for a cluster pool containing clusters that match the specified *bufSize*. If *bestFit* is **TRUE**, this routine returns a **CL_POOL_ID** for a pool that contains clusters greater than or equal to *bufSize*. If *bestFit* is **FALSE**, this routine returns a **CL_POOL_ID** for a cluster from whatever cluster pool is available. If the memory pool specified by *pNetPool* contains only one cluster pool, *bestFit* should always be **FALSE**.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. Likewise, the returned ID is valid in the kernel protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS **CL_POOL_ID** or **NULL**.

SEE ALSO **netBufLib**

netClusterGet( )

NAME **netClusterGet( )** – get a cluster from the specified cluster pool

SYNOPSIS

```
char * netClusterGet
    (
    NET_POOL_ID pNetPool,     /* pointer to the net pool */
    CL_POOL_ID  pClPool       /* ptr to the cluster pool */
    )
```

DESCRIPTION This routine gets a cluster from the specified cluster pool *pClPool* within the specified memory pool *pNetPool*.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. Likewise, the returned ID is valid in the kernel protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS This routine returns a character pointer to a cluster buffer or **NULL** if none was available.

SEE ALSO **netBufLib**

netDevCreate( )

NAME **netDevCreate( )** – create a remote file device

SYNOPSIS

```
STATUS netDevCreate
    (
    char * devName,       /* name of device to create */
    char * host,          /* host this device will talk to */
    int    protocol       /* remote file access protocol 0 = RSH, 1 = FTP */
    )
```

DESCRIPTION This routine creates a remote file device. Normally, a network device is created for each remote machine whose files are to be accessed. By convention, a network device name is the remote machine name followed by a colon ":". For example, for a UNIX host on the network whose name is “wrs”, files can be accessed by creating a device called “wrs:”. Files can be accessed via RSH as follows:

```
netDevCreate ("wrs:", "wrs", rsh);
```

The file **/usr/dog** on the UNIX system “wrs” can now be accessed as “**wrs:/usr/dog**” via RSH.

Before creating a device, the host must have already been created with **hostAdd( )**.

RETURNS **OK** or **ERROR**.

SEE ALSO **netDrv**, **hostAdd( )**

netDevCreate2( )

NAME **netDevCreate2( )** – create a remote file device with fixed buffer size

SYNOPSIS

```
STATUS netDevCreate2
    (
    char * devName,       /* name of device to create */
    char * host,          /* host this device will talk to */
    int    protocol,      /* remote file access protocol 0 = RSH, 1 = FTP */
    UINT   bufSize        /* size of buffer in NET_FD */
    )
```

DESCRIPTION This routine creates a remote file device, just like **netDevCreate( )**, but it allows very large files to be accessed without loading the entire file to memory. The fourth parameter *bufSize* specifies the amount of memory. If *bufSize* is zero, the behavior is exactly the same as **netDevCreate( )**. If *bufSize* is not zero, the following restrictions apply:

- **O_RDONLY**, **O_WRONLY** open mode are supported, but not **O_RDWR** open mode.
- seek is supported in **O_RDONLY** open mode, but not in **O_WRONLY** open mode.
- backward seek might be slow if it is beyond the buffer.

RETURNS **OK** or **ERROR**.

SEE ALSO **netDrv**, **netDevCreate( )**

netDrv( )

NAME **netDrv( )** – install the network remote file driver

SYNOPSIS

```
STATUS netDrv (void)
```

DESCRIPTION This routine initializes and installs the network driver. It must be called before other network remote file functions are performed. It is called automatically when **INCLUDE_NET_DRV** is defined.

VXWORKS AE PROTECTION DOMAINS
Under VxWorks AE, you can call this function from within the kernel protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS **OK** or **ERROR**.

SEE ALSO **netDrv**

N

netDrvDebugLevelSet( )

NAME **netDrvDebugLevelSet( )** – set the debug level of the **netDrv** library routines

SYNOPSIS

```
STATUS netDrvDebugLevelSet
    (
    UINT32 debugLevel          /* NETDRV_DEBUG_OFF, NETDRV_DEBUG_ERRORS, */
                               /* NETDRV_DEBUG_ALL */
    )
```

DESCRIPTION This routine enables the debugging of calls to the net driver. The argument **NETLIB_DEBUG_ERRORS** will display only error messages to the console. The argument **NETLIB_DEBUG_ALL** will display warnings and errors to the console.

RETURNS **OK**, or **ERROR** if the debug level is invalid

SEE ALSO **netDrv**

netDrvFileDoesNotExistInstall( )

NAME

netDrvFileDoesNotExistInstall( ) – install an applette to test if a file exists

SYNOPSIS

```
STATUS netDrvFileDoesNotExistInstall
    (
    FUNCPTR pAppletteRtn /* function that returns TRUE or FALSE */
    )
```

DESCRIPTION

Install a function to test if a file exists. *pAppletteRtn* should be of the following format:

```
STATUS appletteRoutine
    (
    char *filename, /* filename queried */
    char *response /* server response string */
    )
```

The **netDrv( )** routine calls the applette during an open with O_CREAT. The system performs an NLST command and uses the applette to parse the response. The routine compensates for server response implementation variations. The applette should return **OK** if the file is not found and **ERROR** if the file is found.

RETURNS

OK, installation successful; **ERROR**, installation error.

SEE ALSO

netDrv, **open( )**

netHelp( )

NAME

netHelp( ) – print a synopsis of network routines

SYNOPSIS

```
void netHelp (void)
```

DESCRIPTION

This command prints a brief synopsis of network facilities typically called from the shell.

```
hostAdd      "hostname","inetaddr" - add a host to remote host table;
                                     "inetaddr" must be in standard
                                     Internet address format e.g. "90.0.0.4"
hostShow                           - print current remote host table
netDevCreate "devname","hostname",protocol
                                   - create an I/O device to access
                                     files on the specified host
                                     (protocol 0=rsh, 1=ftp)
```

```
routeAdd      "destaddr","gateaddr" - add route to route table
routeDelete   "destaddr","gateaddr" - delete route from route table
routeShow                           - print current route table
iam           "usr"[,"passwd"]      - specify the user name by which
                                      you will be known to remote
                                      hosts (and optional password)
whoami                              - print the current remote ID
rlogin        "host"                - log in to a remote host;
                                      "host" can be inet address or
                                      host name in remote host table
ifShow        ["ifname"]            - show info about network interfaces
inetstatShow                        - show all Internet protocol sockets
tcpstatShow                         - show statistics for TCP
udpstatShow                         - show statistics for UDP
ipstatShow                          - show statistics for IP
icmpstatShow                        - show statistics for ICMP
arptabShow                          - show a list of known ARP entries
mbufShow                            - show mbuf statistics
```

RETURNS
N/A

SEE ALSO
usrLib, *VxWorks Programmer's Guide: Target Shell*

N

netLibInit( )

NAME
netLibInit( ) – initialize the network package

SYNOPSIS
```
STATUS netLibInit (void)
```

DESCRIPTION
This creates the network task job queue, and spawns the network task **netTask( )**. It should be called once to initialize the network. This is done automatically when **INCLUDE_NET_LIB** is defined.

VXWORKS AE PROTECTION DOMAINS
Under VxWorks AE, you can call this function from within the kernel protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS
OK, or **ERROR** if network support cannot be initialized.

SEE ALSO
netLib, **usrConfig**, **netTask( )**

netMblkChainDup( )

NAME **netMblkChainDup( )** – duplicate an **mBlk** chain

SYNOPSIS

```
M_BLK_ID netMblkChainDup
    (
    NET_POOL_ID pNetPool,      /* pointer to the pool */
    M_BLK_ID    pMblk,         /* pointer to source mBlk chain*/
    int         offset,        /* offset to duplicate from */
    int         len,           /* length to copy */
    int         canWait        /* M_DONTWAIT/M_WAIT */
    )
```

DESCRIPTION This routine makes a copy of an **mBlk** chain starting at *offset* bytes from the beginning of the chain and continuing for *len* bytes. If *len* is **M_COPYALL**, then this routine will copy the entire **mBlk** chain from the *offset*.

This routine copies the references from a source *pMblk* chain to a newly allocated **mBlk** chain. This lets the two **mBlk** chains share the same **clBlk**-cluster constructs. This routine also increments the reference count in the shared **clBlk**. The *pMblk* expects a pointer to the source **mBlk** chain. The *pNetPool* parameter expects a pointer to the netPool from which the new **mBlk** chain is allocated.

The *canWait* parameter determines the behavior if any required **mBlk** is not immediately available. A value of **M_WAIT** allows this routine to repeat the allocation attempt after performing garbage collection. The **M_DONTWAIT** value prevents those extra steps.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. Likewise, the returned ID is valid in the kernel protection domain only. This restriction does not apply under non-AE versions of VxWorks.

SEE ALSO **netBufLib**, **netMblkDup( )**

RETURNS A pointer to the newly allocated **mBlk** chain or **NULL**.

ERRNO **S_netBufLib_INVALID_ARGUMENT**
S_netBufLib_NO_POOL_MEMORY

netMblkClChainFree( )

NAME **netMblkClChainFree( )** – free a chain of **mBlk-clBlk**-cluster constructs

SYNOPSIS

```
void netMblkClChainFree
    (
    M_BLK_ID pMblk            /* pointer to the mBlk */
    )
```

DESCRIPTION For the specified chain of **mBlk-clBlk**-cluster constructs, this routine frees all the **mBlk** structures back to the specified memory pool. It also decrements the reference count in all the **clBlk** structures. If the reference count in a **clBlk** falls to zero, this routine also frees that **clBlk** and its associated cluster back to the specified memory pool.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS N/A

ERRNO **S_netBufLib_MBLK_INVALID**

SEE ALSO **netBufLib**

netMblkClFree( )

NAME **netMblkClFree( )** – free an **mBlk-clBlk**-cluster construct

SYNOPSIS

```
M_BLK_ID netMblkClFree
    (
    M_BLK_ID pMblk            /* pointer to the mBlk */
    )
```

DESCRIPTION For the specified **mBlk-clBlk**-cluster construct, this routine frees the **mBlk** back to the specified memory pool. It also decrements the reference count in the **clBlk** structure. If the reference count falls to zero, no other **mBlk** structure reference this **clBlk**. In that case, this routine also frees the **clBlk** structure and its associated cluster back to the specified memory pool.

N

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. Likewise, the returned ID is valid in the kernel protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

If the specified **mBlk** was part of an **mBlk** chain, this routine returns a pointer to the next **mBlk**. Otherwise, it returns a **NULL**.

ERRNO

S_netBufLib_MBLK_INVALID

SEE ALSO

netBufLib

netMblkClGet( )

NAME

netMblkClGet( ) – get a **clBlk**-cluster and join it to the specified **mBlk**

SYNOPSIS

```
STATUS netMblkClGet
    (
    NET_POOL_ID pNetPool,     /* pointer to the net pool */
    M_BLK_ID    pMblk,        /* mBlk to embed the cluster in */
    int         bufSize,      /* size of the buffer to get */
    int         canWait,      /* wait or dontwait */
    BOOL        bestFit       /* TRUE/FALSE */
    )
```

DESCRIPTION

This routine gets a **clBlk**-cluster pair from the specified memory pool and joins it to the specified **mBlk** structure. The **mBlk**-**clBlk**-cluster triplet it produces is the basic structure for handling data at all layers of the network stack.

pNetPool
: Expects a pointer to the memory pool from which you want to get a free **clBlk**-cluster pair.

pMbkl
: Expects a pointer to the **mBlk** structure (previously allocated) to which you want to join the retrieved **clBlk**-cluster pair.

bufSize
: Expects the size, in bytes, of the cluster in the **clBlk**-cluster pair.

canWait
Expects either **M_WAIT** or **M_DONTWAIT**. If either item is not immediately available, the **M_WAIT** value allows this routine to repeat the allocation attempt after performing garbage collection. It omits those steps when the **M_DONTWAIT** value is used.

bestFit
Expects either **TRUE** or **FALSE**. If *bestFit* is **TRUE** and a cluster of the exact size is unavailable, this routine gets a larger cluster (if available). If *bestFit* is **FALSE** and an exact size cluster is unavailable, this routine gets either a smaller or a larger cluster (depending on what is available). Otherwise, it returns immediately with an **ERROR** value. For memory pools containing only one cluster size, *bestFit* should always be set to **FALSE**.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS **OK** or **ERROR**.

ERRNO **S_netBufLib_CLSIZE_INVALID**

SEE ALSO **netBufLib**

netMblkClJoin( )

NAME **netMblkClJoin( )** – join an **mBlk** to a **clBlk**-cluster construct

SYNOPSIS

```
M_BLK_ID netMblkClJoin
    (
    M_BLK_ID  pMblk,          /* pointer to an mBlk */
    CL_BLK_ID pClBlk          /* pointer to a cluster Blk */
    )
```

DESCRIPTION This routine joins the previously reserved **mBlk** referenced in *pMblk* to the **clBlk**-cluster construct referenced in *pClBlk*. Internally, this routine sets the **M_EXT** flag in **mBlk.mBlkHdr.mFlags**. It also and sets the **mBlk.mBlkHdr.mData** to point to the start of the data in the cluster.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. Likewise, the returned ID is valid in the kernel protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

M_BLK_ID or **NULL**.

SEE ALSO

netBufLib

netMblkDup( )

NAME

netMblkDup( ) – duplicate an **mBlk**

SYNOPSIS

```
M_BLK_ID netMblkDup
    (
    M_BLK_ID pSrcMblk,        /* pointer to source mBlk */
    M_BLK_ID pDestMblk        /* pointer to the destination mBlk */
    )
```

DESCRIPTION

This routine copies the references from a source **mBlk** in an **mBlk-clBlk**-cluster construct to a stand-alone **mBlk**. This lets the two **mBlk** structures share the same **clBlk**-cluster construct. This routine also increments the reference count in the shared **clBlk**. The *pSrcMblk* expects a pointer to the source **mBlk**. The *pDescMblk* parameter expects a pointer to the destination **mBlk**.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. Likewise, the returned ID is valid in the kernel protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

A pointer to the destination **mBlk** or **NULL** if the source **mBlk** referenced in *pSrcMblk* is not part of a valid **mBlk-clBlk**-cluster construct.

SEE ALSO

netBufLib

netMblkFree( )

NAME

netMblkFree( ) – free an **mBlk** back to its memory pool

SYNOPSIS

```
void netMblkFree
    (
    NET_POOL_ID pNetPool,     /* pointer to the net pool */
    M_BLK_ID    pMblk         /* mBlk to free */
    )
```

DESCRIPTION

This routine frees the specified **mBlk** back to the specified memory pool.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

N/A

SEE ALSO

netBufLib

N

netMblkGet( )

NAME

netMblkGet( ) – get an **mBlk** from a memory pool

SYNOPSIS

```
M_BLK_ID netMblkGet
    (
    NET_POOL_ID pNetPool,     /* pointer to the net pool */
    int         canWait,      /* M_WAIT/M_DONTWAIT */
    UCHAR       type          /* mBlk type */
    )
```

DESCRIPTION

This routine allocates an **mBlk** from the specified memory pool, if available.

pNetPool
Expects a pointer to the pool from which you want an **mBlk**.

canWait
Expects either **M_WAIT** or **M_DONTWAIT**. If no **mBlk** is immediately available, the

M_WAIT value allows this routine to repeat the allocation attempt after performing garbage collection. It omits these steps when the **M_DONTWAIT** value is used.

type
Expects the type value that you want to associate with the returned **mBlk**.

VXWORKS AE PROTECTION DOMAINS
Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. Likewise, the returned ID is valid in the kernel protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS **M_BLK_ID** or **NULL** if no **mBlk** is available.

ERRNO **S_netBufLib_MBLK_INVALID**

SEE ALSO **netBufLib**

netMblkToBufCopy( )

NAME **netMblkToBufCopy( )** – copy data from an **mBlk** to a buffer

SYNOPSIS

```
int netMblkToBufCopy
    (
    M_BLK_ID pMblk,           /* pointer to an mBlk */
    char *   pBuf,            /* pointer to the buffer to copy */
    FUNCPTR  pCopyRtn         /* function pointer for copy routine */
    )
```

DESCRIPTION This routine copies data from the **mBlk** chain referenced in *pMblk* to the buffer referenced in *pBuf*. It is assumed that *pBuf* points to enough memory to contain all the data in the entire **mBlk** chain. The argument *pCopyRtn* expects either a **NULL** or a function pointer to a copy routine. The arguments passed to the copy routine are source pointer, destination pointer and the length of data to copy. If *pCopyRtn* is **NULL**, **netMblkToBufCopy( )** uses a default routine to extract the data from the chain.

RETURNS The length of data copied or zero.

SEE ALSO **netBufLib**

netPoolDelete( )

NAME **netPoolDelete( )** – delete a memory pool

SYNOPSIS

```
STATUS netPoolDelete
    (
    NET_POOL_ID pNetPool      /* pointer to a net pool */
    )
```

DESCRIPTION This routine deletes the specified **netBufLib**-managed memory pool.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS **OK** or **ERROR**.

ERRNO **S_netBufLib_NETPOOL_INVALID**

SEE ALSO **netBufLib**

N

netPoolInit( )

NAME **netPoolInit( )** – initialize a **netBufLib**-managed memory pool

SYNOPSIS

```
STATUS netPoolInit
    (
    NET_POOL_ID   pNetPool,        /* pointer to a net pool */
    M_CL_CONFIG * pMclBlkConfig,   /* pointer to a mBlk configuration */
    CL_DESC *     pClDescTbl,      /* pointer to cluster desc table */
    int           clDescTblNumEnt, /* number of cluster desc entries */
    POOL_FUNC *   pFuncTbl         /* pointer to pool function table */
    )
```

DESCRIPTION Call this routine to set up a **netBufLib**-managed memory pool. Within this pool, **netPoolInit( )** organizes several sub-pools: one for **mBlk** structures, one for **clBlk** structures, and as many cluster sub-pools are there are cluster sizes. As input, this routine expects the following parameters:

pNetPool
Expects a **NET_POOL_ID** that points to a previously allocated **NET_POOL** structure. You need not initialize any values in this structure. That is handled by **netPoolInit( )**.

pMclBlkConfig
Expects a pointer to a previously allocated and initialized **M_CL_CONFIG** structure. Within this structure, you must provide four values: **mBlkNum**, a count of **mBlk** structures; **clBlkNum**, a count of **clBlk** structures; **memArea**, a pointer to an area of memory that can contain all the **mBlk** and **clBlk** structures; and **memSize**, the size of that memory area. For example, you can set up an **M_CL_CONFIG** structure as follows:

```
M_CL_CONFIG mClBlkConfig = /* mBlk, clBlk configuration table */
    {
    mBlkNum     clBlkNum        memArea         memSize
    ----------  ----            -------         -------
    400,        245,            0xfe000000,     21260
    };
```

You can calculate the **memArea** and **memSize** values. Such code could first define a table as shown above, but set both **memArea** and **memSize** as follows:

```
mClBlkConfig.memSize = (mClBlkConfig.mBlkNum * (M_BLK_SZ + sizeof(long))) +
                       (mClBlkConfig.clBlkNum * CL_BLK_SZ);
```

You can set the **memArea** value to a pointer to private memory, or you can reserve the memory with a call to **malloc( )**. For example:

```
mClBlkConfig.memArea = malloc(mClBlkConfig.memSize);
```

The **netBufLib.h** file defines **M_BLK_SZ** as:

```
sizeof(struct mBlk)
```

Currently, this evaluates to 32 bytes. Likewise, this file defines **CL_BLK_SZ** as:

```
sizeof(struct clBlk)
```

Currently, this evaluates to 32 bytes.

When choosing values for **mBlkNum** and **clBlkNum**, remember that you need as many **clBlk** structures as you have clusters (data buffers). You also need at least as many **mBlk** structures as you have **clBlk** structures, but you will most likely need more. That is because **netBufLib** shares buffers by letting multiple **mBlk** structures join to the same **clBlk** and thus to its underlying cluster. The **clBlk** keeps a count of the number of **mBlk** structures that reference it.

pClDescTbl
Expects a pointer to a table of previously allocated and initialized **CL_DESC** structures. Each structure in this table describes a single cluster pool. You need a dedicated cluster pool for each cluster size you want to support. Within each **CL_DESC** structure, you must provide four values: **clusterSize**, the size of a cluster in

this cluster pool; **num**, the number of clusters in this cluster pool; **memArea**, a pointer to an area of memory that can contain all the clusters; and **memSize**, the size of that memory area.

Thus, if you need to support six different cluster sizes, this parameter must point to a table containing six **CL_DESC** structures. For example, consider the following:

```
CL_DESC clDescTbl [] =   /* cluster descriptor table */
    {
    /*
    clusterSize        num     memArea         memSize
    ----------         ----    -------         -------
    */
    {64,               100,    0x10000,        6800},
    {128,              50,     0x20000,        6600},
    {256,              50,     0x30000,        13000},
    {512,              25,     0x40000,        12900},
    {1024,             10,     0x50000,        10280},
    {2048,             10,     0x60000,        20520}
    };
```

As with the **memArea** and **memSize** members in the **M_CL_CONFIG** structure, you can set these members of the **CL_DESC** structures by calculation after you create the table. The formula would be as follows:

```
clDescTbl[n].memSize =
    (clDescTbl[n].num * (clDescTbl[n].clusterSize + sizeof(long)));
```

The **memArea** member can point to a private memory area that you know to be available for storing clusters, or you can use **malloc( )**.

```
clDescTbl[n].memArea =  malloc( clDescTbl[n].memSize );
```

Valid cluster sizes range from 64 bytes to 65536 bytes. If there are multiple cluster pools, valid sizes are further restricted to powers of two (for example, 64, 128, 256, and so on). If there is only one cluster pool (as is often the case for the memory pool specific to a single device driver), there is no power of two restriction. Thus, the cluster can be of any size between 64 bytes and 65536 bytes on 4-byte alignment. A typical buffer size for Ethernet devices is 1514 bytes. However, because a cluster size requires a 4-byte alignment, the cluster size for this Ethernet buffer would have to be increased to at least 1516 bytes.

clDescTblNumEnt
Expects a count of the elements in the **CL_DESC** table referenced by the *pClDescTbl* parameter. This is a count of the number of cluster pools. You can get this value using the NELEMENTS macro defined in **vxWorks.h**. For example:

```
int clDescTblNumEnt = (NELEMENTS(clDescTbl));
```

pFuncTbl
Expects a **NULL** or a pointer to a function table. This table contains pointers to the

functions used to manage the buffers in this memory pool. Using a **NULL** for this parameter tells **netBufLib** to use its default function table. If you opt for the default function table, every **mBlk** and every cluster is prepended by a 4-byte header (which is why the size calculations above for clusters and **mBlk** structures contained an extra **sizeof(long)**). However, users need not concern themselves with this header when accessing these buffers. The returned pointers from functions such as **netClusterGet( )** return pointers to the start of data, which is just after the header.

Assuming you have set up the configuration tables as shown above, a typical call to **netPoolInit( )** would be as follows:

```
    int clDescTblNumEnt = (NELEMENTS(clDescTbl));
    NET_POOL  netPool;
    NET_POOL_ID       pNetPool = &netPool;
    if (netPoolInit (pNetPool, &mClBlkConfig, &clDescTbl [0],
clDescTblNumEnt,
        NULL) != OK)
        return (ERROR);
```

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, access to the contents of a memory pool is limited to the protection domain within which you made the **netPoolInit( )** call that created the pool. In addition, all parameters to a **netPoolInit( )** call must be valid within the protection domain from which you make the call. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

OK or **ERROR**.

ERRNO

S_netBufLib_MEMSIZE_INVALID
S_netBufLib_CLSIZE_INVALID
S_netBufLib_NO_SYSTEM_MEMORY
S_netBufLib_MEM_UNALIGNED
S_netBufLib_MEMSIZE_UNALIGNED
S_netBufLib_MEMAREA_INVALID

SEE ALSO

netBufLib, **netPoolDelete( )**

netPoolKheapInit( )

NAME **netPoolKheapInit( )** – kernel heap version of **netPoolInit( )**

SYNOPSIS

```
STATUS netPoolKheapInit
    (
    NET_POOL_ID   pNetPool,        /* pointer to a net pool */
    M_CL_CONFIG * pMclBlkConfig,   /* pointer to a mBlk configuration */
    CL_DESC *     pClDescTbl,      /* pointer to cluster desc table */
    int           clDescTblNumEnt, /* number of cluster desc entries */
    POOL_FUNC *   pFuncTbl         /* pointer to pool function table */
    )
```

DESCRIPTION This initializes a **netBufLib**-managed memory pool from Kernel heap. See **netPoolInit( )** for more detail.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS **OK** or **ERROR**.

ERRNO N/A

SEE ALSO **netBufLib**, **netPoolInit( )**, **netPoolDelete( )**

netPoolShow( )

NAME **netPoolShow( )** – show pool statistics

SYNOPSIS

```
void netPoolShow
    (
    NET_POOL_ID pNetPool
    )
```

DESCRIPTION This routine displays the distribution of **mBlk**s and clusters in a given network pool ID.

EXAMPLE

```
void endPoolShow
   (
   char * devName,     /* The inteface name: "dc", "ln" ...*/
   int    unit         /* the unit number: usually 0       */
   )
   {
   END_OBJ * pEnd;
   if ((pEnd = endFindByName (devName, unit)) != NULL)
       netPoolShow (pEnd->pNetPool);
   else
       printf ("Could not find device %s\n", devName);
   return;
   }
```

RETURNS N/A

SEE ALSO **netShow**

netShowInit( )

NAME **netShowInit( )** – initialize network show routines

SYNOPSIS

```
void netShowInit (void)
```

DESCRIPTION This routine links the network show facility into the VxWorks system. These routines are included automatically if **INCLUDE_NET_SHOWis** defined.

RETURNS N/A

SEE ALSO **netShow**

netStackDataPoolShow( )

NAME **netStackDataPoolShow( )** – show network stack data pool statistics

SYNOPSIS
```
void netStackDataPoolShow (void)
```

DESCRIPTION This routine displays the distribution of **mBlk**s and clusters in a the network data pool. The network data pool is used only for data transfer through the network stack.

The "clusters" column indicates the total number of clusters of that size that have been allocated. The "free" column indicates the number of available clusters of that size (the total number of clusters minus those clusters that are in use). The "usage" column indicates the number of times clusters have been allocated (not, as you might expect, the number of clusters currently in use).

RETURNS N/A

SEE ALSO **netShow**, **netStackSysPoolShow( )**, **netBufLib**

N

netStackSysPoolShow( )

NAME **netStackSysPoolShow( )** – show network stack system pool statistics

SYNOPSIS
```
void netStackSysPoolShow (void)
```

DESCRIPTION This routine displays the distribution of **mBlk**s and clusters in a the network system pool. The network system pool is used only for system structures such as sockets, routes, interface addresses, protocol control blocks, multicast addresses, and multicast route entries.

The "clusters" column indicates the total number of clusters of that size that have been allocated. The "free" column indicates the number of available clusters of that size (the total number of clusters minus those clusters that are in use). The "usage" column indicates the number of times clusters have been allocated (not, as you might expect, the number of clusters currently in use).

RETURNS N/A

SEE ALSO **netShow**, **netStackDataPoolShow( )**, **netBufLib**

netTask( )

NAME **netTask( )** – network task entry point

SYNOPSIS

```
void netTask (void)
```

DESCRIPTION This routine is the VxWorks network support task. Most of the VxWorks network runs in this task's context.

NOTE: To prevent an application task from monopolizing the CPU if it is in an infinite loop or is never blocked, the priority of **netTask( )** relative to an application may need to be adjusted. Network communication may be lost if **netTask( )** is "starved" of CPU time. The default task priority of **netTask( )** is 50. Use **taskPrioritySet( )** to change the priority of a task.

This task is spawned by **netLibInit( )**.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS N/A

SEE ALSO **netLib**, **netLibInit( )**

netTupleGet( )

NAME **netTupleGet( )** – get an **mBlk-clBlk**-cluster

SYNOPSIS

```
M_BLK_ID netTupleGet
    (
    NET_POOL_ID pNetPool,     /* pointer to the net pool */
    int         bufSize,      /* size of the buffer to get */
    int         canWait,      /* wait or dontwait */
    UCHAR       type,         /* type of data */
    BOOL        bestFit       /* TRUE/FALSE */
    )
```

DESCRIPTION This routine gets an **mBlk-clBlk**-cluster triplet from the specified memory pool. The resulting structure is the basic method for accessing data at all layers of the network stack.

pNetPool
Expects a pointer to the memory pool with which you want to build a **mBlk-clBlk**-cluster triplet.

bufSize
Expects the size, in bytes, of the cluster in the **clBlk**-cluster pair.

canWait
Expects either **M_WAIT** or **M_DONTWAIT**. If any item in the triplet is not immediately available, the **M_WAIT** value allows this routine to repeat the allocation attempt after performing garbage collection. The **M_DONTWAIT** value prevents those extra steps.

type
Expects the type of data, for example **MT_DATA**, **MT_HEADER**. The various values for this type are defined in **netBufLib.h**.

bestFit
Expects either **TRUE** or **FALSE**. If *bestFit* is **TRUE** and a cluster of the exact size is unavailable, this routine gets a larger cluster (if available). If *bestFit* is **FALSE** and an exact size cluster is unavailable, this routine gets either a smaller or a larger cluster (depending on what is available). Otherwise, it returns immediately with an **ERROR** value. For memory pools containing only one cluster size, *bestFit* should always be set to **FALSE**.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. Likewise, the returned ID is valid in the kernel protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

M_BLK_ID or **NULL**.

ERRNO

S_netBufLib_MBLK_INVALID
S_netBufLib_CLSIZE_INVALID
S_netBufLib_NETPOOL_INVALID

SEE ALSO

netBufLib

nextIndex( )

NAME **nextIndex( )** – the comparison routine for the AVL tree

SYNOPSIS

```
int nextIndex
    (
    void *           pAvlNode, /* The node to compare with */
    GENERIC_ARGUMENT key       /* The given index */
    )
```

DESCRIPTION This routine compares the two indexes and returns a code based on wether the index, in question, is lesser than, equal to or greater than the one being compared.

RETURNS -1, if the given index is lesser; 0, if equal; and 1, if greater.

SEE ALSO **m2IfLib**

nfsAuthUnixGet( )

NAME **nfsAuthUnixGet( )** – get the NFS UNIX authentication parameters

SYNOPSIS

```
void nfsAuthUnixGet
    (
    char * machname,          /* where to store host machine */
    int *  pUid,              /* where to store user ID */
    int *  pGid,              /* where to store group ID */
    int *  pNgids,            /* where to store number of group IDs */
    int *  gids               /* where to store array of group IDs */
    )
```

DESCRIPTION This routine gets the previously set UNIX authentication values.

RETURNS N/A

SEE ALSO **nfsLib**, **nfsAuthUnixPrompt( )**, **nfsAuthUnixShow( )**, **nfsAuthUnixSet( )**, **nfsIdSet( )**

nfsAuthUnixPrompt( )

NAME **nfsAuthUnixPrompt( )** – modify the NFS UNIX authentication parameters

SYNOPSIS
```
void nfsAuthUnixPrompt (void)
```

DESCRIPTION This routine allows UNIX authentication parameters to be changed from the shell. The user is prompted for each parameter, which can be changed by entering the new value next to the current one.

EXAMPLE
```
-> nfsAuthUnixPrompt
machine name:   yuba
user ID:        2001 128
group ID:       100
num of groups:  1 3
group #1:        100 100
group #2:        0 120
group #3:        0 200
value = 3 = 0x3
```

SEE ALSO **nfsLib**, **nfsAuthUnixShow( )**, **nfsAuthUnixSet( )**, **nfsAuthUnixGet( )**, **nfsIdSet( )**

N

nfsAuthUnixSet( )

NAME **nfsAuthUnixSet( )** – set the NFS UNIX authentication parameters

SYNOPSIS
```
void nfsAuthUnixSet
    (
    char * machname,          /* host machine */
    int    uid,               /* user ID */
    int    gid,               /* group ID */
    int    ngids,             /* number of group IDs */
    int *  aup_gids           /* array of group IDs */
    )
```

DESCRIPTION This routine sets UNIX authentication parameters. It is initially called by **usrNetInit( )**. *machname* should be set with the name of the mounted system (i.e., the target name itself) to distinguish hosts from hosts on a NFS network.

RETURNS N/A

SEE ALSO **nfsLib**, **nfsAuthUnixPrompt( )**, **nfsAuthUnixShow( )**, **nfsAuthUnixGet( )**, **nfsIdSet( )**

nfsAuthUnixShow( )

NAME **nfsAuthUnixShow( )** – display the NFS UNIX authentication parameters

SYNOPSIS

```
void nfsAuthUnixShow (void)
```

DESCRIPTION This routine displays the parameters set by **nfsAuthUnixSet( )** or **nfsAuthUnixPrompt( )**.

EXAMPLE

```
-> nfsAuthUnixShow
machine name = yuba
user ID      = 2001
group ID     = 100
group [0]    = 100
value = 1 = 0x1
```

RETURNS N/A

SEE ALSO **nfsLib**, **nfsAuthUnixPrompt( )**, **nfsAuthUnixSet( )**, **nfsAuthUnixGet( )**, **nfsIdSet( )**

nfsDevInfoGet( )

NAME **nfsDevInfoGet( )** – read configuration information from the requested NFS device

SYNOPSIS

```
STATUS nfsDevInfoGet
    (
    unsigned long  nfsDevHandle, /* NFS device handle */
    NFS_DEV_INFO * pnfsInfo      /* ptr to struct to hold config info */
    )
```

DESCRIPTION This routine accesses the NFS device specified in the parameter *nfsDevHandle*and fills in the structure pointed to by *pnfsInfo*. The calling function should allocate memory for *pnfsInfo* and for the two character buffers, **remFileSys** and **locFileSys**, that are part of *pnfsInfo*. These buffers should have a size of **nfsMaxPath**.

RETURNS **OK** if *pnfsInfo* information is valid, otherwise **ERROR**.

SEE ALSO **nfsDrv**, **nfsDevListGet( )**

nfsDevListGet( )

NAME **nfsDevListGet( )** – create list of all the NFS devices in the system

SYNOPSIS

```
int nfsDevListGet
    (
    unsigned long nfsDevList[], /* NFS dev list of handles */
    int           listSize      /* number of elements available in list */
    )
```

DESCRIPTION This routine fills the array *nfsDevlist* up to *listSize*, with handles to NFS devices currently in the system.

RETURNS The number of entries filled in the *nfsDevList* array.

SEE ALSO **nfsDrv**, **nfsDevInfoGet( )**

nfsDevShow( )

NAME **nfsDevShow( )** – display the mounted NFS devices

SYNOPSIS

```
void nfsDevShow (void)
```

DESCRIPTION This routine displays the device names and their associated NFS file systems.

EXAMPLE

```
-> nfsDevShow
device name          file system
-----------          -----------
/yuba1/              yuba:/yuba1
/wrs1/               wrs:/wrs1
```

RETURNS N/A

SEE ALSO **nfsDrv**

N

nfsdInit( )

NAME

nfsdInit( ) – initialize the NFS server

SYNOPSIS

```
STATUS nfsdInit
    (
    int     nServers,         /* the number of NFS servers to create */
    int     nExportedFs,      /* maximum number of exported file systems */
    int     priority,         /* the priority for the NFS servers */
    FUNCPTR authHook,         /* authentication hook */
    FUNCPTR mountAuthHook,    /* authentication hook for mount daemon */
    int     options           /* currently unused */
    )
```

DESCRIPTION

This routine initializes the NFS server. *nServers* specifies the number of tasks to be spawned to handle NFS requests. *priority* is the priority that those tasks will run at. *authHook* is a pointer to an authorization routine. *mountAuthHook* is a pointer to a similar routine, passed to **mountdInit( )**. *options* is provided for future expansion.

Normally, no authorization is performed by either mountd or nfsd. If you want to add authorization, set *authHook* to a function pointer to a routine declared as follows:

```
nfsstat routine
    (
    int                 progNum,       /* RPC program number */
    int                 versNum,       /* RPC program version number */
    int                 procNum,       /* RPC procedure number */
    struct sockaddr_in clientAddr,     /* address of the client */
    NFSD_ARGUMENT *     nfsdArg        /* argument of the call */
    )
```

The *authHook* routine should return **NFS_OK** if the request is authorized, and **NFSERR_ACCES** if not. (**NFSERR_ACCES** is not required; any legitimate NFS error code can be returned.)

See **mountdInit( )** for documentation on *mountAuthHook*. Note that *mountAuthHook* and *authHook* can point to the same routine. Simply use the *progNum*, *versNum*, and *procNum* fields to decide whether the request is an NFS request or a mountd request.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS **OK**, or **ERROR** if the NFS server cannot be started.

SEE ALSO **nfsdLib**, **nfsExport( )**, **mountdInit( )**

---

nfsDrv( )

NAME **nfsDrv( )** – install the NFS driver

SYNOPSIS

```
STATUS nfsDrv (void)
```

DESCRIPTION This routine initializes and installs the NFS driver. It must be called before any reads, writes, or other NFS calls. This is done automatically when **INCLUDE_NFS** is defined.

VXWORKS AE PROTECTION DOMAINS
Under VxWorks AE, you can call this function from within the kernel protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS **OK**, or **ERROR** if there is no room for the driver.

SEE ALSO **nfsDrv**

N

---

nfsDrvNumGet( )

NAME **nfsDrvNumGet( )** – return the IO system driver number for the NFS driver

SYNOPSIS

```
int nfsDrvNumGet (void)
```

DESCRIPTION This routine returns the NFS driver number allocated by **iosDrvInstall( )** during the NFS driver initialization. If the NFS driver has yet to be initialized, or if initialization failed, **nfsDrvNumGet( )** will return **ERROR**.

RETURNS the NFS driver number or **ERROR**

SEE ALSO **nfsDrv**

nfsdStatusGet( )

NAME
: **nfsdStatusGet( )** – get the status of the NFS server

SYNOPSIS

```
STATUS nfsdStatusGet
    (
    NFS_SERVER_STATUS * serverStats /* pointer to status structure */
    )
```

DESCRIPTION
: This routine gets status information about the NFS server.

RETURNS
: **OK**, or **ERROR** if the information cannot be obtained.

SEE ALSO
: **nfsdLib**

nfsdStatusShow( )

NAME
: **nfsdStatusShow( )** – show the status of the NFS server

SYNOPSIS

```
STATUS nfsdStatusShow
    (
    int options                 /* unused */
    )
```

DESCRIPTION
: This routine shows status information about the NFS server.

RETURNS
: **OK**, or **ERROR** if the information cannot be obtained.

SEE ALSO
: **nfsdLib**

nfsExport( )

NAME

nfsExport( ) – specify a file system to be NFS exported

SYNOPSIS

```
STATUS nfsExport
    (
    char * directory,         /* Directory to export - FS must support NFS */
    int    id,                /* ID number for file system */
    BOOL   readOnly,          /* TRUE if file system is exported read-only */
    int    options            /* Reserved for future use - set to 0 */
    )
```

DESCRIPTION

This routine makes a file system available for mounting by a client. The client should be in the local host table (see **hostAdd( )**), although this is not required.

The *id* parameter can either be set to a specific value, or to 0. If it is set to 0, an ID number is assigned sequentially. Every time a file system is exported, it must have the same ID number, or clients currently mounting the file system will not be able to access files.

To display a list of exported file systems, use:

```
-> nfsExportShow "localhost"
```

RETURNS

OK, or **ERROR** if the file system could not be exported.

SEE ALSO

mountLib, **nfsLib**, **nfsExportShow( )**, **nfsUnexport( )**

nfsExportShow( )

NAME

nfsExportShow( ) – display the exported file systems of a remote host

SYNOPSIS

```
STATUS nfsExportShow
    (
    char * hostName           /* host machine to show exports for */
    )
```

DESCRIPTION

This routine displays the file systems of a specified host and the groups that are allowed to mount them.

EXAMPLE

```
-> nfsExportShow "wrs"
/d0               staff
```

N

```
/d1                 staff eng
/d2                 eng
/d3
value = 0 = 0x0
```

RETURNS **OK** or **ERROR**.

SEE ALSO **nfsLib**

nfsHelp( )

NAME **nfsHelp( )** – display the NFS help menu

SYNOPSIS

```
void nfsHelp (void)
```

DESCRIPTION This routine displays a summary of NFS facilities typically called from the shell:

```
nfsHelp                       Print this list
netHelp                       Print general network help list
nfsMount "host","filesystem"[,"devname"]  Create device with
                                file system/directory from host
nfsUnmount "devname"          Remove an NFS device
nfsAuthUnixShow               Print current UNIX authentication
nfsAuthUnixPrompt             Prompt for UNIX authentication
nfsIdSet id                   Set user ID for UNIX authentication
nfsDevShow                    Print list of NFS devices
nfsExportShow "host"          Print a list of NFS file systems which
                                are exported on the specified host
mkdir "dirname"               Create directory
rm "file"                     Remove file
EXAMPLE:  -> hostAdd "wrs", "90.0.0.2"
          -> nfsMount "wrs","/disk0/path/mydir","/mydir/"
          -> cd "/mydir/"
          -> nfsAuthUnixPrompt     /* fill in user ID, etc.     */
          -> ls                    /* list /disk0/path/mydir    */
          -> copy < foo            /* copy foo to standard out  */
          -> ld < foo.o            /* load object module foo.o  */
          -> nfsUnmount "/mydir/"  /* remove NFS device /mydir/ */
```

RETURNS N/A

SEE ALSO **nfsLib**

nfsIdSet( )

NAME

nfsIdSet( ) – set the ID number of the NFS UNIX authentication parameters

SYNOPSIS

```
void nfsIdSet
    (
    int uid                   /* user ID on host machine */
    )
```

DESCRIPTION

This routine sets only the UNIX authentication user ID number. For most NFS permission needs, only the user ID needs to be changed. Set *uid* to the user ID on the NFS server.

RETURNS

N/A

SEE ALSO

nfsLib, **nfsAuthUnixPrompt( )**, **nfsAuthUnixShow( )**, **nfsAuthUnixSet( )**, **nfsAuthUnixGet( )**

nfsMount( )

NAME

nfsMount( ) – mount an NFS file system

SYNOPSIS

```
STATUS nfsMount
    (
    char * host,              /* name of remote host */
    char * fileSystem,        /* name of remote directory to mount */
    char * localName          /* local device name for remote dir (NULL = */
                              /* use fileSystem name) */
    )
```

DESCRIPTION

This routine mounts a remote file system. It creates a local device *localName* for a remote file system on a specified host. The host must have already been added to the local host table with **hostAdd( )**. If *localName* is **NULL**, the local name will be the same as the remote name.

RETURNS

OK, or **ERROR** if the driver is not installed, *host* is invalid, or memory is insufficient.

SEE ALSO

nfsDrv, **nfsUnmount( )**, **hostAdd( )**

nfsMountAll( )

NAME **nfsMountAll( )** – mount all file systems exported by a specified host

SYNOPSIS

```
STATUS nfsMountAll
    (
    char * pHostName,   /* name of remote host */
    char * pClientName, /* name of a client specified in access */
                        /* list, if any */
    BOOL   quietFlag    /* FALSE = print name of each mounted file system */
    )
```

DESCRIPTION This routine mounts the file systems exported by the host *pHostName*which are accessible by *pClientName*. A *pClientName* entry of **NULL** will only mount file systems that are accessible by any client. The **nfsMount( )** routine is called to mount each file system. It creates a local device for each mount that has the same name as the remote file system.

If the *quietFlag* setting is **FALSE**, each file system is printed on standard output after it is mounted successfully.

RETURNS **OK**, or **ERROR** if any mount fails.

SEE ALSO **nfsDrv**, **nfsMount( )**

nfsUnexport( )

NAME **nfsUnexport( )** – remove a file system from the list of exported file systems

SYNOPSIS

```
STATUS nfsUnexport
    (
    char * dirName              /* Name of the directory to unexport */
    )
```

DESCRIPTION This routine removes a file system from the list of file systems exported from the target. Any client attempting to mount a file system that is not exported will receive an error (**NFSERR_ACCESS**).

RETURNS **OK**, or **ERROR** if the file system could not be removed from the exports list.

ERRNO ENOENT

SEE ALSO **mountLib**, **nfsLib**, **nfsExportShow( )**, **nfsExport( )**

nfsUnmount( )

NAME
nfsUnmount( ) – unmount an NFS device

SYNOPSIS
```
STATUS nfsUnmount
    (
    char * localName          /* local of nfs device */
    )
```

DESCRIPTION
This routine unmounts file systems that were previously mounted via NFS.

RETURNS
OK, or **ERROR** if *localName* is not an NFS device or cannot be mounted.

SEE ALSO
nfsDrv, **nfsMount( )**

ntPassFsDevInit( )

NAME
ntPassFsDevInit( ) – associate a device with ntPassFs file system functions

SYNOPSIS
```
void *ntPassFsDevInit
    (
    char * devName            /* device name */
    )
```

DESCRIPTION
This routine associates the name *devName* with the file system and installs it in the I/O System's device table. The driver number used when the device is added to the table is that which was assigned to the ntPassFs library during **ntPassFsInit( )**.

RETURNS
A pointer to the volume descriptor, or **NULL** if there is an error.

SEE ALSO
ntPassFsLib

ntPassFsInit( )

NAME

ntPassFsInit( ) – prepare to use the ntPassFs library

SYNOPSIS

```
STATUS ntPassFsInit
    (
    int nPassfs               /* number of ntPass-through file systems */
    )
```

DESCRIPTION

This routine initializes the ntPassFs library. It must be called exactly once, before any other routines in the library. The argument specifies the number of ntPassFs devices that may be open at once. This routine installs **ntPassFsLib** as a driver in the I/O system driver table, allocates and sets up the necessary memory structures, and initializes semaphores.

Normally this routine is called from the root task, **usrRoot( )**, in **usrConfig( )**. To enable this initialization, define **INCLUDE_PASSFS** in **configAll.h**.

NOTE: Maximum number of ntPass-through file systems is 1.

RETURNS

OK, or **ERROR**.

SEE ALSO

ntPassFsLib

open( )

NAME **open( )** – open a file

SYNOPSIS

```
int open
    (
    const char * name,        /* name of the file to open */
    int          flags,       /* O_RDONLY, O_WRONLY, O_RDWR, or O_CREAT */
    int          mode         /* mode of file to create (UNIX chmod style) */
    )
```

DESCRIPTION This routine opens a file for reading, writing, or updating, and returns a file descriptor for that file. The arguments to **open( )** are the filename and the type of access:

O_RDONLY (0)	(or READ)	- open for reading only.
O_WRONLY (1)	(or WRITE)	- open for writing only.
O_RDWR (2)	(or UPDATE)	- open for reading and writing.
O_CREAT (0x0200)		- create a file.

In general, **open( )** can only open pre-existing devices and files. However, for NFS network devices only, files can also be created with **open( )** by performing a logical OR operation with **O_CREAT** and the *flags* argument. In this case, the file is created with a UNIX **chmod**-style file mode, as indicated with *mode*. For example:

```
fd = open ("/usr/myFile", O_CREAT | O_RDWR, 0644);
```

Only the NFS driver uses the *mode* argument.

NOTE: For more information about situations when there are no file descriptors available, see the manual entry for **iosInit( )**.

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**

SEE ALSO **ioLib**, **creat( )**

O

opendir( )

NAME

opendir( ) – open a directory for searching (POSIX)

SYNOPSIS

```
DIR *opendir
    (
    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.

SEE ALSO

dirLib, **closedir( )**, **readdir( )**, **rewinddir( )**, **ls( )**

operator delete( )

NAME

operator delete( ) – default run-time support for memory deallocation (C++)

SYNOPSIS

```
extern void operator delete
    (
    void * pMem               /* pointer to dynamically-allocated object */
    )
```

DESCRIPTION

This function provides the default implementation of operator delete. It returns the memory, previously allocated by operator new, to the VxWorks system memory partition.

RETURNS

N/A

SEE ALSO

cplusLib

operator new( )

NAME

operator new( ) – default run-time support for operator new (C++)

SYNOPSIS

```
extern void * operator new
    (
    size_t n                  /* size of object to allocate */
    )
```

DESCRIPTION

This function provides the default implementation of operator new. It allocates memory from the system memory partition for the requested object. The value, when evaluated, is a pointer of the type **pointer-to-***T* where *T* is the type of the new object.

If allocation fails a new-handler, if one is defined, is called. If the new-handler returns, presumably after attempting to recover from the memory allocation failure, allocation is retried. If there is no new-handler an exception of type "**bad_alloc**" is thrown.

RETURNS

Pointer to new object.

THROWS

std::bad_alloc if allocation failed.

SEE ALSO

cplusLib

O

operator new( )

NAME

operator new( ) – default run-time support for operator new (nothrow) (C++)

SYNOPSIS

```
extern void * operator new
    (
    size_t          n,        /* size of object to allocate */
    const nothrow_t &         /* supply argument of "nothrow" here */
    )
```

DESCRIPTION

This function provides the default implementation of operator new (nothrow). It allocates memory from the system memory partition for the requested object. The value, when evaluated, is a pointer of the type **pointer-to-***T* where *T* is the type of the new object.

If allocation fails, a new-handler, if one is defined, is called. If the new-handler returns, presumably after attempting to recover from the memory allocation failure, allocation is retried. If the **new_handler** throws a **bad_alloc** exception, the exception is caught and 0 is returned. If allocation fails and there is no **new_handler** 0 is returned.

RETURNS Pointer to new object or 0 if allocation fails.

INCLUDE FILES new

SEE ALSO **cplusLib**

operator new( )

NAME **operator new( )** – run-time support for operator new with placement (C++)

SYNOPSIS

```
extern void * operator new
    (
    size_t n,                  /* size of object to allocate (unused) */
    void * pMem                /* pointer to allocated memory */
    )
```

DESCRIPTION This function provides the default implementation of the global new operator, with support for the placement syntax. New-with-placement is used to initialize objects for which memory has already been allocated. *pMem* points to the previously allocated memory.

RETURNS pMem

INCLUDE FILES new

SEE ALSO **cplusLib**

passFsDevInit( )

NAME **passFsDevInit( )** – associate a device with passFs file system functions

SYNOPSIS

```
void *passFsDevInit
    (
    char * devName              /* device name */
    )
```

DESCRIPTION This routine associates the name *devName* with the file system and installs it in the I/O System's device table. The driver number used when the device is added to the table is that which was assigned to the passFs library during **passFsInit( )**.

RETURNS A pointer to the volume descriptor, or **NULL** if there is an error.

SEE ALSO **passFsLib**

passFsInit( )

NAME **passFsInit( )** – prepare to use the passFs library

SYNOPSIS

```
STATUS passFsInit
    (
    int nPassfs                 /* number of pass-through file systems */
    )
```

DESCRIPTION This routine initializes the passFs library. It must be called exactly once, before any other routines in the library. The argument specifies the number of passFs devices that may be open at once. This routine installs **passFsLib** as a driver in the I/O system driver table, allocates and sets up the necessary memory structures, and initializes semaphores.

Normally this routine is called from the root task, **usrRoot( )**, in **usrConfig( )**. This initialization is enabled when the configuration macro **INCLUDE_PASSFS** is defined.

NOTE: Maximum number of pass-through file systems is 1.

RETURNS **OK**, or **ERROR**.

SEE ALSO **passFsLib**

P

pause( )

NAME **pause( )** – suspend the task until delivery of a signal (POSIX)

SYNOPSIS

```
int pause (void)
```

DESCRIPTION This routine suspends the task until delivery of a signal.

NOTE: Since the **pause( )** function suspends thread execution indefinitely, there is no successful completion return value.

RETURNS -1, always.

ERRNO **EINTR**

SEE ALSO **sigLib**

pc( )

NAME **pc( )** – return the contents of the program counter

SYNOPSIS

```
int pc
    (
    int task                  /* task ID */
    )
```

DESCRIPTION This command extracts the contents of the program counter for a specified task from the task's TCB. If *task* is omitted or 0, the current task is used.

RETURNS The contents of the program counter.

SEE ALSO **usrLib**, **ti( )**, *VxWorks Programmer's Guide: Target Shell*

pentiumBtc( )

NAME **pentiumBtc( )** – execute atomic compare-and-exchange instruction to clear a bit

SYNOPSIS

```
STATUS pentiumBtc (pFlag)
    char * pFlag;                    /* flag address */
```

DESCRIPTION This routine compares a byte specified by the first parameter with **TRUE**. If it is **TRUE**, it changes it to 0 and returns **OK**. If it is not **TRUE**, it returns **ERROR**. LOCK and CMPXCHGB are used to get the atomic memory access.

RETURNS **OK** or **ERROR** if the specified flag is not **TRUE**

SEE ALSO **pentiumALib**

pentiumBts( )

NAME **pentiumBts( )** – execute atomic compare-and-exchange instruction to set a bit

SYNOPSIS

```
STATUS pentiumBts (pFlag)
    char * pFlag;                    /* flag address */
```

DESCRIPTION This routine compares a byte specified by the first parameter with 0. If it is 0, it changes it to **TRUE** and returns **OK**. If it is not 0, it returns **ERROR**. LOCK and CMPXCHGB are used to get the atomic memory access.

RETURNS **OK** or **ERROR** if the specified flag is not zero.

SEE ALSO **pentiumALib**

pentiumCr4Get( )

NAME
pentiumCr4Get( ) – get contents of CR4 register

SYNOPSIS
```
int pentiumCr4Get (void)
```

DESCRIPTION
This routine gets the contents of the CR4 register.

RETURNS
Contents of CR4 register.

SEE ALSO
pentiumALib

pentiumCr4Set( )

NAME
pentiumCr4Set( ) – sets specified value to the CR4 register

SYNOPSIS
```
void pentiumCr4Set (cr4)
   int cr4;           /* value to write CR4 register */
```

DESCRIPTION
This routine sets a specified value to the CR4 register.

RETURNS
N/A

SEE ALSO
pentiumALib

pentiumMcaEnable( )

NAME

pentiumMcaEnable( ) – enable/disable the MCA (Machine Check Architecture)

SYNOPSIS

```
void pentiumMcaEnable
    (
    BOOL enable               /* TRUE to enable, FALSE to disable the MCA */
    )
```

DESCRIPTION

This routine enables/disables 1) the Machine Check Architecture and its Error Reporting register banks 2) the Machine Check Exception by toggling the MCE bit in the CR4. This routine works on either P5, P6 or P7 family.

RETURNS

N/A

SEE ALSO

pentiumLib

pentiumMcaShow( )

NAME

pentiumMcaShow( ) – show MCA (Machine Check Architecture) registers

SYNOPSIS

```
void pentiumMcaShow (void)
```

DESCRIPTION

This routine shows Machine-Check global control registers and Error-Reporting register banks. Number of the Error-Reporting register banks is kept in a variable **mcaBanks**. **MCi_ADDR** and **MCi_MISC** registers in the Error-Reporting register bank are showed if **MCi_STATUS** indicates that these registers are valid.

RETURNS

N/A

SEE ALSO

pentiumShow

pentiumMsrGet( )

NAME

pentiumMsrGet( ) – get the contents of the specified MSR (Model Specific Register)

SYNOPSIS

```
void pentiumMsrGet (addr, pData)
    int addr;                 /* MSR address */
    long long int * pData;    /* MSR data */
```

DESCRIPTION

This routine gets the contents of the specified MSR. The first parameter is an address of the MSR. The second parameter is a pointer of 64Bit variable.

RETURNS

N/A

SEE ALSO

pentiumALib

pentiumMsrInit( )

NAME

pentiumMsrInit( ) – initialize all the MSRs (Model Specific Register)

SYNOPSIS

```
STATUS pentiumMsrInit (void)
```

DESCRIPTION

This routine initializes all the MSRs in the processor. This routine works on either P5, P6 or P7 family processors.

RETURNS

OK, or **ERROR** if RDMSR/WRMSR instructions are not supported.

SEE ALSO

pentiumLib

pentiumMsrSet( )

NAME **pentiumMsrSet( )** – set a value to the specified MSR (Model Specific Registers)

SYNOPSIS

```
void pentiumMsrSet (addr, pData)
    int addr;                       /* MSR address */
    long long int * pData;          /* MSR data */
```

DESCRIPTION This routine sets a value to a specified MSR. The first parameter is an address of the MSR. The second parameter is a pointer of 64Bit variable.

RETURNS N/A

SEE ALSO **pentiumALib**

pentiumMsrShow( )

NAME **pentiumMsrShow( )** – show all the MSR (Model Specific Register)

SYNOPSIS

```
void pentiumMsrShow (void)
```

DESCRIPTION This routine shows all the MSRs in the Pentium and Pentium[234].

RETURNS N/A

SEE ALSO **pentiumShow**

pentiumMtrrDisable( )

NAME
pentiumMtrrDisable( ) – disable MTRR (Memory Type Range Register)

SYNOPSIS
```
void pentiumMtrrDisable (void)
```

DESCRIPTION
This routine disables the MTRR that provide a mechanism for associating the memory types with physical address ranges in system memory.

RETURNS
N/A

SEE ALSO
pentiumLib

pentiumMtrrEnable( )

NAME
pentiumMtrrEnable( ) – enable MTRR (Memory Type Range Register)

SYNOPSIS
```
void pentiumMtrrEnable (void)
```

DESCRIPTION
This routine enables the MTRR that provide a mechanism for associating the memory types with physical address ranges in system memory.

RETURNS
N/A

SEE ALSO
pentiumLib

pentiumMtrrGet( )

NAME **pentiumMtrrGet( )** – get MTRRs to a specified MTRR table

SYNOPSIS

```
STATUS pentiumMtrrGet
    (
    MTRR * pMtrr                /* MTRR table */
    )
```

DESCRIPTION This routine gets MTRRs to a specified MTRR table with RDMSR instruction. The read MTRRs are CAP register, DEFTYPE register, fixed range MTRRs, and variable range MTRRs.

RETURNS **OK**, or **ERROR** if MTRR is being accessed.

SEE ALSO **pentiumLib**

pentiumMtrrSet( )

NAME **pentiumMtrrSet( )** – set MTRRs from specified MTRR table with WRMSR instruction.

SYNOPSIS

```
STATUS pentiumMtrrSet
    (
    MTRR * pMtrr                /* MTRR table */
    )
```

DESCRIPTION This routine sets MTRRs from specified MTRR table with WRMSR instruction. The written MTRRs are DEFTYPE register, fixed range MTRRs, and variable range MTRRs.

RETURNS **OK**, or **ERROR** if MTRR is enabled or being accessed.

SEE ALSO **pentiumLib**

pentiumP5PmcGet( )

NAME

pentiumP5PmcGet( ) – get the contents of P5 PMC0 and PMC1

SYNOPSIS

```
void pentiumP5PmcGet (pPmc0, pPmc1)
    long long int * pPmc0;          /* Performance Monitoring Counter 0 */
    long long int * pPmc1;          /* Performance Monitoring Counter 1 */
```

DESCRIPTION

This routine gets the contents of both PMC0 (Performance Monitoring Counter 0) and PMC1. The first parameter is a pointer of 64Bit variable to store the content of the Counter 0, and the second parameter is for the Counter 1.

RETURNS

N/A

SEE ALSO

pentiumALib

pentiumP5PmcGet0( )

NAME

pentiumP5PmcGet0( ) – get the contents of P5 PMC0

SYNOPSIS

```
void pentiumP5PmcGet0 (pPmc0)
    long long int * pPmc0;          /* Performance Monitoring Counter 0 */
```

DESCRIPTION

This routine gets the contents of PMC0 (Performance Monitoring Counter 0). The parameter is a pointer of 64Bit variable to store the content of the Counter.

RETURNS

N/A

SEE ALSO

pentiumALib

pentiumP5PmcGet1( )

NAME

pentiumP5PmcGet1( ) – get the contents of P5 PMC1

SYNOPSIS

```
void pentiumP5PmcGet1 (pPmc1)
    long long int * pPmc1;           /* Performance Monitoring Counter 1 */
```

DESCRIPTION

This routine gets a content of PMC1 (Performance Monitoring Counter 1). Parameter is a pointer of 64Bit variable to store the content of the Counter.

RETURNS

N/A

SEE ALSO

pentiumALib

pentiumP5PmcReset( )

NAME

pentiumP5PmcReset( ) – reset both PMC0 and PMC1

SYNOPSIS

```
void pentiumP5PmcReset (void)
```

DESCRIPTION

This routine resets both PMC0 (Performance Monitoring Counter 0) and PMC1.

RETURNS

N/A

SEE ALSO

pentiumALib

pentiumP5PmcReset0( )

NAME **pentiumP5PmcReset0( )** – reset PMC0

SYNOPSIS

```
void pentiumP5PmcReset0 (void)
```

DESCRIPTION This routine resets PMC0 (Performance Monitoring Counter 0).

RETURNS N/A

SEE ALSO **pentiumALib**

pentiumP5PmcReset1( )

NAME **pentiumP5PmcReset1( )** – reset PMC1

SYNOPSIS

```
void pentiumP5PmcReset1 (void)
```

DESCRIPTION This routine resets PMC1 (Performance Monitoring Counter 1).

RETURNS N/A

SEE ALSO **pentiumALib**

pentiumP5PmcStart0( )

NAME **pentiumP5PmcStart0( )** – start PMC0

SYNOPSIS

```
STATUS pentiumP5PmcStart0 (pmc0Cesr)
      int pmc0Cesr;           /* PMC0 control and event select */
```

DESCRIPTION This routine starts PMC0 (Performance Monitoring Counter 0) by writing specified PMC0 events to Performance Event Select Registers. The only parameter is the content of Performance Event Select Register.

RETURNS **OK** or **ERROR** if PMC0 is already started.

SEE ALSO **pentiumALib**

pentiumP5PmcStart1( )

NAME **pentiumP5PmcStart1( )** – start PMC1

SYNOPSIS

```
STATUS pentiumP5PmcStart1 (pmc1Cesr)
      int pmc1Cesr;           /* PMC1 control and event select */
```

DESCRIPTION This routine starts PMC1 (Performance Monitoring Counter 0) by writing specified PMC1 events to Performance Event Select Registers. The only parameter is the content of Performance Event Select Register.

RETURNS **OK** or **ERROR** if PMC1 is already started.

SEE ALSO **pentiumALib**

pentiumP5PmcStop( )

NAME
pentiumP5PmcStop( ) – stop both P5 PMC0 and PMC1

SYNOPSIS
```
void pentiumP5PmcStop (void)
```

DESCRIPTION
This routine stops both PMC0 (Performance Monitoring Counter 0) and PMC1 by clearing two Performance Event Select Registers.

RETURNS
N/A

SEE ALSO
pentiumALib

pentiumP5PmcStop0( )

NAME
pentiumP5PmcStop0( ) – stop P5 PMC0

SYNOPSIS
```
void pentiumP5PmcStop0 (void)
```

DESCRIPTION
This routine stops only PMC0 (Performance Monitoring Counter 0) by clearing the PMC0 bits of Control and Event Select Register.

RETURNS
N/A

SEE ALSO
pentiumALib

pentiumP5PmcStop1( )

NAME **pentiumP5PmcStop1( )** – stop P5 PMC1

SYNOPSIS

```
void pentiumP5PmcStop1 (void)
```

DESCRIPTION This routine stops only PMC1 (Performance Monitoring Counter 1) by clearing the PMC1 bits of Control and Event Select Register.

RETURNS N/A

SEE ALSO **pentiumALib**

pentiumP6PmcGet( )

NAME **pentiumP6PmcGet( )** – get the contents of PMC0 and PMC1

SYNOPSIS

```
void pentiumP6PmcGet (pPmc0, pPmc1)
    long long int * pPmc0;          /* Performance Monitoring Counter 0 */
    long long int * pPmc1;          /* Performance Monitoring Counter 1 */
```

DESCRIPTION This routine gets the contents of both PMC0 (Performance Monitoring Counter 0) and PMC1. The first parameter is a pointer of 64Bit variable to store the content of the Counter 0, and the second parameter is for the Counter 1.

RETURNS N/A

SEE ALSO **pentiumALib**

pentiumP6PmcGet0( )

NAME

pentiumP6PmcGet0( ) – get the contents of PMC0

SYNOPSIS

```
void pentiumP6PmcGet0 (pPmc0)
    long long int * pPmc0;           /* Performance Monitoring Counter 0 */
```

DESCRIPTION

This routine gets the contents of PMC0 (Performance Monitoring Counter 0). The parameter is a pointer of 64Bit variable to store the content of the Counter.

RETURNS

N/A

SEE ALSO

pentiumALib

pentiumP6PmcGet1( )

NAME

pentiumP6PmcGet1( ) – get the contents of PMC1

SYNOPSIS

```
void pentiumP6PmcGet1 (pPmc1)
    long long int * pPmc1;           /* Performance Monitoring Counter 1 */
```

DESCRIPTION

This routine gets a content of PMC1 (Performance Monitoring Counter 1). Parameter is a pointer of 64Bit variable to store the content of the Counter.

RETURNS

N/A

SEE ALSO

pentiumALib

pentiumP6PmcReset( )

NAME **pentiumP6PmcReset( )** – reset both PMC0 and PMC1

SYNOPSIS
```
void pentiumP6PmcReset (void)
```

DESCRIPTION This routine resets both PMC0 (Performance Monitoring Counter 0) and PMC1.

RETURNS N/A

SEE ALSO **pentiumALib**

pentiumP6PmcReset0( )

NAME **pentiumP6PmcReset0( )** – reset PMC0

SYNOPSIS
```
void pentiumP6PmcReset0 (void)
```

DESCRIPTION This routine resets PMC0 (Performance Monitoring Counter 0).

RETURNS N/A

SEE ALSO **pentiumALib**

pentiumP6PmcReset1( )

NAME **pentiumP6PmcReset1( )** – reset PMC1

SYNOPSIS
```
void pentiumP6PmcReset1 (void)
```

DESCRIPTION This routine resets PMC1 (Performance Monitoring Counter 1).

RETURNS N/A

SEE ALSO **pentiumALib**

pentiumP6PmcStart( )

NAME

pentiumP6PmcStart( ) – start both PMC0 and PMC1

SYNOPSIS

```
STATUS pentiumP6PmcStart (pmcEvtSel0, pmcEvtSel1)
    int pmcEvtSel0;           /* Performance Event Select Register 0 */
    int pmcEvtSel1;           /* Performance Event Select Register 1 */
```

DESCRIPTION

This routine starts both PMC0 (Performance Monitoring Counter 0) and PMC1 by writing specified events to Performance Event Select Registers. The first parameter is a content of Performance Event Select Register 0, and the second parameter is for the Performance Event Select Register 1.

RETURNS

OK or **ERROR** if PMC is already started.

SEE ALSO

pentiumALib

pentiumP6PmcStop( )

NAME

pentiumP6PmcStop( ) – stop both PMC0 and PMC1

SYNOPSIS

```
void pentiumP6PmcStop (void)
```

DESCRIPTION

This routine stops both PMC0 (Performance Monitoring Counter 0) and PMC1 by clearing two Performance Event Select Registers.

RETURNS

N/A

SEE ALSO

pentiumALib

pentiumP6PmcStop1( )

NAME **pentiumP6PmcStop1( )** – stop PMC1

SYNOPSIS

```
void pentiumP6PmcStop1 (void)
```

DESCRIPTION This routine stops only PMC1 (Performance Monitoring Counter 1) by clearing the Performance Event Select Register 1. Note, clearing the Performance Event Select Register 0 stops both counters, PMC0 and PMC1.

RETURNS N/A

SEE ALSO **pentiumALib**

pentiumPmcGet( )

NAME **pentiumPmcGet( )** – get the contents of PMC0 and PMC1

SYNOPSIS

```
void pentiumPmcGet (pPmc0, pPmc1)
    long long int * pPmc0;          /* Performance Monitoring Counter 0 */
    long long int * pPmc1;          /* Performance Monitoring Counter 1 */
```

DESCRIPTION This routine gets the contents of both PMC0 (Performance Monitoring Counter 0) and PMC1. The first parameter is a pointer of 64Bit variable to store the content of the Counter 0, and the second parameter is for the Counter 1.

RETURNS N/A

SEE ALSO **pentiumLib**

pentiumPmcGet0( )

NAME

pentiumPmcGet0( ) – get the contents of PMC0

SYNOPSIS

```
void pentiumPmcGet0 (pPmc0)
    long long int * pPmc0;            /* Performance Monitoring Counter 0 */
```

DESCRIPTION

This routine gets the contents of PMC0 (Performance Monitoring Counter 0). The parameter is a pointer of 64Bit variable to store the content of the Counter.

RETURNS

N/A

SEE ALSO

pentiumLib

pentiumPmcGet1( )

NAME

pentiumPmcGet1( ) – get the contents of PMC1

SYNOPSIS

```
void pentiumPmcGet1 (pPmc1)
    long long int * pPmc1;            /* Performance Monitoring Counter 1 */
```

DESCRIPTION

This routine gets a content of PMC1 (Performance Monitoring Counter 1). Parameter is a pointer of 64Bit variable to store the content of the Counter.

RETURNS

N/A

SEE ALSO

pentiumLib

pentiumPmcReset( )

NAME **pentiumPmcReset( )** – reset both PMC0 and PMC1

SYNOPSIS

```
void pentiumPmcReset (void)
```

DESCRIPTION This routine resets both PMC0 (Performance Monitoring Counter 0) and PMC1.

RETURNS N/A

SEE ALSO **pentiumLib**

pentiumPmcReset0( )

NAME **pentiumPmcReset0( )** – reset PMC0

SYNOPSIS

```
void pentiumPmcReset0 (void)
```

DESCRIPTION This routine resets PMC0 (Performance Monitoring Counter 0).

RETURNS N/A

SEE ALSO **pentiumLib**

pentiumPmcReset1( )

NAME **pentiumPmcReset1( )** – reset PMC1

SYNOPSIS

```
void pentiumPmcReset1 (void)
```

DESCRIPTION This routine resets PMC1 (Performance Monitoring Counter 1).

RETURNS N/A

SEE ALSO **pentiumLib**

pentiumPmcShow( )

NAME

pentiumPmcShow( ) – show PMCs (Performance Monitoring Counters)

SYNOPSIS

```
void pentiumPmcShow
    (
    BOOL zap                   /* 1: reset PMC0 and PMC1 */
    )
```

DESCRIPTION

This routine shows Performance Monitoring Counter 0 and 1. Monitored events are selected by Performance Event Select Registers in **pentiumPmcStart ()**. These counters are cleared to 0 if the parameter "zap" is **TRUE**.

RETURNS

N/A

SEE ALSO

pentiumShow

pentiumPmcStart( )

NAME

pentiumPmcStart( ) – start both PMC0 and PMC1

SYNOPSIS

```
STATUS pentiumPmcStart (pmcEvtSel0, pmcEvtSel1)
    int pmcEvtSel0;            /* Performance Event Select Register 0 */
    int pmcEvtSel1;            /* Performance Event Select Register 1 */
```

DESCRIPTION

This routine starts both PMC0 (Performance Monitoring Counter 0) and PMC1 by writing specified events to Performance Event Select Registers. The first parameter is a content of Performance Event Select Register 0, and the second parameter is for the Performance Event Select Register 1.

RETURNS

OK or **ERROR** if PMC is already started.

SEE ALSO

pentiumLib

pentiumPmcStart0( )

NAME **pentiumPmcStart0( )** – start PMC0

SYNOPSIS

```
STATUS pentiumPmcStart0 (pmcEvtSel0)
      int pmcEvtSel0;            /* PMC0 control and event select */
```

This routine starts PMC0 (Performance Monitoring Counter 0) by writing specified PMC0 events to Performance Event Select Registers. The only parameter is the content of Performance Event Select Register.

RETURNS **OK** or **ERROR** if PMC is already started.

SEE ALSO **pentiumLib**

pentiumPmcStart1( )

NAME **pentiumPmcStart1( )** – start PMC1

SYNOPSIS

```
STATUS pentiumPmcStart1 (pmcEvtSel1)
      int pmcEvtSel1;            /* PMC1 control and event select */
```

This routine starts PMC1 (Performance Monitoring Counter 0) by writing specified PMC1 events to Performance Event Select Registers. The only parameter is the content of Performance Event Select Register.

RETURNS **OK** or **ERROR** if PMC1 is already started.

SEE ALSO **pentiumLib**

pentiumPmcStop( )

NAME
pentiumPmcStop( ) – stop both PMC0 and PMC1

SYNOPSIS
```
void pentiumPmcStop (void)
```

This routine stops both PMC0 (Performance Monitoring Counter 0) and PMC1 by clearing two Performance Event Select Registers.

RETURNS
N/A

SEE ALSO
pentiumLib

pentiumPmcStop0( )

NAME
pentiumPmcStop0( ) – stop PMC0

SYNOPSIS
```
void pentiumPmcStop0 (void)
```

This routine stops only PMC0 (Performance Monitoring Counter 0) by clearing the PMC0 bits of Control and Event Select Register.

RETURNS
N/A

SEE ALSO
pentiumLib

pentiumPmcStop1( )

NAME
pentiumPmcStop1( ) – stop PMC1

SYNOPSIS
```
void pentiumPmcStop1 (void)
```

This routine stops only PMC1 (Performance Monitoring Counter 1) by clearing the PMC1 bits of Control and Event Select Register.

RETURNS
N/A

SEE ALSO
pentiumLib

pentiumSerialize( )

NAME **pentiumSerialize( )** – execute a serializing instruction CPUID

SYNOPSIS

```
void pentiumSerialize (void)
```

DESCRIPTION This routine executes a serializing instruction CPUID. Serialization means that all modifications to flags, registers, and memory by previous instructions are completed before the next instruction is fetched and executed and all buffered writes have drained to memory.

RETURNS N/A

SEE ALSO **pentiumALib**

pentiumTlbFlush( )

NAME **pentiumTlbFlush( )** – flush TLBs (Translation Lookaside Buffers)

SYNOPSIS

```
void pentiumTlbFlush (void)
```

DESCRIPTION This routine flushes TLBs by loading the CR3 register. All of the TLBs are automatically invalidated any time the CR3 register is loaded. The page global enable (PGE) flag in register CR4 and the global flag in a page-directory or page-table entry can be used to frequently used pages from being automatically invalidated in the TLBs on a load of CR3 register. The only way to deterministically invalidate global page entries is to clear the PGE flag and then invalidate the TLBs.

RETURNS N/A

SEE ALSO **pentiumALib**

pentiumTscGet32( )

NAME
pentiumTscGet32( ) – get the lower half of the 64Bit TSC (Timestamp Counter)

SYNOPSIS
```
UINT32 pentiumTscGet32 (void)
```

DESCRIPTION
This routine gets a lower half of the 64Bit TSC by RDTSC instruction. RDTSC instruction saves the lower 32Bit in EAX register, so this routine simply returns after executing RDTSC instruction.

RETURNS
Lower half of the 64Bit TSC (Timestamp Counter)

SEE ALSO
pentiumALib

pentiumTscGet64( )

NAME
pentiumTscGet64( ) – get 64Bit TSC (Timestamp Counter)

SYNOPSIS
```
void pentiumTscGet64 (pTsc)
    long long int * pTsc;            /* Timestamp Counter */
```

DESCRIPTION
This routine gets 64Bit TSC by RDTSC instruction. Parameter is a pointer of 64Bit variable to store the content of the Counter.

RETURNS
N/A

SEE ALSO
pentiumALib

pentiumTscReset( )

NAME
pentiumTscReset( ) – reset the TSC (Timestamp Counter)

SYNOPSIS
```
void pentiumTscReset (void)
```

DESCRIPTION
This routine resets the TSC by writing zero to the TSC with WRMSR instruction.

RETURNS N/A

SEE ALSO **pentiumALib**

period( )

NAME **period( )** – spawn a task to call a function periodically

SYNOPSIS

```
int period
    (
    int      secs,             /* period in seconds */
    FUNCPTR func,              /* function to call repeatedly */
    int      arg1,             /* first of eight args to pass to func */
    int      arg2,
    int      arg3,
    int      arg4,
    int      arg5,
    int      arg6,
    int      arg7,
    int      arg8
    )
```

DESCRIPTION This command spawns a task that repeatedly calls a specified function, with up to eight of its arguments, delaying the specified number of seconds between calls.

For example, to have **i( )** display task information every 5 seconds, just type:

```
-> period 5, i
```

NOTE: The task is spawned using the **sp( )** routine. See the description of **sp( )** for details about priority, options, stack size, and task ID.

RETURNS A task ID, or **ERROR** if the task cannot be spawned.

SEE ALSO **usrLib**, **periodRun( )**, **sp( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

P

periodRun( )

NAME

periodRun( ) – call a function periodically

SYNOPSIS

```
void periodRun
    (
    int      secs,              /* no. of seconds to delay between calls */
    FUNCPTR  func,              /* function to call repeatedly */
    int      arg1,              /* first of eight args to pass to func */
    int      arg2,
    int      arg3,
    int      arg4,
    int      arg5,
    int      arg6,
    int      arg7,
    int      arg8
    )
```

DESCRIPTION

This command repeatedly calls a specified function, with up to eight of its arguments, delaying the specified number of seconds between calls.

Normally, this routine is called only by **period( )**, which spawns it as a task.

RETURNS

N/A

SEE ALSO

usrLib, **period( )**, *VxWorks Programmer's Guide: Target Shell*

perror( )

NAME

perror( ) – map an error number in **errno** to an error message (ANSI)

SYNOPSIS

```
void perror
    (
    const char * __s            /* error string */
    )
```

DESCRIPTION

This routine maps the error number in the integer expression **errno** to an error message. It writes a sequence of characters to the standard error stream as follows: first (if *__s* is not a null pointer and the character pointed to by *__s* is not the null character), the string pointed to by *__s* followed by a colon (:) and a space; then an appropriate error message string followed by a new-line character. The contents of the error message strings are the same as those returned by **strerror( )** with the argument **errno**.

INCLUDE FILES **stdio.h**

RETURNS N/A

SEE ALSO **ansiStdio**, **strerror( )**

ping( )

NAME **ping( )** – test that a remote host is reachable

SYNOPSIS

```
STATUS ping
    (
    char *   host,              /* host to ping */
    int      numPackets,        /* number of packets to receive */
    ulong_t  options            /* option flags */
    )
```

DESCRIPTION This routine tests that a remote host is reachable by sending ICMP echo request packets, and waiting for replies. It may called from the VxWorks shell as follows:

```
-> ping "remoteSystem", 1, 0
```

where *remoteSystem* is either a host name that has been previously added to the remote host table by a call to **hostAdd( )**, or an Internet address in dot notation (for example, "90.0.0.2").

The second parameter, *numPackets*, specifies the number of ICMP packets to receive from the remote host. If *numPackets* is 1, this routine waits for a single echo reply packet, and then prints a short message indicating whether the remote host is reachable. For all other values of *numPackets*, timing and sequence information is printed as echoed packets are received. If *numPackets* is 0, this routine runs continuously.

If no replies are received within a 5-second timeout period, the routine exits. An **ERROR** status is returned if no echo replies are received from the remote host.

The following flags may be given through the *options* parameter:

PING_OPT_SILENT
Suppress output. This option is useful for applications that use **ping( )** programmatically to examine the return status.

PING_OPT_DONTROUTE
Do not route packets past the local network. This also prevents pinging local addresses (*i.e.*, the IP address of the host itself). The 127.x.x.x addresses will still work however.

PING_OPT_NOHOST
Suppress host lookup. This is useful when you have the DNS resolver but the DNS server is down and not returning host names.

PING_OPT_DEBUG
Enables debug output.

NOTE The following global variables can be set from the target shell or **Windsh** to configure the **ping( )** parameters:

_pingTxLen
Size of the ICMP echo packet (default 64).

_pingTxInterval
Packet interval in seconds (default 1 second).

_pingTxTmo
Packet timeout in seconds (default 5 seconds).

RETURNS **OK**, or **ERROR** if the remote host is not reachable.

ERRNO **EINVAL**, **S_pingLib_NOT_INITIALIZED**, **S_pingLib_TIMEOUT**

SEE ALSO **pingLib**

pingLibInit( )

NAME **pingLibInit( )** – initialize the **ping( )** utility

SYNOPSIS
```
STATUS pingLibInit (void)
```

DESCRIPTION This routine allocates resources used by the **ping( )** utility. It is called automatically when **INCLUDE_PING** is defined.

RETURNS **OK**

SEE ALSO **pingLib**

pipeDevCreate( )

NAME

pipeDevCreate( ) – create a pipe device

SYNOPSIS

```
STATUS pipeDevCreate
    (
    char * name,              /* name of pipe to be created */
    int    nMessages,         /* max. number of messages in pipe */
    int    nBytes             /* size of each message */
    )
```

DESCRIPTION

This routine creates a pipe device. It cannot be called from an interrupt service routine. It allocates memory for the necessary structures and initializes the device. The pipe device will have a maximum of *nMessages* messages of up to *nBytes* each in the pipe at once. When the pipe is full, a task attempting to write to the pipe will be suspended until a message has been read. Messages are lost if written to a full pipe at interrupt level.

RETURNS

OK, or **ERROR** if the call fails.

ERRNO

S_ioLib_NO_DRIVER - driver not initialized
S_intLib_NOT_ISR_CALLABLE - cannot be called from an ISR

SEE ALSO

pipeDrv

P

pipeDevDelete( )

NAME

pipeDevDelete( ) – delete a pipe device

SYNOPSIS

```
STATUS pipeDevDelete
    (
    char * name,              /* name of pipe to be deleted */
    BOOL   force              /* if TRUE, force pipe deletion */
    )
```

DESCRIPTION

This routine deletes a pipe device of a given name. The name must match that passed to **pipeDevCreate( )** else **ERROR** will be returned. This routine frees memory for the necessary structures and deletes the device. It cannot be called from an interrupt service routine.

A pipe device cannot be deleted until its number of open requests has been reduced to zero by an equal number of close requests and there are no tasks pending in its select list. If the optional force flag is asserted, the above restrictions are ignored, resulting in forced deletion of any select list and freeing of pipe resources.

WARNING: 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 - driver not initialized
S_intLib_NOT_ISR_CALLABLE - cannot be called from an ISR
EMFILE - pipe still has other openings
EBUSY - pipe is selected by at least one pending task

SEE ALSO **pipeDrv**

pipeDrv( )

NAME **pipeDrv( )** – initialize the pipe driver

SYNOPSIS

```
STATUS pipeDrv (void)
```

DESCRIPTION This routine initializes and installs the driver. It must be called before any pipes are created. It is called automatically by the root task, **usrRoot( )**, in **usrConfig.c** when the configuration macro **INCLUDE_PIPES** is defined.

RETURNS **OK**, or **ERROR** if the driver installation fails.

SEE ALSO **pipeDrv**

pow( )

NAME **pow( )** – compute the value of a number raised to a specified power (ANSI)

SYNOPSIS

```
double pow
    (
    double x,                  /* operand */
    double y                   /* exponent */
    )
```

DESCRIPTION This routine returns *x* to the power of *y* in double precision (IEEE double, 53 bits).

A domain error occurs if *x* is negative and *y* is not an integral value. A domain error occurs if the result cannot be represented when *x* is zero and *y* is less than or equal to zero. A range error may occur.

INCLUDE FILES **math.h**

RETURNS The double-precision value of *x* to the power of *y*. Special cases:

(anything) ** 0	is	1
(anything) ** 1	is	itself
(anything) ** NaN	is	NaN
NaN ** (anything except 0)	is	NaN
+-(anything> 1) ** +INF	is	+INF
+-(anything> 1) ** -INF	is	+0
+-(anything \< 1) ** +INF	is	+0
+-(anything \< 1) ** -INF	is	+INF
+-1 ** +-INF	is	NaN, signal INVALID
+0 ** +(anything non-0, NaN)	is	+0
-0 ** +(anything non-0, NaN, odd int)	is	+0
+0 ** -(anything non-0, NaN)	is	+INF, signal DIV-BY-ZERO
-0 ** -(anything non-0, NaN, odd int)	is	+INF with signal
-0 ** (odd integer)	=	-(+0 ** (odd integer))
+INF ** +(anything except 0, NaN)	is	+INF
+INF ** -(anything except 0, NaN)	is	+0
-INF ** (odd integer)	=	-(+INF ** (odd integer))
-INF ** (even integer)	=	(+INF ** (even integer))
-INF ** -(any non-integer, NaN)	is	NaN with signal
-(x=anything) ** (k=integer)	is	(-1)**k * (x ** k)
-(anything except 0) ** (non-integer)	is	NaN with signal

SEE ALSO **ansiMath**, **mathALib**

powf( )

NAME

powf( ) – compute the value of a number raised to a specified power (ANSI)

SYNOPSIS

```
float powf
    (
    float x,    /* operand  */
    float y     /* exponent */
    )
```

DESCRIPTION

This routine returns the value of *x* to the power of *y* in single precision.

INCLUDE FILES

math.h

RETURNS

The single-precision value of *x* to the power of *y*.

SEE ALSO

mathALib

pppDelete( )

NAME

pppDelete( ) – delete a PPP network interface

SYNOPSIS

```
void pppDelete
    (
    int unit                     /* PPP interface unit number to delete */
    )
```

DESCRIPTION

This routine deletes the Point-to-Point Protocol (PPP) network interface specified by the unit number *unit*.

A Link Control Protocol (LCP) terminate request packet is sent to notify the peer of the impending PPP link shut-down. The associated serial interface (*tty*) is then detached from the PPP driver, and the PPP interface is deleted from the list of network interfaces. Finally, all resources associated with the PPP link are returned to the VxWorks system.

RETURNS

N/A

SEE ALSO

pppLib

pppHookAdd( )

NAME **pppHookAdd( )** – add a hook routine on a unit basis

SYNOPSIS

```
STATUS pppHookAdd
    (
    int      unit,              /* unit number */
    FUNCPTR  hookRtn,           /* hook routine */
    int      hookType           /* hook type connect/disconnect */
    )
```

DESCRIPTION This routine adds a hook to the Point-to-Point Protocol (PPP) channel. The parameters to this routine specify the unit number (*unit*) of the PPP interface, the hook routine (*hookRtn*), and the type of hook specifying either a connect hook or a disconnect hook (*hookType*). The following hook types can be specified for the *hookType* parameter:

PPP_HOOK_CONNECT
Specify a connect hook.

PPP_HOOK_DISCONNECT
Specify a disconnect hook.

RETURNS **OK**, or **ERROR** if the hook cannot be added to the unit.

SEE ALSO **pppHookLib**, **pppHookDelete( )**

P

pppHookDelete( )

NAME **pppHookDelete( )** – delete a hook routine on a unit basis

SYNOPSIS

```
STATUS pppHookDelete
    (
    int unit,                   /* unit number */
    int hookType                /* hook type connect/disconnect */
    )
```

DESCRIPTION This routine deletes a hook added previously to the Point-to-Point Protocol (PPP) channel. The parameters to this routine specify the unit number (*unit*) of the PPP interface and the type of hook specifying either a connect hook or a disconnect hook (*hookType*). The following hook types can be specified for the *hookType* parameter:

PPP_HOOK_CONNECT
Specify a connect hook.

PPP_HOOK_DISCONNECT
Specify a disconnect hook.

RETURNS **OK**, or **ERROR** if the hook cannot be deleted for the unit.

SEE ALSO **pppHookLib**, **pppHookAdd( )**

pppInfoGet( )

NAME **pppInfoGet( )** – get PPP link status information

SYNOPSIS

```
STATUS pppInfoGet
    (
    int        unit,          /* PPP interface unit number to examine */
    PPP_INFO * pInfo          /* PPP_INFO structure to be filled */
    )
```

DESCRIPTION This routine gets status information pertaining to the specified Point-to-Point Protocol (PPP) link, regardless of the link state. State and option information is gathered for the Link Control Protocol (LCP), Internet Protocol Control Protocol (IPCP), Password Authentication Protocol (PAP), and Challenge-Handshake Authentication Protocol (CHAP).

The PPP link information is returned through a **PPP_INFO** structure, which is defined in **h/netinet/ppp/pppShow.h**.

RETURNS **OK**, or **ERROR** if *unit* is an invalid PPP unit number.

SEE ALSO **pppShow**, **pppLib**

pppInfoShow( )

NAME **pppInfoShow( )** – display PPP link status information

SYNOPSIS

```
void pppInfoShow (void)
```

DESCRIPTION This routine displays status information pertaining to each initialized Point-to-Point Protocol (PPP) link, regardless of the link state. State and option information is gathered for the Link Control Protocol (LCP), Internet Protocol Control Protocol (IPCP), Password Authentication Protocol (PAP), and Challenge-Handshake Authentication Protocol (CHAP).

RETURNS N/A

SEE ALSO **pppShow**, **pppLib**

pppInit( )

NAME **pppInit( )** – initialize a PPP network interface

SYNOPSIS

```
int pppInit
    (
    int           unit,        /* PPP interface unit number to initialize */
    char *        devname,     /* name of the tty device to be used */
    char *        local_addr,  /* local IP address of the PPP interface */
    char *        remote_addr, /* remote peer IP address of the PPP link */
    int           baud,        /* baud rate of tty; NULL = default */
    PPP_OPTIONS * pOptions,    /* PPP options structure pointer */
    char *        fOptions     /* PPP options file name */
    )
```

DESCRIPTION This routine initializes a Point-to-Point Protocol (PPP) network interface. The parameters to this routine specify the unit number (*unit*) of the PPP interface, the name of the serial interface (*tty*) device (*devname*), the IP addresses of the local and remote ends of the link, the interface baud rate, an optional configuration options structure pointer, and an optional configuration options file name.

IP ADDRESSES The *local_addr* and *remote_addr* parameters specify the IP addresses of the local and remote ends of the PPP link, respectively. If *local_addr* is **NULL**, the local IP address will be negotiated with the remote peer. If the remote peer does not assign a local IP address, it will default to the address associated with the local target's machine name. If *remote_addr* is **NULL**, the remote peer's IP address will obtained from the remote peer. A routing table entry to the remote peer will be automatically added once the PPP link is established.

CONFIGURATION OPTIONS STRUCTURE

The optional parameter *pOptions* specifies configuration options for the PPP link. If **NULL**, this parameter is ignored, otherwise it is assumed to be a pointer to a **PPP_OPTIONS** options structure (defined in **h/netinet/ppp/options.h**).

The "flags" member of the **PPP_OPTIONS** structure is a bit-mask, where the following bit-flags may be specified:

OPT_NO_ALL
: Do not request/allow any options.

OPT_PASSIVE_MODE
: Set passive mode.

OPT_SILENT_MODE
: Set silent mode.

OPT_DEFAULTROUTE
: Add default route.

OPT_PROXYARP
: Add proxy ARP entry.

OPT_IPCP_ACCEPT_LOCAL
: Accept peer's idea of the local IP address.

OPT_IPCP_ACCEPT_REMOTE
: Accept peer's idea of the remote IP address.

OPT_NO_IP
: Disable IP address negotiation.

OPT_NO_ACC
: Disable address/control compression.

OPT_NO_PC
: Disable protocol field compression.

OPT_NO_VJ
: Disable VJ (Van Jacobson) compression.

OPT_NO_VJCCOMP
: Disable VJ (Van Jacobson) connnection ID compression.

OPT_NO_ASYNCMAP
: Disable async map negotiation.

OPT_NO_MN
: Disable magic number negotiation.

OPT_NO_MRU
Disable MRU (Maximum Receive Unit) negotiation.

OPT_NO_PAP
Do not allow PAP authentication with peer.

OPT_NO_CHAP
Do not allow CHAP authentication with peer.

OPT_REQUIRE_PAP
Require PAP authentication with peer.

OPT_REQUIRE_CHAP
Require CHAP authentication with peer.

OPT_LOGIN
Use the login password database for PAP authentication of peer.

OPT_DEBUG
Enable PPP daemon debug mode.

OPT_DRIVER_DEBUG
Enable PPP driver debug mode.

The remaining members of the **PPP_OPTIONS** structure specify PPP configurations options that require string values. These options are:

char *asyncmap
Set the desired async map to the specified string.

char ***escape_chars**
Set the chars to escape on transmission to the specified string.

char ***vj_max_slots**
Set maximum number of VJ compression header slots to the specified string.

char *netmask
Set netmask value for negotiation to the specified string.

char *mru
Set MRU value for negotiation to the specified string.

char *mtu
Set MTU (Maximum Transmission Unit) value for negotiation to the specified string.

char ***lcp_echo_failure**
Set the maximum number of consecutive LCP echo failures to the specified string.

char ***lcp_echo_interval**
Set the interval in seconds between LCP echo requests to the specified string.

char ***lcp_restart**
Set the timeout in seconds for the LCP negotiation to the specified string.

P

char ***lcp_max_terminate**
Set the maximum number of transmissions for LCP termination requests to the specified string.

char ***lcp_max_configure**
Set the maximum number of transmissions for LCP configuration requests to the specified string.

char ***lcp_max_failure**
Set the maximum number of LCP configuration NAKs to the specified string.

char ***ipcp_restart**
Set the timeout in seconds for IPCP negotiation to the specified string.

char ***ipcp_max_terminate**
Set the maximum number of transmissions for IPCP termination requests to the specified string.

char ***ipcp_max_configure**
Set the maximum number of transmissions for IPCP configuration requests to the specified string.

char ***ipcp_max_failure**
Set the maximum number of IPCP configuration NAKs to the specified string.

char ***local_auth_name**
Set the local name for authentication to the specified string.

char ***remote_auth_name**
Set the remote name for authentication to the specified string.

char ***pap_file**
Get PAP secrets from the specified file. This option is necessary if either peer requires PAP authentication.

char ***pap_user_name**
Set the user name for PAP authentication with the peer to the specified string.

char ***pap_passwd**
Set the password for PAP authentication with the peer to the specified string.

char ***pap_restart**
Set the timeout in seconds for PAP negotiation to the specified string.

char ***pap_max_authreq**
Set the maximum number of transmissions for PAP authentication requests to the specified string.

char ***chap_file**
Get CHAP secrets from the specified file. This option is necessary if either peer requires CHAP authentication.

char ***chap_restart**
Set the timeout in seconds for CHAP negotiation to the specified string.

char ***chap_interval**
Set the interval in seconds for CHAP re-challenge to the specified string.

char ***chap_max_challenge**
Set the maximum number of transmissions for CHAP challenge to the specified string.

CONFIGURATION OPTIONS FILE

The optional parameter *fOptions* specifies configuration options for the PPP link. If **NULL**, this parameter is ignored, otherwise it is assumed to be the name of a configuration options file. The format of the options file is one option per line; comment lines start with "#". The following options are recognized:

no_all
Do not request/allow any options.

passive_mode
Set passive mode.

silent_mode
Set silent mode.

defaultroute
Add default route.

proxyarp
Add proxy ARP entry.

ipcp_accept_local
Accept peer's idea of the local IP address.

ipcp_accept_remote
Accept peer's idea of the remote IP address.

no_ip
Disable IP address negotiation.

no_acc
Disable address/control compression.

no_pc
Disable protocol field compression.

no_vj
Disable VJ (Van Jacobson) compression.

no_vjccomp
Disable VJ (Van Jacobson) connection ID compression.

no_asyncmap
: Disable async map negotiation.

no_mn
: Disable magic number negotiation.

no_mru
: Disable MRU (Maximum Receive Unit) negotiation.

no_pap
: Do not allow PAP authentication with peer.

no_chap
: Do not allow CHAP authentication with peer.

require_pap
: Require PAP authentication with peer.

require_chap
: Require CHAP authentication with peer.

login
: Use the login password database for PAP authentication of peer.

debug
: Enable PPP daemon debug mode.

driver_debug
: Enable PPP driver debug mode.

asyncmap *value*
: Set the desired async map to the specified value.

escape_chars *value*
: Set the chars to escape on transmission to the specified value.

vj_max_slots *value*
: Set maximum number of VJ compression header slots to the specified value.

netmask *value*
: Set netmask value for negotiation to the specified value.

mru *value*
: Set MRU value for negotiation to the specified value.

mtu *value*
: Set MTU value for negotiation to the specified value.

lcp_echo_failure *value*
: Set the maximum consecutive LCP echo failures to the specified value.

lcp_echo_interval *value*
: Set the interval in seconds between LCP echo requests to the specified value.

lcp_restart *value*
Set the timeout in seconds for the LCP negotiation to the specified value.

lcp_max_terminate *value*
Set the maximum number of transmissions for LCP termination requests.

lcp_max_configure *value*
Set the maximum number of transmissions for LCP configuration requests to the specified value.

lcp_max_failure *value*
Set the maximum number of LCP configuration NAKs to the specified value.

ipcp_restart *value*
Set the timeout in seconds for IPCP negotiation to the specified value.

ipcp_max_terminate *value*
Set the maximum number of transmissions for IPCP termination requests to the specified value.

ipcp_max_configure *value*
Set the maximum number of transmissions for IPCP configuration requests to the specified value.

ipcp_max_failure *value*
Set the maximum number of IPCP configuration NAKs to the specified value.

local_auth_name *name*
Set the local name for authentication to the specified name.

remote_auth_name *name*
Set the remote name for authentication to the specified name.

pap_file *file*
Get PAP secrets from the specified file. This option is necessary if either peer requires PAP authentication.

pap_user_name *name*
Set the user name for PAP authentication with the peer to the specified name.

-
Set the password for PAP authentication with the peer to the specified password.

pap_restart *value*
Set the timeout in seconds for PAP negotiation to the specified value.

pap_max_authreq *value*
Set the maximum number of transmissions for PAP authentication requests to the specified value.

chap_file *file*
Get CHAP secrets from the specified file. This option is necessary if either peer requires CHAP authentication.

chap_restart *value*
Set the timeout in seconds for CHAP negotiation to the specified value.

chap_interval *value*
Set the interval in seconds for CHAP re-challenge to the specified value.

chap_max_challenge *value*
Set the maximum number of transmissions for CHAP challenge to the specified value.

AUTHENTICATION

The VxWorks PPP implementation supports two separate user authentication protocols: the Password Authentication Protocol (PAP) and the Challenge-Handshake Authentication Protocol (CHAP). If authentication is required by either peer, it must be satisfactorily completed before the PPP link becomes fully operational. If authentication fails, the link will be automatically terminated.

EXAMPLES

The following routine initializes a PPP interface that uses the target's second serial port (**/tyCo/1**). The local IP address is 90.0.0.1; the IP address of the remote peer is 90.0.0.10. The baud rate is the default rate for the *tty* device. VJ compression and authentication have been disabled, and LCP echo requests have been enabled.

```
PPP_OPTIONS pppOpt;   /* PPP configuration options */
void routine ()
    {
    pppOpt.flags = OPT_PASSIVE_MODE | OPT_NO_PAP | OPT_NO_CHAP | OPT_NO_VJ;
    pppOpt.lcp_echo_interval = "30";
    pppOpt.lcp_echo_failure = "10";
    pppInit (0, "/tyCo/1", "90.0.0.1", "90.0.0.10", 0, &pppOpt, NULL);
    }
```

The following routine generates the same results as the previous example. The difference is that the configuration options are obtained from a file rather than a structure.

```
pppFile = "phobos:/tmp/ppp_options";  /* PPP configuration options file */
void routine ()
    {
    pppInit (0, "/tyCo/1", "90.0.0.1", "90.0.0.10", 0, NULL, pppFile);
    }
```

where **phobos:/tmp/ppp_options** contains:

```
    passive
    no_pap
    no_chap
    no_vj
    lcp_echo_interval 30
    lcp_echo_failure 10
```

RETURNS **OK**, or **ERROR** if the PPP interface cannot be initialized because the daemon task cannot be spawned or memory is insufficient.

SEE ALSO **pppLib**, **pppShow**, **pppDelete( )**, *VxWorks Programmer's Guide: Network*

pppSecretAdd( )

NAME **pppSecretAdd( )** – add a secret to the PPP authentication secrets table

SYNOPSIS

```
STATUS pppSecretAdd
    (
    char * client,              /* client being authenticated */
    char * server,              /* server performing authentication */
    char * secret,              /* secret used for authentication */
    char * addrs                /* acceptable client IP addresses */
    )
```

DESCRIPTION This routine adds a secret to the Point-to-Point Protocol (PPP) authentication secrets table. This table may be used by the Password Authentication Protocol (PAP) and Challenge-Handshake Authentication Protocol (CHAP) user authentication protocols.

When a PPP link is established, a "server" may require a "client" to authenticate itself using a "secret". Clients and servers obtain authentication secrets by searching secrets files, or by searching the secrets table constructed by this routine. Clients and servers search the secrets table by matching client and server names with table entries, and retrieving the associated secret.

Client and server names in the table consisting of "*" are considered wildcards; they serve as matches for any client and/or server name if an exact match cannot be found.

If *secret* starts with "@", *secret* is assumed to be the name of a file, wherein the actual secret can be read.

If *addrs* is not **NULL**, it should contain a list of acceptable client IP addresses. When a server is authenticating a client and the client's IP address is not contained in the list of acceptable addresses, the link is terminated. Any IP address will be considered acceptable if *addrs* is **NULL**. If this parameter is "-", all IP addresses are disallowed.

RETURNS **OK**, or **ERROR** if the secret cannot be added to the table.

SEE ALSO **pppSecretLib**, **pppSecretDelete( )**, **pppSecretShow( )**

pppSecretDelete( )

NAME

pppSecretDelete( ) – delete a secret from the PPP authentication secrets table

SYNOPSIS

```
STATUS pppSecretDelete
    (
    char * client,            /* client being authenticated */
    char * server,            /* server performing authentication */
    char * secret             /* secret used for authentication */
    )
```

DESCRIPTION

This routine deletes a secret from the Point-to-Point Protocol (PPP) authentication secrets table. When searching for a secret to delete from the table, the wildcard substitution (using "*") is not performed for client and/or server names. The *client*, *server*, and *secret*strings must match the table entry exactly in order to be deleted.

RETURNS

OK, or **ERROR** if the table entry being deleted is not found.

SEE ALSO

pppSecretLib, **pppSecretAdd( )**, **pppSecretShow( )**

pppSecretShow( )

NAME

pppSecretShow( ) – display the PPP authentication secrets table

SYNOPSIS

```
void pppSecretShow (void)
```

DESCRIPTION

This routine displays the Point-to-Point Protocol (PPP) authentication secrets table. The information in the secrets table may be used by the Password Authentication Protocol (PAP) and Challenge-Handshake Authentication Protocol (CHAP) user authentication protocols.

RETURNS

N/A

SEE ALSO

pppShow, **pppLib**, **pppSecretAdd( )**, **pppSecretDelete( )**

pppstatGet( )

NAME **pppstatGet( )** – get PPP link statistics

SYNOPSIS

```
STATUS pppstatGet
    (
    int        unit,          /* PPP interface unit number to examine */
    PPP_STAT * pStat          /* PPP_STAT structure to be filled */
    )
```

DESCRIPTION This routine gets statistics for the specified Point-to-Point Protocol (PPP) link. Detailed are the numbers of bytes and packets received and sent through the PPP interface.

The PPP link statistics are returned through a **PPP_STAT** structure, which is defined in **h/netinet/ppp/pppShow.h**.

RETURNS **OK**, or **ERROR** if *unit* is an invalid PPP unit number.

SEE ALSO **pppShow**, **pppLib**

pppstatShow( )

NAME **pppstatShow( )** – display PPP link statistics

SYNOPSIS

```
void pppstatShow (void)
```

DESCRIPTION This routine displays statistics for each initialized Point-to-Point Protocol (PPP) link. Detailed are the numbers of bytes and packets received and sent through each PPP interface.

RETURNS N/A

SEE ALSO **pppShow**, **pppLib**

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.

SEE ALSO **fioLib**, **printf( )**

printErrno( )

NAME **printErrno( )** – print the definition of a specified error status value

SYNOPSIS

```
void printErrno
    (
    int errNo                  /* status code whose name is to be printed */
    )
```

DESCRIPTION This command displays the error-status string, corresponding to a specified error-status value. It is only useful if the error-status symbol table has been built and included in the system. If *errNo* is zero, then the current task status is used by calling **errnoGet( )**.

This facility is described in **errnoLib**.

RETURNS N/A

SEE ALSO **usrLib**, **errnoLib**, **errnoGet( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

printf( )

NAME

printf( ) – write a formatted string to the standard output stream (ANSI)

SYNOPSIS

```
int printf
    (
    const char * fmt,         /* format string to write */
                 ...          /* optional arguments to format string */
    )
```

DESCRIPTION

This routine writes output to standard output under control of the string *fmt*. The string *fmt* contains ordinary characters, which are written unchanged, plus conversion specifications, which cause the arguments that follow *fmt* to be converted and printed as part of the formatted string.

The number of arguments for the format is arbitrary, but they must correspond to the conversion specifications in *fmt*. If there are insufficient arguments, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored. The routine returns when the end of the format string is encountered.

The format is a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: ordinary multibyte characters (not %) that are copied unchanged to the output stream; and conversion specification, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the % character. After the %, the following appear in sequence:

– Zero or more flags (in any order) that modify the meaning of the conversion specification.

– An optional minimum field width. If the converted value has fewer characters than the field width, it will be padded with spaces (by default) on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of an asterisk (*) (described later) or a decimal integer.

– An optional precision that gives the minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, and **X** conversions, the number of digits to appear after the decimal-point character for **e**, **E**, and **f** conversions, the maximum number of significant digits for the **g** and **G** conversions, or the maximum number of characters to be written from a string in the **s** conversion. The precision takes the form of a period (**.**) followed either by an asterisk (*) (described later) or by an optional decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.

– An optional **h** specifying that a following **d**, **i**, **o**, **u**, **x**, and **X** conversion specifier applies to a **short int** or **unsigned short int** argument (the argument will have been

P

promoted according to the integral promotions, and its value converted to **short int** or **unsigned short int** before printing); an optional **h** specifying that a following **n** conversion specifier applies to a pointer to a **short int** argument. An optional **l** (ell) specifying that a following **d**, **i**, **o**, **u**, **x**, and **X** conversion specifier applies to a **long int** or **unsigned long int** argument; or an optional **l** specifying that a following **n** conversion specifier applies to a pointer to a **long int** argument. An optional **ll** (ell-ell) specifying that a following **d**, **i**, **o**, **u**, **x**, and **X** conversion specifier applies to a **long long int** or **unsigned long long int** argument; or an optional **ll** specifying that a following **n** conversion specifier applies to a pointer to a **long long int** argument. If a **h**, **l** or **ll** appears with any other conversion specifier, the behavior is undefined.

WARNING: ANSI C also specifies an optional **L** in some of the same contexts as **l** above, corresponding to a **long double** argument. However, the current release of VxWorks does not support **long double** data; using the optional **L** gives unpredictable results.

– A character that specifies the type of conversion to be applied.

As noted above, a field width, or precision, or both, can be indicated by an asterisk (*). In this case, an **int** argument supplies the field width or precision. The arguments specifying field width or precision, or both, should appear (in that order) before the argument to be converted. A negative field width argument is taken as a - flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

The flag characters and their meanings are:

-
: The result of the conversion will be left-justified within the field. (it will be right-justified if this flag is not specified.)

+
: The result of a signed conversion will always begin with a plus or minus sign. (It will begin with a sign only when a negative value is converted if this flag is not specified.)

space
: If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space will be prefixed to the result. If the **space** and **+** flags both appear, the **space** flag will be ignored.

#
: The result is to be converted to an "alternate form." For **o** conversion it increases the precision to force the first digit of the result to be a zero. For **x** (or **X**) conversion, a non-zero result will have "0x" (or "0X") prefixed to it. For **e**, **E**, **f**, **g**, and **g** conversions, the result will always contain a decimal-point character, even if no digits follow it. (Normally, a decimal-point character appears in the result of these conversions only if no digit follows it). For **g** and **G** conversions, trailing zeros will not be removed from the result. For other conversions, the behavior is undefined.

0
: For **d**, **i**, **o**, **u**, **x**, **X**, **e**, **E**, **f**, **g**, and **G** conversions, leading zeros (following any

indication of sign or base) are used to pad to the field width; no space padding is performed. If the **0** and **-**flags both appear, the **0** flag will be ignored. For **d**, **i**, **o**, **u**, **x**, and **X** conversions, if a precision is specified, the **0** flag will be ignored. For other conversions, the behavior is undefined.

The conversion specifiers and their meanings are:

d, **i**
: The **int** argument is converted to signed decimal in the style **[-]dddd**. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

o, **u**, **x**, **X**
: The **unsigned int** argument is converted to unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal notation (**x** or **X**) in the style **dddd**; the letters abcdef are used for **x** conversion and the letters ABCDEF for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

f
: The **double** argument is converted to decimal notation in the style **[-]ddd.ddd**, where the number of digits after the decimal point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the **#** flag is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

e, **E**
: The **double** argument is converted in the style **[-]d.ddde+/-dd**, where there is one digit before the decimal-point character (which is non-zero if the argument is non-zero) and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero and the **#** flag is not specified, no decimal-point character appears. The value is rounded to the appropriate number of digits. The **E** conversion specifier will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits. If the value is zero, the exponent is zero.

g, **G**
: The **double** argument is converted in style **f** or **e** (or in style **E** in the case of a **G** conversion specifier), with the precision specifying the number of significant digits. If the precision is zero, it is taken as 1. The style used depends on the value converted; style **e** (or **E**) will be used only if the exponent resulting from such a conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result; a decimal-point character appears only if it is followed

P

by a digit.

c
: The **int** argument is converted to **unsigned char**; the resulting character is written.

s
: The argument should be a pointer to an array of character type. Characters from the array are written up to (but not including) a terminating null character; if the precision is specified, no more than that many characters are written. If the precision is not specified or is greater than the size of the array, the array will contain a null character.

p
: The argument should be a pointer to **void**. The value of the pointer is converted to a sequence of printable characters, in hexadecimal representation (prefixed with "0x").

n
: The argument should be a pointer to an integer into which the number of characters written to the output stream so far by this call to **fprintf( )** is written. No argument is converted.

%
: A **%** is written. No argument is converted. The complete conversion specification is %%.

If a conversion specification is invalid, the behavior is undefined.

If any argument is, or points to, a union or an aggregate (except for an array of character type using **s** conversion, or a pointer using **p** conversion), the behavior is undefined.

In no case does a non-existent or small field width cause truncation of a field if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

INCLUDE FILES **fioLib.h**

RETURNS The number of characters written, or a negative value if an output error occurs.

SEE ALSO **fioLib**, **fprintf( )**, *American National Standard for Information Systems -Programming Language - C, ANSI X3.159-1989: Input/Output (**stdio.h**)*

printLogo( )

NAME **printLogo( )** – print the VxWorks logo

SYNOPSIS

```
void printLogo (void)
```

DESCRIPTION This command displays the VxWorks banner seen at boot time. It also displays the VxWorks version number and kernel version number.

RETURNS N/A

SEE ALSO **usrLib**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

proxyArpLibInit( )

NAME **proxyArpLibInit( )** – initialize proxy ARP

SYNOPSIS

```
STATUS proxyArpLibInit
    (
    int clientSizeLog2,       /* client table size as power of two */
    int portSizeLog2          /* port table size as power of two */
    )
```

DESCRIPTION This routine starts the proxy ARP server by initializing the required data structures and installing the necessary input hooks. It should be called only once; subsequent calls have no effect. The *clientSizeLog2* and *portSizeLog2* parameters specify the internal hash table sizes. Each must be equal to a power of two, or zero to use a default size value.

RETURNS **OK**, or **ERROR** if unsuccessful.

SEE ALSO **proxyArpLib**

proxyNetCreate( )

NAME

proxyNetCreate( ) – create a proxy ARP network

SYNOPSIS

```
STATUS proxyNetCreate
    (
    char * proxyAddr,          /* address of proxy network interface */
    char * mainAddr            /* address of main network interface */
    )
```

DESCRIPTION

This routine activates proxy services between the proxy network connected to the interface with the *proxyAddr* IP address and the main network connected to the interface with the *mainAddr* address. Once registration is complete, the proxy server will disguise the physically separated networks as a single logical network.

The corresponding interfaces must be attached and configured with IP addresses before calling this routine. If the proxy network shares the same logical subnet number as the main network, the corresponding interface to the proxy network must use a value of 255.255.255.255 for the netmask.

RETURNS

OK, or **ERROR** if unsuccessful.

ERRNO

S_proxyArpLib_INVALID_ADDRESS

SEE ALSO

proxyArpLib

proxyNetDelete( )

NAME

proxyNetDelete( ) – delete a proxy network

SYNOPSIS

```
STATUS proxyNetDelete
    (
    char * proxyAddr           /* proxy net address */
    )
```

DESCRIPTION

This routine deletes the proxy network specified by *proxyAddr*. It also removes all the proxy clients that exist on that network.

RETURNS

OK, or **ERROR** if unsuccessful.

SEE ALSO

proxyArpLib

proxyNetShow( )

NAME **proxyNetShow( )** – show proxy ARP networks

SYNOPSIS

```
void proxyNetShow (void)
```

DESCRIPTION This routine displays the proxy networks and their associated clients.

EXAMPLE

```
-> proxyNetShow
main interface 147.11.1.182 proxy interface 147.11.1.183
   client 147.11.1.184
```

RETURNS N/A

SEE ALSO **proxyArpLib**

proxyPortFwdOff( )

NAME **proxyPortFwdOff( )** – disable broadcast forwarding for a particular port

SYNOPSIS

```
STATUS proxyPortFwdOff
    (
    int port                  /* port number */
    )
```

DESCRIPTION This routine disables broadcast forwarding on port number *port*. To disable the (previously enabled) forwarding of all ports via **proxyPortFwdOn( )**, specify zero for *port*.

RETURNS **OK**, or **ERROR** if unsuccessful.

SEE ALSO **proxyArpLib**

P

proxyPortFwdOn( )

NAME **proxyPortFwdOn( )** – enable broadcast forwarding for a particular port

SYNOPSIS

```
STATUS proxyPortFwdOn
    (
    int port                    /* port number */
    )
```

DESCRIPTION This routine enables broadcasts destined for the port, *port*, to be forwarded to and from the proxy network. To enable all ports, specify zero for *port*.

RETURNS **OK**, or **ERROR** if unsuccessful.

SEE ALSO **proxyArpLib**

proxyPortShow( )

NAME **proxyPortShow( )** – show ports enabled for broadcast forwarding

SYNOPSIS

```
void proxyPortShow (void)
```

DESCRIPTION This routine displays the destination ports for which the proxy ARP server will forward broadcast messages between the physically separate networks.

EXAMPLE

```
-> proxyPortShow
enabled ports:
   port 67
```

RETURNS N/A

SEE ALSO **proxyArpLib**

proxyReg( )

NAME **proxyReg( )** – register a proxy client

SYNOPSIS

```
STATUS proxyReg
    (
    char * ifName,            /* interface name */
    char * proxyAddr          /* proxy address */
    )
```

DESCRIPTION This routine sends a message over the network interface *ifName* to register *proxyAddr* as a proxy client.

RETURNS **OK**, or **ERROR** if unsuccessful.

SEE ALSO **proxyLib**

proxyUnreg( )

NAME **proxyUnreg( )** – unregister a proxy client

SYNOPSIS

```
STATUS proxyUnreg
    (
    char * ifName,            /* interface name */
    char * proxyAddr          /* proxy address */
    )
```

DESCRIPTION This routine sends a message over the network interface *ifName* to unregister *proxyAddr* as a proxy client.

RETURNS **OK**, or **ERROR** if unsuccessful.

SEE ALSO **proxyLib**

psrShow( )

NAME **psrShow( )** – display the meaning of a specified **psr** value, symbolically (ARM)

SYNOPSIS

```
void psrShow
    (
    ULONG psrValue            /* psr value to show */
    )
```

DESCRIPTION This routine displays the meaning of all the fields in a specified **psr** value, symbolically.

Extracted from **psl.h**:

```
Definition of bits in the Sun-4 PSR (Processor Status Register)
 ------------------------------------------------------------------------
| IMPL | VER |      ICC        | resvd | EC | EF | PIL | S | PS | ET | CWP |
|      |     | N | Z | V | C |       |    |    |     |   |    |    |     |
|------|-----|---|---|---|---|-------|----|----|-----|---|----|----|-----|
 31  28 27 24  23  22  21  20 19   14  13   12  11  8   7   6    5  4   0
```

For compatibility with future revisions, reserved bits are defined to be initialized to zero and, if written, must be preserved.

EXAMPLE

```
 -> psrShow 0x00001FE7
Implementation 0, mask version 0:
Fujitsu MB86900 or LSI L64801, 7 windows
        no SWAP, FSQRT, CP, extended fp instructions
    Condition codes: . . . .
    Coprocessor enables: . EF
    Processor interrupt level: f
    Flags: S PS ET
    Current window pointer: 0x07
 ->
```

RETURNS N/A

SEE ALSO **dbgArchLib**, **psr( )**, *ARM Architecture Reference Manual*

pthreadLibInit( )

NAME **pthreadLibInit( )** – initialize POSIX threads support

SYNOPSIS

```
void pthreadLibInit (void)
```

DESCRIPTION This routine initializes the POSIX threads (*pthreads*) support for VxWorks. It should be called before any POSIX threads functions are used; normally it will be called as part of the kernel's initialization sequence.

RETURNS 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 re-initialized.

RETURNS On success zero; on failure a non-zero error code.

ERRNOS EINVAL

SEE ALSO **pthreadLib**, **pthread_attr_init( )**

P

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

Always returns zero.

ERRNOS

None.

SEE ALSO

pthreadLib, **pthread_attr_init( )**, **pthread_attr_setdetachstate( )**

pthread_attr_getinheritsched( )

NAME

pthread_attr_getinheritsched( ) – get value of inheritsched attribute in thread attributes object (POSIX)

SYNOPSIS

```
int pthread_attr_getinheritsched
    (
    const pthread_attr_t * pAttr,        /* thread attributes object */
    int *                  pInheritsched /* inheritance mode (out) */
    )
```

DESCRIPTION

This routine gets the scheduling inheritance value from the thread attributes object *pAttr*.

Possible values are:

PTHREAD_INHERIT_SCHED
Inherit scheduling parameters from parent thread.

PTHREAD_EXPLICIT_SCHED
Use explicitly provided scheduling parameters (*i.e.*, those specified in the thread attributes object).

RETURNS On success zero; on failure a non-zero error code.

ERRNOS EINVAL

SEE ALSO **pthreadLib**, **pthread_attr_init( )**, **pthread_attr_getschedparam( )**, **pthread_attr_getschedpolicy( )**, **pthread_attr_setinheritsched( )**

pthread_attr_getname( )

NAME **pthread_attr_getname( )** – get name of thread attribute object

SYNOPSIS

```
int pthread_attr_getname
    (
    pthread_attr_t * pAttr,
    char *           *name
    )
```

DESCRIPTION This routine gets the name in the specified thread attributes object, *pAttr*.

RETURNS Always returns zero

ERRNOS None.

SEE ALSO **pthreadLib**, **pthread_attr_setname( )**,

pthread_attr_getschedparam( )

NAME **pthread_attr_getschedparam( )** – get value of schedparam attribute from thread attributes object (POSIX)

SYNOPSIS

```
int pthread_attr_getschedparam
    (
    const pthread_attr_t * pAttr, /* thread attributes */
    struct sched_param *   pParam /* current parameters (out) */
    )
```

P

DESCRIPTION Return, via the pointer *pParam*, the current scheduling parameters from the thread attributes object *pAttr*.

RETURNS On success zero; on failure a non-zero error code.

ERRNOS EINVAL

SEE ALSO **pthreadLib**, **schedPxLib**, **pthread_attr_init( )**, **pthread_attr_setschedparam( )**, **pthread_getschedparam( )**, **pthread_setschedparam( )**, **sched_getparam( )**, **sched_setparam( )**

pthread_attr_getschedpolicy( )

NAME **pthread_attr_getschedpolicy( )** – get schedpolicy attribute from thread attributes object (POSIX)

SYNOPSIS

```
int pthread_attr_getschedpolicy
    (
    const pthread_attr_t * pAttr,  /* thread attributes */
    int *                  pPolicy /* current policy (out) */
    )
```

DESCRIPTION This routine returns, via the pointer *pPolicy*, the current scheduling policy in the thread attributes object specified by *pAttr*. Possible values for VxWorks systems are **SCHED_RR** and **SCHED_FIFO**; **SCHED_OTHER** is not supported.

RETURNS On success zero; on failure a non-zero error code.

ERRNOS EINVAL

SEE ALSO **pthreadLib**, **schedPxLib**, **pthread_attr_init( )**, **pthread_attr_setschedpolicy( )**, **pthread_getschedparam( )**, **pthread_setschedparam( )**, **sched_setscheduler( )**, **sched_getscheduler( )**

pthread_attr_getscope( )

NAME

pthread_attr_getscope( ) – get contention scope from thread attributes (POSIX)

SYNOPSIS

```
int pthread_attr_getscope
    (
    const pthread_attr_t * pAttr,            /* thread attributes object */
    int *                  pContentionScope /* contention scope (out) */
    )
```

DESCRIPTION

Reads the current contention scope setting from a thread attributes object. For VxWorks this is always **PTHREAD_SCOPE_SYSTEM**. If the thread attributes object is uninitialized then **EINVAL** will be returned. The contention scope is returned in the location pointed to by *pContentionScope*.

RETURNS

On success zero; on failure a non-zero error code.

ERRNOS

EINVAL

SEE ALSO

pthreadLib, **pthread_attr_init( )**, **pthread_attr_setscope( )**

P

pthread_attr_getstackaddr( )

NAME

pthread_attr_getstackaddr( ) – get value of stackaddr attribute from thread attributes object (POSIX)

SYNOPSIS

```
int pthread_attr_getstackaddr
    (
    const pthread_attr_t * pAttr,        /* thread attributes */
    void *                 *ppStackaddr /* current stack address (out) */
    )
```

DESCRIPTION

This routine returns the stack address from the thread attributes object *pAttr* in the location pointed to by *ppStackaddr*.

RETURNS

Always returns zero.

ERRNOS

None.

SEE ALSO

pthreadLib, **pthread_attr_init( )**, **pthread_attr_setstacksize( )**

pthread_attr_getstacksize( )

NAME

pthread_attr_getstacksize( ) – get stack value of stacksize attribute from thread attributes object (POSIX)

SYNOPSIS

```
int pthread_attr_getstacksize
    (
    const pthread_attr_t * pAttr,     /* thread attributes */
    size_t *               pStacksize /* current stack size (out) */
    )
```

DESCRIPTION

This routine gets the current stack size from the thread attributes object *pAttr* and places it in the location pointed to by *pStacksize*.

RETURNS

Always returns zero.

ERRNOS

None.

SEE ALSO

pthreadLib, **pthread_attr_init( )**, **pthread_attr_setstacksize( )**

pthread_attr_init( )

NAME

pthread_attr_init( ) – initialize thread attributes object (POSIX)

SYNOPSIS

```
int pthread_attr_init
    (
    pthread_attr_t * pAttr    /* thread attributes */
    )
```

DESCRIPTION

This routine initializes a thread attributes object. If *pAttr* is **NULL** then this function will return **EINVAL**.

The attributes that are set by default are as follows:

Stack Address
: **NULL** - allow the system to allocate the stack.

Stack Size
: 0 - use the VxWorks **taskLib** default stack size.

Detach State
: **PTHREAD_CREATE_JOINABLE**

Contention Scope
PTHREAD_SCOPE_SYSTEM

Scheduling Inheritance
PTHREAD_INHERIT_SCHED

Scheduling Policy
SCHED_RR

Scheduling Priority
Use **pthreadLib** default priority

Note that the scheduling policy and priority values are only used if the scheduling inheritance mode is changed to **PTHREAD_EXPLICIT_SCHED** - see **pthread_attr_setinheritsched( )** for information.

RETURNS On success zero; on failure a non-zero error code.

ERRNOS **EINVAL**

SEE ALSO **pthreadLib**, **pthread_attr_destroy( )**, **pthread_attr_getdetachstate( )**, **pthread_attr_getinheritsched( )**, **pthread_attr_getschedparam( )**, **pthread_attr_getschedpolicy( )**, **pthread_attr_getscope( )**, **pthread_attr_getstackaddr( )**, **pthread_attr_getstacksize( )**, **pthread_attr_setdetachstate( )**, **pthread_attr_setinheritsched( )**, **pthread_attr_setschedparam( )**, **pthread_attr_setschedpolicy( )**, **pthread_attr_setscope( )**, **pthread_attr_setstackaddr( )**, **pthread_attr_setstacksize( )**

P

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 a non-zero error code.

ERRNOS EINVAL

SEE ALSO **pthreadLib**, **pthread_attr_getdetachstate( )**, **pthread_attr_init( )**

pthread_attr_setinheritsched( )

NAME **pthread_attr_setinheritsched( )** – set inheritsched attribute in thread attribute object (POSIX)

SYNOPSIS

```
int pthread_attr_setinheritsched
    (
    pthread_attr_t * pAttr,       /* thread attributes object */
    int              inheritsched /* inheritance mode */
    )
```

DESCRIPTION This routine sets the scheduling inheritance to be used when creating a thread with the thread attributes object specified by *pAttr*.

Possible values are:

PTHREAD_INHERIT_SCHED
Inherit scheduling parameters from parent thread.

PTHREAD_EXPLICIT_SCHED
Use explicitly provided scheduling parameters (*i.e.*, those specified in the thread attributes object).

RETURNS On success zero; on failure a non-zero error code.

ERRNOS EINVAL

SEE ALSO **pthreadLib**, **pthread_attr_getinheritsched( )**, **pthread_attr_init( )**, **pthread_attr_setschedparam( )**, **pthread_attr_setschedpolicy( )**

pthread_attr_setname( )

NAME

pthread_attr_setname( ) – set name in thread attribute object

SYNOPSIS

```
int pthread_attr_setname
    (
    pthread_attr_t * pAttr,
    char *           name
    )
```

DESCRIPTION

This routine sets the name in the specified thread attributes object, *pAttr*.

RETURNS

Always returns zero.

ERRNOS

None.

SEE ALSO

pthreadLib, **pthread_attr_getname( )**

pthread_attr_setschedparam( )

NAME

pthread_attr_setschedparam( ) – set schedparam attribute in thread attributes object (POSIX)

SYNOPSIS

```
int pthread_attr_setschedparam
    (
    pthread_attr_t *           pAttr, /* thread attributes */
    const struct sched_param * pParam /* new parameters */
    )
```

DESCRIPTION

Set the scheduling parameters in the thread attributes object *pAttr*. The scheduling parameters are essentially the thread's priority.

RETURNS

On success zero; on failure a non-zero error code.

ERRNOS

EINVAL

SEE ALSO

pthreadLib, **schedPxLib**, **pthread_attr_getschedparam( )**, **pthread_attr_init( )**, **pthread_getschedparam( )**, **pthread_setschedparam( )**, **sched_getparam( )**, **sched_setparam( )**

P

pthread_attr_setschedpolicy( )

NAME
pthread_attr_setschedpolicy( ) – set schedpolicy attribute in thread attributes object (POSIX)

SYNOPSIS

```
int pthread_attr_setschedpolicy
    (
    pthread_attr_t * pAttr,   /* thread attributes */
    int              policy   /* new policy */
    )
```

DESCRIPTION
Select the thread scheduling policy. The default scheduling policy is to inherit the current system setting. Unlike the POSIX model, scheduling policies under VxWorks are global. If a scheduling policy is being set explicitly, the **PTHREAD_EXPLICIT_SCHED** mode must be set (see **pthread_attr_setinheritsched( )** for information), and the selected scheduling policy must match the global scheduling policy in place at the time; failure to do so will result in **pthread_create( )** failing with the non-POSIX error **ENOTTY**.

POSIX defines the following policies:

SCHED_RR
Real-time, round-robin scheduling.

SCHED_FIFO
Real-time, first-in first-out scheduling.

SCHED_OTHER
Other, non-real-time scheduling.

VxWorks only supports **SCHED_RR** and **SCHED_FIFO**.

RETURNS
On success zero; on failure a non-zero error code.

ERRNOS
EINVAL

SEE ALSO
pthreadLib, **schedPxLib**, **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 a non-zero error code.

ERRNOS **EINVAL**

SEE ALSO **pthreadLib**, **pthread_attr_getscope( )**, **pthread_attr_init( )**

pthread_attr_setstackaddr( )

NAME **pthread_attr_setstackaddr( )** – set stackaddr attribute in thread attributes object (POSIX)

SYNOPSIS

```
int pthread_attr_setstackaddr
    (
    pthread_attr_t * pAttr,     /* thread attributes */
    void *           pStackaddr /* new stack address */
    )
```

DESCRIPTION This routine sets the stack address in the thread attributes object *pAttr* to be *pStackaddr*.

Note that the size of this stack must be large enough to also include the task's TCB. The size of the TCB varies by architecture but can be determined by calling **sizeof (WIND_TCB)**. Set stack size using the routine **pthread_attr_setstacksize( )**.

RETURNS Zero, always.

ERRNOS None.

SEE ALSO **pthreadLib**, **pthread_attr_getstacksize( )**, **pthread_attr_setstacksize( )**, **pthread_attr_init( )**

pthread_attr_setstacksize( )

NAME

pthread_attr_setstacksize( ) – set stacksize attribute in thread attributes object (POSIX)

SYNOPSIS

```
int pthread_attr_setstacksize
    (
    pthread_attr_t * pAttr,    /* thread attributes */
    size_t           stacksize /* new stack size */
    )
```

DESCRIPTION

This routine sets the thread stack size in the specified thread attributes object, *pAttr*.

RETURNS

Always returns zero.

ERRNOS

None.

SEE ALSO

pthreadLib, **pthread_attr_getstacksize( )**, **pthread_attr_init( )**

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**.

NOTE: In VxWorks, asynchronous thread cancellation is accomplished using a signal. The signal **SIGCNCL** has been reserved for this purpose. Applications should take care not to block or handle this signal.

RETURNS

On success zero; on failure a non-zero error code.

ERRNOS

ESRCH

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 is no longer called when the thread exits.

RETURNS

N/A

ERRNOS

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

ERRNOS

N/A

SEE ALSO

pthreadLib, **pthread_cleanup_pop( )**, **pthread_exit( )**

pthread_cond_broadcast( )

NAME

pthread_cond_broadcast( ) – unblock all threads waiting on a condition (POSIX)

SYNOPSIS

```
int pthread_cond_broadcast
    (
    pthread_cond_t * pCond
    )
```

DESCRIPTION

This function unblocks all threads blocked on the condition variable *pCond*. Nothing happens if no threads are waiting on the specified condition variable.

RETURNS

On success zero; on failure a non-zero error code.

ERRNOS

EINVAL

SEE ALSO

pthreadLib, **pthread_condattr_init( )**, **pthread_condattr_destroy( )**, **pthread_cond_destroy( )**, **pthread_cond_init( )**, **pthread_cond_signal( )**, **pthread_cond_timedwait( )**, **pthread_cond_wait( )**

pthread_cond_destroy( )

NAME

pthread_cond_destroy( ) – destroy a condition variable (POSIX)

SYNOPSIS

```
int pthread_cond_destroy
    (
    pthread_cond_t * pCond    /* condition variable */
    )
```

DESCRIPTION

This routine destroys the condition variable pointed to by *pCond*. No threads can be waiting on the condition variable when this function is called. If there are threads waiting on the condition variable, then **pthread_cond_destroy( )** returns **EBUSY**.

RETURNS

On success zero; on failure a non-zero error code.

ERRNOS

EINVAL, **EBUSY**

SEE ALSO

pthreadLib, **pthread_condattr_init( )**, **pthread_condattr_destroy( )**, **pthread_cond_broadcast( )**, **pthread_cond_init( )**, **pthread_cond_signal( )**, **pthread_cond_timedwait( )**, **pthread_cond_wait( )**

pthread_cond_init( )

NAME

pthread_cond_init( ) – initialize condition variable (POSIX)

SYNOPSIS

```
int pthread_cond_init
    (
    pthread_cond_t *     pCond, /* condition variable */
    pthread_condattr_t * pAttr  /* condition variable attributes */
    )
```

DESCRIPTION

This function initializes a condition variable. A condition variable is a synchronization device that allows threads to block until some predicate on shared data is satisfied. The basic operations on conditions are to signal the condition (when the predicate becomes true), and wait for the condition, blocking the thread until another thread signals the condition.

A condition variable must always be associated with a mutex to avoid a race condition between the wait and signal operations.

If *pAttr* is **NULL** then the default attributes are used as specified by POSIX; if *pAttr* is non-**NULL** then it is assumed to point to a condition attributes object initialized by **pthread_condattr_init( )**, and those are the attributes used to create the condition variable.

RETURNS

On success zero; on failure a non-zero error code.

ERRNOS

EINVAL, **EBUSY**

SEE ALSO

pthreadLib, **pthread_condattr_init( )**, **pthread_condattr_destroy( )**, **pthread_cond_broadcast( )**, **pthread_cond_destroy( )**, **pthread_cond_signal( )**, **pthread_cond_timedwait( )**, **pthread_cond_wait( )**

pthread_cond_signal( )

NAME

pthread_cond_signal( ) – unblock a thread waiting on a condition (POSIX)

SYNOPSIS

```
int pthread_cond_signal
    (
    pthread_cond_t * pCond
    )
```

DESCRIPTION

This routine unblocks one thread waiting on the specified condition variable *pCond*. If no threads are waiting on the condition variable then this routine does nothing; if more than one thread is waiting, then one will be released, but it is not specified which one.

RETURNS

On success zero; on failure a non-zero error code.

ERRNOS

EINVAL

SEE ALSO

pthreadLib, **pthread_condattr_init( )**, **pthread_condattr_destroy( )**, **pthread_cond_broadcast( )**, **pthread_cond_destroy( )**, **pthread_cond_init( )**, **pthread_cond_timedwait( )**, **pthread_cond_wait( )**

pthread_cond_timedwait( )

NAME

pthread_cond_timedwait( ) – wait for a condition variable with a timeout (POSIX)

SYNOPSIS

```
int pthread_cond_timedwait
    (
    pthread_cond_t *        pCond,   /* condition variable */
    pthread_mutex_t *       pMutex,  /* POSIX mutex */
    const struct timespec * pAbstime /* timeout time */
    )
```

DESCRIPTION

This function atomically releases the mutex *pMutex* and waits for another thread to signal the condition variable *pCond*. As with **pthread_cond_wait( )**, the mutex must be locked by the calling thread when **pthread_cond_timedwait( )** is called.

If the condition variable is signalled before the system time reaches the time specified by *pAbsTime*, then the mutex is re-acquired and the calling thread unblocked.

If the system time reaches or exceeds the time specified by *pAbsTime* before the condition is signalled, then the mutex is re-acquired, the thread unblocked and **ETIMEDOUT** returned.

NOTE: The timeout is specified as an absolute value of the system clock in a *timespec* structure (see **clock_gettime( )** for more information). This is different from most VxWorks timeouts which are specified in ticks relative to the current time.

RETURNS On success zero; on failure a non-zero error code.

ERRNOS **EINVAL**, **ETIMEDOUT**

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 *  pCond,   /* condition variable */
    pthread_mutex_t * pMutex   /* POSIX mutex */
    )
```

DESCRIPTION This function atomically releases the mutex *pMutex* and waits for the condition variable *pCond* to be signalled by another thread. The mutex must be locked by the calling thread when **pthread_cond_wait( )** is called; if it is not then this function returns an error (**EINVAL**).

Before returning to the calling thread, **pthread_cond_wait( )** re-acquires the mutex.

RETURNS On success zero; on failure a non-zero error code.

ERRNOS **EINVAL**

SEE ALSO **pthreadLib**, **pthread_condattr_init( )**, **pthread_condattr_destroy( )**, **pthread_cond_broadcast( )**, **pthread_cond_destroy( )**, **pthread_cond_init( )**, **pthread_cond_signal( )**, **pthread_cond_timedwait( )**

P

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 re-initialized.

RETURNS Always returns zero.

ERRNOS None.

SEE ALSO **pthreadLib**, **pthread_cond_init( )**, **pthread_condattr_init( )**

pthread_condattr_init( )

NAME **pthread_condattr_init( )** – initialize a condition attribute object (POSIX)

SYNOPSIS

```
int pthread_condattr_init
    (
    pthread_condattr_t * pAttr /* condition variable attributes */
    )
```

DESCRIPTION This routine initializes the condition attribute object *pAttr* and fills it with default values for the attributes.

RETURNS On success zero; on failure a non-zero error code.

ERRNOS **EINVAL**

SEE ALSO **pthreadLib**, **pthread_cond_init( )**, **pthread_condattr_destroy( )**

pthread_create( )

NAME **pthread_create( )** – create a thread (POSIX)

SYNOPSIS

```
int pthread_create
    (
    pthread_t *             pThread,    /* Thread ID (out) */
    const pthread_attr_t * pAttr,       /* Thread attributes object */
    void * (* startRoutine)(void * ),   /* Entry function */
    void *                  arg         /* Entry function argument */
    )
```

DESCRIPTION This routine creates a new thread and if successful writes its ID into the location pointed to by *pThread*. If *pAttr* is **NULL** then default attributes are used. The new thread executes *startRoutine* with *arg* as its argument.

RETURNS On success zero; on failure a non-zero error code.

ERRNOS EINVAL, EAGAIN

SEE ALSO **pthreadLib**, **pthread_exit( )**, **pthread_join( )**

P

pthread_detach( )

NAME **pthread_detach( )** – dynamically detach a thread (POSIX)

SYNOPSIS

```
int pthread_detach
    (
    pthread_t thread          /* thread to detach */
    )
```

DESCRIPTION This routine puts the thread *thread* into the detached state. This prevents other threads from synchronizing on the termination of the thread using **pthread_join( )**.

RETURNS On success zero; on failure a non-zero error code.

ERRNOS EINVAL, ESRCH

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.

SEE ALSO

pthreadLib

pthread_exit( )

NAME

pthread_exit( ) – terminate a thread (POSIX)

SYNOPSIS

```
void pthread_exit
    (
    void * status             /* exit status */
    )
```

DESCRIPTION

This function terminates the calling thread. All cleanup handlers that have been set for the calling thread with **pthread_cleanup_push( )** are executed in reverse order (the most recently added handler is executed first). Termination functions for thread-specific data are then called for all keys that have non-**NULL** values associated with them in the calling thread (see **pthread_key_create( )** for more details). Finally, execution of the calling thread is stopped.

The *status* argument is the return value of the thread and can be consulted from another thread using **pthread_join( )** unless this thread was detached (*i.e.,* a call to **pthread_detach( )** had been made for it, or it was created in the detached state).

All threads that remain *joinable* at the time they exit should ensure that **pthread_join( )** is called on their behalf by another thread to reclaim the resources that they hold.

RETURNS

Does not return.

ERRNOS
N/A

SEE ALSO
pthreadLib, **pthread_cleanup_push( )**, **pthread_detach( )**, **pthread_join( )**, **pthread_key_create( )**

pthread_getschedparam( )

NAME
pthread_getschedparam( ) – get value of schedparam attribute from a thread (POSIX)

SYNOPSIS
```
int pthread_getschedparam
    (
    pthread_t            thread,  /* thread */
    int *                pPolicy, /* current policy (out) */
    struct sched_param * pParam   /* current parameters (out) */
    )
```

DESCRIPTION
This routine reads the current scheduling parameters and policy of the thread specified by *thread*. The information is returned via *pPolicy* and *pParam*.

RETURNS
On success zero; on failure a non-zero error code.

ERRNOS
ESRCH

SEE ALSO
pthreadLib, **schedPxLib**, **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**.

ERRNOS
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**, 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 a non-zero error code.

ERRNOS
EINVAL, **ESRCH**, **EDEADLK**

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.

RETURNS On success zero; on failure a non-zero error code.

ERRNOS **EAGAIN**

SEE ALSO **pthreadLib**, **pthread_getspecific( )**, **pthread_key_delete( )**, **pthread_setspecific( )**

P

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.

RETURNS On success zero; on failure a non-zero error code.

ERRNOS EINVAL

SEE ALSO **pthreadLib**, **pthread_key_create( )**

pthread_kill( )

NAME **pthread_kill( )** – send a signal to a thread (POSIX)

SYNOPSIS

```
int pthread_kill
    (
    pthread_t thread,         /* thread to signal */
    int       sig             /* signal to send */
    )
```

DESCRIPTION This routine sends signal number *sig* to the thread specified by *thread*. The signal is delivered and handled as described for the **kill( )** function.

RETURNS On success zero; on failure a non-zero error code.

ERRNOS ESRCH, EINVAL

SEE ALSO **pthreadLib**, **kill( )**, **pthread_sigmask( )**, **sigprocmask( )**, **sigaction( )**, **sigsuspend( )**, **sigwait( )**

pthread_mutex_destroy( )

NAME **pthread_mutex_destroy( )** – destroy a mutex (POSIX)

SYNOPSIS

```
int pthread_mutex_destroy
    (
    pthread_mutex_t * pMutex  /* POSIX mutex */
    )
```

DESCRIPTION This routine destroys a mutex object, freeing the resources it might hold. The mutex must be unlocked when this function is called, otherwise it will return an error (**EBUSY**).

RETURNS On success zero; on failure a non-zero error code.

ERRNOS EINVAL, EBUSY

SEE ALSO **pthreadLib**, **semLib**, **semMLib**, **pthread_mutex_init( )**, **pthread_mutex_lock( )**, **pthread_mutex_trylock( )**, **pthread_mutex_unlock( )**, **pthread_mutexattr_init( )**, **semDelete( )**

pthread_mutex_getprioceiling( )

NAME **pthread_mutex_getprioceiling( )** – get the value of the prioceiling attribute of a mutex (POSIX)

SYNOPSIS

```
int pthread_mutex_getprioceiling
    (
    pthread_mutex_t * pMutex,      /* POSIX mutex */
    int *             pPrioceiling /* current priority ceiling (out) */
    )
```

DESCRIPTION This function gets the current value of the prioceiling attribute of a mutex. Unless the mutex was created with a protocol attribute value of **PTHREAD_PRIO_PROTECT**, this value is meaningless.

RETURNS On success zero; on failure a non-zero error code.

ERRNOS EINVAL

SEE ALSO **pthreadLib**, **pthread_mutex_setprioceiling( )**, **pthread_mutexattr_getprioceiling( )**, **pthread_mutexattr_setprioceiling( )**

pthread_mutex_init( )

NAME **pthread_mutex_init( )** – initialize mutex from attributes object (POSIX)

SYNOPSIS

```
int pthread_mutex_init
    (
    pthread_mutex_t *           pMutex, /* POSIX mutex */
    const pthread_mutexattr_t * pAttr   /* mutex attributes */
    )
```

DESCRIPTION

This routine initializes the mutex object pointed to by *pMutex* according to the mutex attributes specified in *pAttr*. If *pAttr* is **NULL**, default attributes are used as defined in the POSIX specification.

RETURNS

On success zero; on failure a non-zero error code.

ERRNOS

EINVAL, **EBUSY**

SEE ALSO

pthreadLib, **semLib**, **semMLib**, **pthread_mutex_destroy( )**, **pthread_mutex_lock( )**, **pthread_mutex_trylock( )**, **pthread_mutex_unlock( )**, **pthread_mutexattr_init( )**, **semMCreate( )**

pthread_mutex_lock( )

NAME

pthread_mutex_lock( ) – lock a mutex (POSIX)

SYNOPSIS

```
int pthread_mutex_lock
    (
    pthread_mutex_t * pMutex  /* POSIX mutex */
    )
```

DESCRIPTION

This routine locks the mutex specified by *pMutex*. If the mutex is currently unlocked, it becomes locked, and is said to be owned by the calling thread. In this case **pthread_mutex_lock( )** returns immediately.

If the mutex is already locked by another thread, **pthread_mutex_lock( )** blocks the calling thread until the mutex is unlocked by its current owner.

If it is already locked by the calling thread, **pthread_mutex_lock** will deadlock on itself and the thread will block indefinitely.

RETURNS

On success zero; on failure a non-zero error code.

ERRNOS

EINVAL

SEE ALSO

pthreadLib, **semLib**, **semMLib**, **pthread_mutex_init( )**, **pthread_mutex_lock( )**, **pthread_mutex_trylock( )**, **pthread_mutex_unlock( )**, **pthread_mutexattr_init( )**, **semTake( )**

pthread_mutex_setprioceiling( )

NAME

pthread_mutex_setprioceiling( ) – dynamically set the prioceiling attribute of a mutex (POSIX)

SYNOPSIS

```
int pthread_mutex_setprioceiling
    (
    pthread_mutex_t * pMutex,          /* POSIX mutex */
    int               prioceiling,     /* new priority ceiling */
    int *             pOldPrioceiling /* old priority ceiling (out) */
    )
```

DESCRIPTION

This function dynamically sets the value of the prioceiling attribute of a mutex. Unless the mutex was created with a protocol value of **PTHREAD_PRIO_PROTECT**, this function does nothing.

RETURNS

On success zero; on failure a non-zero error code.

ERRNOS

EINVAL, EPERM, S_objLib_OBJ_ID_ERROR, S_semLib_NOT_ISR_CALLABLE

SEE ALSO

pthreadLib, **pthread_mutex_getprioceiling( )**, **pthread_mutexattr_getprioceiling( )**, **pthread_mutexattr_setprioceiling( )**

P

pthread_mutex_trylock( )

NAME

pthread_mutex_trylock( ) – lock mutex if it is available (POSIX)

SYNOPSIS

```
int pthread_mutex_trylock
    (
    pthread_mutex_t * pMutex  /* POSIX mutex */
    )
```

DESCRIPTION

This routine locks the mutex specified by *pMutex*. If the mutex is currently unlocked, it becomes locked and owned by the calling thread. In this case **pthread_mutex_trylock( )** returns immediately.

If the mutex is already locked by another thread, **pthread_mutex_trylock( )** returns immediately with the error code **EBUSY**.

RETURNS

On success zero; on failure a non-zero error code.

ERRNOS **EINVAL, EBUSY**

SEE ALSO **pthreadLib**, **semLib**, **semMLib**, **pthread_mutex_init( )**, **pthread_mutex_lock( )**, **pthread_mutex_trylock( )**, **pthread_mutex_unlock( )**, **pthread_mutexattr_init( )**, **semTake( )**

pthread_mutex_unlock( )

NAME **pthread_mutex_unlock( )** – unlock a mutex (POSIX)

SYNOPSIS

```
int pthread_mutex_unlock
    (
    pthread_mutex_t * pMutex
    )
```

DESCRIPTION This routine unlocks the mutex specified by *pMutex*. If the calling thread is not the current owner of the mutex, **pthread_mutex_unlock( )** returns with the error code **EPERM**.

RETURNS On success zero; on failure a non-zero error code.

ERRNOS **EINVAL, EPERM, S_objLib_OBJ_ID_ERROR, S_semLib_NOT_ISR_CALLABLE**

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 re-initialized.

RETURNS On success zero; on failure a non-zero error code.

ERRNOS **EINVAL**

SEE ALSO **pthreadLib**, **pthread_mutexattr_getprioceiling( )**, **pthread_mutexattr_getprotocol( )**, **pthread_mutexattr_init( )**, **pthread_mutexattr_setprioceiling( )**, **pthread_mutexattr_setprotocol( )**, **pthread_mutex_init( )**

---

pthread_mutexattr_getprioceiling( )

NAME **pthread_mutexattr_getprioceiling( )** – get the current value of the prioceiling attribute in a mutex attributes object (POSIX)

SYNOPSIS

```
int pthread_mutexattr_getprioceiling
    (
    pthread_mutexattr_t * pAttr,       /* mutex attributes */
    int *                 pPrioceiling /* current priority ceiling (out) */
    )
```

DESCRIPTION This function gets the current value of the prioceiling attribute in a mutex attributes object. Unless the value of the protocol attribute is **PTHREAD_PRIO_PROTECT**, this value is ignored.

RETURNS On success zero; on failure a non-zero error code.

ERRNOS **EINVAL**

SEE ALSO **pthreadLib**, **pthread_mutexattr_destroy( )**, **pthread_mutexattr_getprotocol( )**, **pthread_mutexattr_init( )**, **pthread_mutexattr_setprioceiling( )**, **pthread_mutexattr_setprotocol( )**, **pthread_mutex_init( )**

P

pthread_mutexattr_getprotocol( )

NAME **pthread_mutexattr_getprotocol( )** – get value of protocol in mutex attributes object (POSIX)

SYNOPSIS

```
int pthread_mutexattr_getprotocol
    (
    pthread_mutexattr_t * pAttr,    /* mutex attributes */
    int *                 pProtocol /* current protocol (out) */
    )
```

DESCRIPTION This function gets the current value of the protocol attribute in a mutex attributes object.

RETURNS On success zero; on failure a non-zero error code.

ERRNOS EINVAL

SEE ALSO **pthreadLib**, **pthread_mutexattr_destroy( )**, **pthread_mutexattr_getprioceiling( )**, **pthread_mutexattr_init( )**, **pthread_mutexattr_setprioceiling( )**, **pthread_mutexattr_setprotocol( )**, **pthread_mutex_init( )**

pthread_mutexattr_init( )

NAME **pthread_mutexattr_init( )** – initialize mutex attributes object (POSIX)

SYNOPSIS

```
int pthread_mutexattr_init
    (
    pthread_mutexattr_t * pAttr /* mutex attributes */
    )
```

DESCRIPTION This routine initializes the mutex attribute object *pAttr* and fills it with default values for the attributes as defined by the POSIX specification.

RETURNS On success zero; on failure a non-zero error code.

ERRNOS EINVAL

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 a non-zero error code.

ERRNOS EINVAL

SEE ALSO **pthreadLib**, **pthread_mutexattr_destroy( )**, **pthread_mutexattr_getprioceiling( )**, **pthread_mutexattr_getprotocol( )**, **pthread_mutexattr_init( )**, **pthread_mutexattr_setprotocol( )**, **pthread_mutex_init( )**

P

pthread_mutexattr_setprotocol( )

NAME **pthread_mutexattr_setprotocol( )** – set protocol attribute in mutex attribute object (POSIX)

SYNOPSIS

```
int pthread_mutexattr_setprotocol
    (
    pthread_mutexattr_t * pAttr,   /* mutex attributes */
    int                   protocol /* new protocol */
    )
```

DESCRIPTION This function selects the locking protocol to be used when a mutex is created using this attributes object. The protocol to be selected is either **PTHREAD_PRIO_INHERIT** or **PTHREAD_PRIO_PROTECT**.

RETURNS On success zero; on failure a non-zero error code.

ERRNOS EINVAL, ENOTSUP

SEE ALSO

pthreadLib, **pthread_mutexattr_destroy( )**, **pthread_mutexattr_getprioceiling( )**, **pthread_mutexattr_getprotocol( )**, **pthread_mutexattr_init( )**, **pthread_mutexattr_setprioceiling( )**, **pthread_mutex_init( )**

pthread_once( )

NAME

pthread_once( ) – dynamic package initialization (POSIX)

SYNOPSIS

```
int pthread_once
    (
    pthread_once_t * onceControl, /* 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, *onceControl*, will result in a call to *initFunc*, but subsequent calls will not. The *onceControl* 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 *onceControl* is the same as if **pthread_once( )** had never been called.

WARNING: If *onceControl* 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

Always returns zero.

ERRNOS

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.

RETURNS Calling thread's ID.

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 a non-zero error code.

ERRNOS **EINVAL**

SEE ALSO **pthreadLib**, **pthread_cancel( )**, **pthread_setcanceltype( )**, **pthread_testcancel( )**

P

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 a non-zero error code.

ERRNOS **EINVAL**

SEE ALSO **pthreadLib**, **pthread_cancel( )**, **pthread_setcancelstate( )**, **pthread_testcancel( )**

pthread_setschedparam( )

NAME **pthread_setschedparam( )** – dynamically set schedparam attribute for a thread (POSIX)

SYNOPSIS

```
int pthread_setschedparam
    (
    pthread_t                    thread, /* thread */
    int                          policy, /* new policy */
    const struct sched_param * pParam   /* new parameters */
    )
```

DESCRIPTION This routine will set the scheduling parameters (*pParam*) and policy (*policy*) for the thread specified by *thread*.

In VxWorks the scheduling policy is global and not set on a per-thread basis; if the selected policy does not match the current global setting then this function will return an error (**EINVAL**).

RETURNS On success zero; on failure a non-zero error code.

ERRNOS **EINVAL**, **ESRCH**

SEE ALSO **pthreadLib**, **schedPxLib**, **pthread_attr_getschedparam( )**, **pthread_attr_getschedpolicy( )**, **pthread_attr_setschedparam( )**, **pthread_attr_setschedpolicy( )**, **pthread_getschedparam( )**, **sched_getparam( )**, **sched_setparam( )**

pthread_setspecific( )

NAME **pthread_setspecific( )** – set thread specific data (POSIX)

SYNOPSIS

```
int pthread_setspecific
    (
    pthread_key_t key,        /* thread specific data key */
    const void *  value       /* new value */
    )
```

DESCRIPTION Sets the value of the thread specific data associated with *key* to *value* for the calling thread.

RETURNS On success zero; on failure a non-zero error code.

ERRNOS **EINVAL**, **ENOMEM**

SEE ALSO **pthreadLib**, **pthread_getspecific( )**, **pthread_key_create( )**, **pthread_key_delete( )**

P

pthread_sigmask( )

NAME

pthread_sigmask( ) – change and/or examine calling thread's signal mask (POSIX)

SYNOPSIS

```
int pthread_sigmask
    (
    int              how,     /* method for changing set */
    const sigset_t * set,     /* new set of signals */
    sigset_t *       oset     /* old set of signals */
    )
```

DESCRIPTION

This routine changes the signal mask for the calling thread as described by the *how* and *set* arguments. If *oset* is not **NULL**, the previous signal mask is stored in the location pointed to by it.

The value of *how* indicates the manner in which the set is changed and consists of one of the following defined in **signal.h**:

SIG_BLOCK
The resulting set is the union of the current set and the signal set pointed to by *set*.

SIG_UNBLOCK
The resulting set is the intersection of the current set and the complement of the signal set pointed to by *set*.

SIG_SETMASK
The resulting set is the signal set pointed to by *oset*.

RETURNS

On success zero; on failure a non-zero error code is returned.

ERRNOS

EINVAL

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

ERRNOS
N/A

SEE ALSO
pthreadLib, **pthread_cancel( )**, **pthread_setcancelstate( )**, **pthread_setcanceltype( )**

P

ptyDevCreate( )

NAME
ptyDevCreate( ) – create a pseudo terminal

SYNOPSIS
```
STATUS ptyDevCreate
    (
    char * name,              /* name of pseudo terminal */
    int    rdBufSize,         /* size of terminal read buffer */
    int    wrtBufSize         /* size of write buffer */
    )
```

DESCRIPTION
This routine creates a master and slave device which can then be opened by the master and slave processes. The master process simulates the "hardware" side of the driver, while the slave process is the application program that normally talks to a tty driver. Data written to the master device can then be read on the slave device, and vice versa.

RETURNS
OK, or **ERROR** if memory is insufficient.

SEE ALSO
ptyDrv

ptyDevRemove( )

NAME **ptyDevRemove( )** – destroy a pseudo terminal

SYNOPSIS

```
STATUS ptyDevRemove
    (
    char * pName              /* name of pseudo terminal to remove */
    )
```

DESCRIPTION This routine removes an existing master and slave device and releases all allocated memory. It will close any open files using either device.

RETURNS **OK**, or **ERROR** if terminal not found

SEE ALSO **ptyDrv**

ptyDrv( )

NAME **ptyDrv( )** – initialize the pseudo-terminal driver

SYNOPSIS

```
STATUS ptyDrv (void)
```

DESCRIPTION This routine initializes the pseudo-terminal driver. It must be called before any other routine in this module.

RETURNS **OK**, or **ERROR** if the master or slave devices cannot be installed.

SEE ALSO **ptyDrv**

ptyShow( )

NAME **ptyShow( )** – show the state of the Pty Buffers

SYNOPSIS

```
void ptyShow (void)
```

SEE ALSO **ptyDrv**

putc( )

NAME **putc( )** – write a character to a stream (ANSI)

SYNOPSIS

```
int putc
    (
    int    c,                 /* character to write */
    FILE * fp                 /* stream to write to */
    )
```

DESCRIPTION

This routine writes a character *c* to a specified stream, at the position indicated by the stream's file position indicator (if defined), and advances the indicator appropriately.

This routine is equivalent to **fputc( )**, except that if it is implemented as a macro, it may evaluate *fp* more than once; thus, the argument should never be an expression with side effects.

INCLUDE FILES **stdio.h**

RETURNS The character written, or EOF if a write error occurs, with the error indicator set for the stream.

SEE ALSO **ansiStdio**, **fputc( )**

putchar( )

NAME

putchar( ) – write a character to the standard output stream (ANSI)

SYNOPSIS

```
int putchar
    (
    int c                    /* character to write */
    )
```

DESCRIPTION

This routine writes a character *c* to the standard output stream, at the position indicated by the stream's file position indicator (if defined), and advances the indicator appropriately.

This routine is equivalent to **putc( )** with a second argument of **stdout**.

INCLUDE FILES

stdio.h

RETURNS

The character written, or EOF if a write error occurs, with the error indicator set for the standard output stream.

SEE ALSO

ansiStdio, **putc( )**, **fputc( )**

putenv( )

NAME

putenv( ) – set an environment variable

SYNOPSIS

```
STATUS putenv
    (
    char * pEnvString        /* string to add to env */
    )
```

DESCRIPTION

This routine sets an environment variable to a value by altering an existing variable or creating a new one. The parameter points to a string of the form "variableName=value". Unlike the UNIX implementation, the string passed as a parameter is copied to a private buffer.

RETURNS

OK, or **ERROR** if space cannot be malloc'd.

SEE ALSO

envLib, **envLibInit( )**, **getenv( )**

puts( )

NAME **puts( )** – write a string to the standard output stream (ANSI)

SYNOPSIS

```
int puts
    (
    char const * s            /* string to write */
    )
```

DESCRIPTION This routine writes to the standard output stream a specified string *s*, minus the terminating null character, and appends a new-line character to the output.

INCLUDE FILES **stdio.h**

RETURNS A non-negative value, or EOF if a write error occurs.

SEE ALSO **ansiStdio**, **fputs( )**

putw( )

P

NAME **putw( )** – write a word (32-bit integer) to a stream

SYNOPSIS

```
int putw
    (
    int    w,                 /* word (32-bit integer) */
    FILE * fp                 /* output stream */
    )
```

DESCRIPTION This routine appends the 32-bit quantity *w* to a specified stream.

This routine is provided for compatibility with earlier VxWorks releases.

INCLUDE FILES **stdio.h**

RETURNS The value written.

SEE ALSO **ansiStdio**

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 Tornado Shell (**windsh**), which has a built-in command of the same name that operates on the Host's I/O system.

RETURNS N/A

SEE ALSO **usrFsLib**, **cd( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

qsort( )

NAME **qsort( )** – sort an array of objects (ANSI)

SYNOPSIS

```
void qsort
    (
    void * bot,                /* initial element in array */
    size_t nmemb,              /* no. of objects in array */
    size_t size,               /* size of array element */
    int (* compar) (const void * , const void * )
                               /* comparison function */
    )
```

DESCRIPTION This routine sorts an array of *nmemb* objects, the initial element of which is pointed to by *bot*. The size of each object is specified by *size*.

The contents of the array are sorted into ascending order according to a comparison function pointed to by *compar*, which is called with two arguments that point to the objects being compared. The function shall return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

If two elements compare as equal, their order in the sorted array is unspecified.

INCLUDE FILES **stdlib.h**

RETURNS N/A

SEE ALSO **ansiStdlib**

r0( )

NAME
r0( ) – return the contents of register **r0** (also **r1** - **r14**, **r1-r15** for SH) (ARM, SH)

SYNOPSIS

```
int r0
    (
    int taskId                /* task ID, 0 means default task */
    )
```

DESCRIPTION
This command extracts the contents of register **r0** from the TCB of a specified task. If *taskId* is omitted or zero, the last task referenced is assumed.

Similar routines are provided for registers (**r1** - **r15**): **r1( )** - **r15( )**.

RETURNS
The contents of register **r0** (or the requested register).

SEE ALSO
dbgArchLib, *VxWorks Programmer's Guide: Debugging*

raise( )

NAME
raise( ) – send a signal to the caller's task

SYNOPSIS

```
int raise
    (
    int signo                 /* signal to send to caller's task */
    )
```

DESCRIPTION
This routine sends the signal *signo* to the task invoking the call.

RETURNS
OK (0), or **ERROR** (-1) if the signal number or task ID is invalid.

ERRNO
EINVAL

SEE ALSO
sigLib

ramDevCreate( )

NAME

ramDevCreate( ) – create a RAM disk device

SYNOPSIS

```
BLK_DEV *ramDevCreate
    (
    char * ramAddr,           /* where it is in memory (0 = malloc) */
    int    bytesPerBlk,       /* number of bytes per block */
    int    blksPerTrack,      /* number of blocks per track */
    int    nBlocks,           /* number of blocks on this device */
    int    blkOffset          /* no. of blks to skip at start of device */
    )
```

DESCRIPTION

This routine creates a RAM disk device.

Memory for the RAM disk can be pre-allocated separately; if so, the *ramAddr* parameter should be the address of the pre-allocated device memory. Or, memory can be automatically allocated with **malloc( )** by setting *ramAddr* to zero.

The *bytesPerBlk* parameter specifies the size of each logical block on the RAM disk. If *bytesPerBlk* is zero, 512 is used.

The *blksPerTrack* parameter specifies the number of blocks on each logical track of the RAM disk. If *blksPerTrack* is zero, the count of blocks per track is set to *nBlocks* (*i.e.*, the disk is defined as having only one track).

The *nBlocks* parameter specifies the size of the disk, in blocks. If *nBlocks* is zero, a default size is used. The default is calculated using a total disk size of either 51,200 bytes or one-half of the size of the largest memory area available, whichever is less. This default disk size is then divided by *bytesPerBlk* to determine the number of blocks.

The *blkOffset* parameter specifies an offset, in blocks, from the start of the device to be used when writing or reading the RAM disk. This offset is added to the block numbers passed by the file system during disk accesses. (VxWorks file systems always use block numbers beginning at zero for the start of a device.) This offset value is typically useful only if a specific address is given for *ramAddr*. Normally, *blkOffset*is 0.

FILE SYSTEMS

Once the device has been created, it must be associated with a name and a file system (dosFs, rt11Fs, or rawFs). This is accomplished using the file system's device initialization routine or make-file-system routine, *e.g.*, **dosFsDevInit( )** or **dosFsMkfs( )**. The **ramDevCreate( )** call returns a pointer to a block device structure (**BLK_DEV**). This structure contains fields that describe the physical properties of a disk device and specify the addresses of routines within the **ramDrv** driver. The **BLK_DEV** structure address must be passed to the desired file system (dosFs, rt11Fs or rawFs) via the file system's device initialization or make-file-system routine. Only then is a name and file system associated with the device, making it available for use.

EXAMPLE

In the following example, a 200-Kbyte RAM disk is created with automatically allocated memory, 512-byte blocks, a single track, and no block offset. The device is then initialized for use with dosFs and assigned the name "DEV1:":

```
BLK_DEV *pBlkDev;
DOS_VOL_DESC *pVolDesc;
pBlkDev = ramDevCreate (0,  512,  400,  400,  0);
pVolDesc = dosFsMkfs ("DEV1:", pBlkDev);
```

The **dosFsMkfs( )** routine calls **dosFsDevInit( )** with default parameters and initializes the file system on the disk by calling **ioctl( )** with the **FIODISKINIT** function.

If the RAM disk memory already contains a disk image created elsewhere, the first argument to **ramDevCreate( )** should be the address in memory, and the formatting parameters -- *bytesPerBlk*, *blksPerTrack*, *nBlocks*, and *blkOffset* -- must be identical to those used when the image was created. For example:

```
pBlkDev = ramDevCreate (0xc0000, 512, 400, 400, 0);
pVolDesc = dosFsDevInit ("DEV1:", pBlkDev, NULL);
```

In this case, **dosFsDevInit( )** must be used instead of **dosFsMkfs( )**, because the file system already exists on the disk and should not be re-initialized. This procedure is useful if a RAM disk is to be created at the same address used in a previous boot of VxWorks. The contents of the RAM disk will then be preserved.

These same procedures apply when creating a RAM disk with rt11Fs using **rt11FsDevInit( )** and **rt11FsMkfs( )**, or creating a RAM disk with rawFs using **rawFsDevInit( )**.

RETURNS

A pointer to a block device structure (**BLK_DEV**) or **NULL** if memory cannot be allocated for the device structure or for the RAM disk.

SEE ALSO

ramDrv, **dosFsMkfs( )**, **dosFsDevInit( )**, **rt11FsDevInit( )**, **rt11FsMkfs( )**, **rawFsDevInit( )**

ramDiskDevCreate( )

NAME

ramDiskDevCreate( ) – initialize a RAM Disk device

SYNOPSIS

```
CBIO_DEV_ID ramDiskDevCreate
    (
    char * pRamAddr,          /* where it is in memory (0 = malloc) */
    int    bytesPerBlk,       /* number of bytes per block */
    int    blksPerTrack,      /* number of blocks per track */
    int    nBlocks,           /* number of blocks on this device */
```

```
int    blkOffset          /* no. of blks to skip at start of device */
)
```

DESCRIPTION

This function creates a compact RAM-Disk device that can be directly utilized by **dosFsLib**, without the intermediate disk cache. It can be used for non-volatile RAM as well as volatile RAM disks.

The RAM size is specified in terms of total number of blocks in the device and the block size in bytes. The minimal block size is 32 bytes. If *pRamAddr* is **NULL**, space will be allocated from the default memory pool.

RETURNS

a CBIO handle that can be directly used by **dosFsDevCreate( )** or **NULL** if the requested amount of RAM is not available.

WARNING: When used with NV-RAM, this module can not eliminate mid-block write interruption, which may cause file system corruption not existent in common disk drives.

SEE ALSO

ramDiskCbio, **dosFsDevCreate( )**.

ramDrv( )

NAME

ramDrv( ) – prepare a RAM disk driver for use (optional)

SYNOPSIS

```
STATUS ramDrv (void)
```

DESCRIPTION

This routine performs no real function, except to provide compatibility with earlier versions of **ramDrv** and to parallel the initialization function found in true disk device drivers. It also is used in **usrConfig.c** to link in the RAM disk driver when building VxWorks. Otherwise, there is no need to call this routine before using the RAM disk driver.

RETURNS

OK, always.

SEE ALSO

ramDrv

rand( )

NAME
rand( ) – generate a pseudo-random integer between 0 and **RAND_MAX** (ANSI)

SYNOPSIS
```
int rand (void)
```

DESCRIPTION
This routine generates a pseudo-random integer between 0 and **RAND_MAX**. The seed value for **rand( )** can be reset with **srand( )**.

INCLUDE FILES
stdlib.h

RETURNS
A pseudo-random integer.

SEE ALSO
ansiStdlib, **srand( )**

rawFsDevInit( )

NAME
rawFsDevInit( ) – associate a block device with raw volume functions

SYNOPSIS
```
RAW_VOL_DESC *rawFsDevInit
    (
    char *    pVolName,       /* volume name to be used with iosDevAdd */
    BLK_DEV * pDevice         /* a pointer to a BLK_DEV or a CBIO_DEV_ID */
    )
```

DESCRIPTION
This routine takes a block device created by a device driver and defines it as a raw file system volume. As a result, when high-level I/O operations, such as **open( )** and **write( )**, are performed on the device, the calls will be routed through **rawFsLib**.

This routine associates *pVolName* with a device and installs it in the VxWorks I/O System's device table. The driver number used when the device is added to the table is that which was assigned to the raw library during **rawFsInit( )**. (The driver number is kept in the global variable **rawFsDrvNum**.)

The pDevice is a **CBIO_DEV_ID** or **BLK_DEV** ptr and contains configuration data describing the device and the addresses of routines which will be called to access device. These routines will not be called until they are required by subsequent I/O operations.

RETURNS
A pointer to the volume descriptor (**RAW_VOL_DESC**), or **NULL** if there is an error.

SEE ALSO
rawFsLib

rawFsInit( )

NAME

rawFsInit( ) – prepare to use the raw volume library

SYNOPSIS

```
STATUS rawFsInit
    (
    int maxFiles              /* max no. of simultaneously open files */
    )
```

DESCRIPTION

This routine initializes the raw volume library. It must be called exactly once, before any other routine in the library. The argument specifies the number of file descriptors that may be open at once. This routine allocates and sets up the necessary memory structures and initializes semaphores.

This routine also installs raw volume library routines in the VxWorks I/O system driver table. The driver number assigned to **rawFsLib** is placed in the global variable **rawFsDrvNum**. This number will later be associated with system file descriptors opened to rawFs devices.

To enable this initialization, define **INCLUDE_RAWFS** in **configAll.h**; **rawFsInit( )** will then be called from the root task, **usrRoot( )**, in **usrConfig.c**.

RETURNS

OK or **ERROR**.

SEE ALSO

rawFsLib

R

rawFsModeChange( )

NAME

rawFsModeChange( ) – modify the mode of a raw device volume

SYNOPSIS

```
void rawFsModeChange
    (
    RAW_VOL_DESC * pVd,       /* pointer to volume descriptor */
    int            newMode    /* O_RDONLY/O_WRONLY/O_RDWR (both) */
    )
```

DESCRIPTION

This routine sets the device's mode to *newMode* by setting the mode field in the device structure. This routine should be called whenever the read and write capabilities are determined, usually after a ready change.

The driver's device initialization routine should initially set the mode to **O_RDWR** (*i.e.*, both **O_RDONLY** and **O_WRONLY**).

RETURNS N/A

SEE ALSO **rawFsLib**, **rawFsReadyChange( )**

rawFsReadyChange( )

NAME **rawFsReadyChange( )** – notify **rawFsLib** of a change in ready status

SYNOPSIS

```
void rawFsReadyChange
    (
    RAW_VOL_DESC * pVd        /* pointer to volume descriptor */
    )
```

DESCRIPTION This routine sets the volume descriptor state to **RAW_VD_READY_CHANGED**. It should be called whenever a driver senses that a device has come on-line or gone off-line, (*e.g.*, a disk has been inserted or removed).

After this routine has been called, the next attempt to use the volume will result in an attempted remount.

RETURNS N/A

SEE ALSO **rawFsLib**

rawFsVolUnmount( )

NAME **rawFsVolUnmount( )** – disable a raw device volume

SYNOPSIS

```
STATUS rawFsVolUnmount
    (
    RAW_VOL_DESC * pVd        /* pointer to volume descriptor */
    )
```

DESCRIPTION This routine is called when I/O operations on a volume are to be discontinued. This is commonly done before changing removable disks. All buffered data for the volume is written to the device (if possible), any open file descriptors are marked as obsolete, and the volume is marked as not mounted.

Because this routine will flush data from memory to the physical device, it should not be used in situations where the disk-change is not recognized until after a new disk has been inserted. In these circumstances, use the ready-change mechanism. (See the manual entry for **rawFsReadyChange( )**.)

This routine may also be called by issuing an **ioctl( )** call using the **FIOUNMOUNT** function code.

RETURNS **OK**, or **ERROR** if the routine cannot access the volume.

SEE ALSO **rawFsLib**, **rawFsReadyChange( )**

rcmd( )

NAME **rcmd( )** – execute a shell command on a remote machine

SYNOPSIS

```
int rcmd
    (
    char * host,            /* host name or inet address */
    int    remotePort,      /* remote port to connect to (rshd) */
    char * localUser,       /* local user name */
    char * remoteUser,      /* remote user name */
    char * cmd,             /* command */
    int *  fd2p             /* if this pointer is non-zero, stderr */
                            /* socket is opened and socket descriptor is */
                            /* filled in */
    )
```

DESCRIPTION This routine uses a remote shell daemon, **rshd**, to execute a command on a remote system. It is analogous to the BSD **rcmd( )** routine.

Internally, this **rcmd( )** implementation uses a **select( )** call to wait for a response from the **rshd** daemon. If **rcmd( )** receives a response within its timeout, **rcmd( )** calls **accept( )** and completes by returning a socket descriptor for the data generated on the remote machine.

The default timeout lets the **rcmd( )** call wait forever. However, you can change the timeout value using the **RSH_STDERR_SETUP_TIMEOUT** parameter associated with the **NETWRS_REMLIB** configuration component.

R

RETURNS A socket descriptor if the remote shell daemon accepts, or **ERROR** if the remote command fails.

ERRNO **S_remLib_RSH_ERROR**, **S_remLib_RSH_STDERR_SETUP_FAILED**

SEE ALSO **remLib**, BSD reference entry for **rcmd( )**

rcvEtherAddrAdd( )

NAME **rcvEtherAddrAdd( )** – add a physical address into the linked list

SYNOPSIS

```
STATUS rcvEtherAddrAdd
    (
    M2_IFINDEX *    pIfIndexEntry, /* the avl node */
    unsigned char * pEnetAddr      /* the addr to be added */
    )
```

DESCRIPTION This function is a helper function for **rcvEtherAddrGet( )**. It is called to add a single physical address into the linked list of addresses maintained by the AVL node.

RETURNS **OK**, if successful; **ERROR**, otherwise.

SEE ALSO **m2IfLib**

rcvEtherAddrGet( )

NAME **rcvEtherAddrGet( )** – populate the rcvAddr fields for the ifRcvAddressTable

SYNOPSIS

```
STATUS rcvEtherAddrGet
    (
    struct ifnet * pIfNet,       /* pointer to the interface's ifnet */
    M2_IFINDEX *   pIfIndexEntry /* avl node */
    )
```

DESCRIPTION This function needs to be called to add all physical addresses for which an interface may receive or send packets. This includes unicast and multicast addresses. The address is inserted into the linked list maintained in the AVL node corresponding to the interface.

Given the ifnet struct and the AVL node corresponding to the interface, this function goes through all the physical addresses associated with this interface and adds them into the linked list.

RETURNS **OK**, if successful; **ERROR**, otherwise.

SEE ALSO **m2IfLib**

rdCtl( )

NAME **rdCtl( )** – implement the ICMP router discovery control function

SYNOPSIS

```
STATUS rdCtl
    (
    char * ifName,
    int    cmd,
    void*  value              /* my be an int (set-cmds) or an int* */
                              /* (get-cmds) */
    )
```

DESCRIPTION This routine allows a user to get and set router discovery parameters, and control the mode of operation.

OPTIONS The following section discuss the various flags that may be passed to **rdCtl( )**.

SET_MODE
Set debug mode or exit router discovery

This flag does not require an *interface* to be specified it is best to specify **NULL**.

This flag is used in conjunction with the following values:

MODE_DEBUG_ON
Turn debugging messages on.

```
rdCtl (NULL, SET_MODE, MODE_DEBUG_ON);
```

MODE_DEBUG_OFF
Turn debugging messages off.

```
rdCtl (NULL, SET_MODE, MODE_DEBUG_OFF);
```

MODE_STOP
Exit from router discovery.

```
rdCtl (NULL, SET_MODE, MODE_STOP);
```

R

SET_MIN_ADVERT_INT
Set minimum advertisement interval in seconds

Specify the minimum time between advertisements in seconds. The minimum value allowed is 4 seconds, the maximum is 1800.

```
rdCtl (NULL, SET_MIN_ADVERT_INT, <seconds>);
```

SET_MAX_ADVERT_INT
Set maximum advertisement interval in seconds

Specify the maximum time between advertisements in seconds. The minimum value allowed is 4 seconds, the maximum is 1800.

```
rdCtl (NULL, SET_MAX_ADVERT_INT, <seconds>);
```

SET_FLAG
Set whether advertisements are sent on an interface.

If this flag is 1 then advertisements are sent on this interface. If it is 0 then they are not.

```
rdCtl (<interface>, SET_FLAG, <0 or 1>);
```

SET_ADVERT_ADDRESS
Set the IP address to which advertisements are sent.

Set the multicast IP address to which advertisements are sent.

```
rdCtl (<interface>, SET_ADVERT_ADDRESS, <multicast address>);
```

SET_ADVERT_LIFETIME
Set the lifetime for advertisements in seconds.

Set the lifetime in seconds to be contained in each advertisement.

```
rdCtl (<interface>, SET_ADVERT_LIFETIME, seconds);
```

SET_ADVERT_PREF
Set the preference level contained in advertisements.

```
rdCtl (<interface>, SET_ADVERT_PREF, value);
```

GET_MIN_ADVERT_INT
Get the minimum advertisement interval.

```
rdCtl (NULL, GET_MIN_ADVERT_INT, &value);
```

GET_MAX_ADVERT_INT
Get the maximum advertisement interval.

```
rdCtl (NULL, GET_MAX_ADVERT_INT, &value);
```

GET_FLAG
Get the flag on an interface.

```
rdCtl (<interface>, GET_FLAG, &value);
```

GET_ADVERT_ADDRESS
Get the advertisement address for an interface.

```
rdCtl (<interface>, GET_ADVERT_ADDRESS, &value);
```

GET_ADVERT_LIFETIME
Get the advertisement lifetime.

```
rdCtl (<interface>, GET_ADVERT_LIFETIME, &value);
```

GET_ADVERT_PREF
Get the advertisement preference.

```
rdCtl (<interface>, GET_ADVERT_PREF, value);
```

RETURNS **OK** on success, **ERROR** on failure

SEE ALSO **rdiscLib**

---

rdisc( )

NAME **rdisc( )** – implement the ICMP router discovery function

SYNOPSIS

```
void rdisc ()
```

DESCRIPTION This routine is the entry point for the router discovery function. It allocates and initializes resources, listens for solicitation messages on the **ALL_ROUTERS** (224.0.0.1) multicast address and processes the messages.

This routine usually runs until explicitly killed by a system operator, but can also be terminated cleanly (see **rdCtl( )** routine).

RETURNS N/A

SEE ALSO **rdiscLib**

rdiscIfReset( )

NAME **rdiscIfReset( )** – check for new or removed interfaces for router discovery

SYNOPSIS

```
STATUS rdiscIfReset ()
```

DESCRIPTION This routine *must* be called any time an interface is added to or removed from the system so that the router discovery code can deal with this case. Failure to do so will cause the sending of packets on missing interfaces to fail as well as no transmission of packets on new interfaces.

SEE ALSO **rdiscLib**

rdiscInit( )

NAME **rdiscInit( )** – initialize the ICMP router discovery function

SYNOPSIS

```
STATUS rdiscInit ()
```

DESCRIPTION This routine allocates resources for the router discovery function. Since it called in the **rdisc( )** routine, it should not be called subsequently.

RETURNS **OK** on successful initialization, **ERROR** otherwise

SEE ALSO **rdiscLib**

rdiscLibInit( )

NAME **rdiscLibInit( )** – initialize router discovery

SYNOPSIS

```
void rdiscLibInit
    (
    int priority,           /* Priority of router discovery task. */
    int options,            /* Options to taskSpawn(1) for router */
                            /* discovery task. */
    int stackSize           /* Stack size for router discovery task. */
    )
```

DESCRIPTION This routine links the ICMP Router Discovery facility into the VxWorks system. The arguments are the task's priority, options and stack size.

RETURNS N/A

SEE ALSO **rdiscLib**

rdiscTimerEvent( )

NAME **rdiscTimerEvent( )** – called after watchdog timeout

SYNOPSIS

```
void rdiscTimerEventRestart
    (
    int stackNum
    )
```

DESCRIPTION This routine is called when a new advertisement is to be sent.

RETURNS N/A

SEE ALSO **rdiscLib**

read( )

NAME

read( ) – read bytes from a file or device

SYNOPSIS

```
int read
    (
    int    fd,              /* file descriptor from which to read */
    char * buffer,          /* pointer to buffer to receive bytes */
    size_t maxbytes         /* max no. of bytes to read into buffer */
    )
```

DESCRIPTION

This routine reads a number of bytes (less than or equal to *maxbytes*) from a specified file descriptor and places them in *buffer*. It calls the device driver to do the work.

RETURNS

The number of bytes read (between 1 and *maxbytes*, 0 if end of file), or **ERROR** if the file descriptor does not exist, the driver does not have a read routines, or the driver returns **ERROR**. If the driver does not have a read routine, **errno** is set to **ENOTSUP**.

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.

SEE ALSO
dirLib, **opendir( )**, **closedir( )**, **rewinddir( )**, **ls( )**

realloc( )

NAME
realloc( ) – reallocate a block of memory (ANSI)

SYNOPSIS

```
void *realloc
    (
    void * pBlock,            /* block to reallocate */
    size_t newSize            /* new block size */
    )
```

DESCRIPTION
This routine changes the size of a specified block of memory and returns a pointer to the new block of memory. The contents that fit inside the new size (or old size if smaller) remain unchanged. The memory alignment of the new block is not guaranteed to be the same as the original block.

RETURNS
A pointer to the new block of memory, or **NULL** if the call fails.

SEE ALSO
memLib, *American National Standard for Information Systems -Programming Language - C, ANSI X3.159-1989: General Utilities (**stdlib.h**)*

R

reboot( )

NAME

reboot( ) – reset network devices and transfer control to boot ROMs

SYNOPSIS

```
void reboot
    (
    int startType              /* how the boot ROMS will reboot */
    )
```

DESCRIPTION

This routine returns control to the boot ROMs after calling a series of preliminary shutdown routines that have been added via **rebootHookAdd( )**, including routines to reset all network devices. After calling the shutdown routines, interrupts are locked, all caches are cleared, and control is transferred to the boot ROMs.

The bit values for *startType* are defined in **sysLib.h**:

BOOT_NORMAL (0x00)
: causes the system to go through the countdown sequence and try to reboot VxWorks automatically. Memory is not cleared.

BOOT_NO_AUTOBOOT (0x01)
: causes the system to display the VxWorks boot prompt and wait for user input to the boot ROM monitor. Memory is not cleared.

BOOT_CLEAR (0x02)
: the same as **BOOT_NORMAL**, except that memory is cleared.

BOOT_QUICK_AUTOBOOT (0x04)
: the same as **BOOT_NORMAL**, except the countdown is shorter.

RETURNS

N/A

SEE ALSO

rebootLib, **sysToMonitor( )**, **rebootHookAdd( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

rebootHookAdd( )

NAME **rebootHookAdd( )** – add a routine to be called at reboot

SYNOPSIS

```
STATUS rebootHookAdd
    (
    FUNCPTR rebootHook        /* routine to be called at reboot */
    )
```

DESCRIPTION This routine adds the specified routine to a list of routines to be called when VxWorks is rebooted. The specified routine should be declared as follows:

```
    void rebootHook
        (
        int startType   /* startType is passed to all hooks */
        )
```

RETURNS **OK**, or **ERROR** if memory is insufficient.

SEE ALSO **rebootLib**, **reboot( )**

recv( )

NAME **recv( )** – receive data from a socket

SYNOPSIS

```
int recv
    (
    int    s,                 /* socket to receive data from */
    char * buf,               /* buffer to write data to */
    int    bufLen,            /* length of buffer */
    int    flags              /* flags to underlying protocols */
    )
```

DESCRIPTION This routine receives data from a connection-based (stream) socket.

The maximum length of *buf* is subject to the limits on TCP buffer size; see the discussion of **SO_RCVBUF** in the **setsockopt( )** manual entry.

You may OR the following values into the *flags* parameter with this operation:

R

MSG_OOB (0x1)
Out-of-band data.

MSG_PEEK (0x2)
Return data without removing it from socket.

RETURNS The number of bytes received, or **ERROR** if the call fails.

SEE ALSO **sockLib**, **setsockopt( )**

recvfrom( )

NAME **recvfrom( )** – receive a message from a socket

SYNOPSIS

```
int recvfrom
    (
    int               s,       /* socket to receive from */
    char *            buf,     /* pointer to data buffer */
    int               bufLen,  /* length of buffer */
    int               flags,   /* flags to underlying protocols */
    struct sockaddr * from,    /* where to copy sender's addr */
    int *             pFromLen /* value/result length of from */
    )
```

DESCRIPTION This routine receives a message from a datagram socket regardless of whether it is connected. If *from* is non-zero, the address of the sender's socket is copied to it. The value-result parameter *pFromLen* should be initialized to the size of the *from* buffer. On return, *pFromLen* contains the actual size of the address stored in *from*.

The maximum length of *buf* is subject to the limits on UDP buffer size; see the discussion of **SO_RCVBUF** in the **setsockopt( )** manual entry.

You may OR the following values into the *flags* parameter with this operation:

MSG_OOB (0x1)
Out-of-band data.

MSG_PEEK (0x2)
Return data without removing it from socket.

RETURNS The number of number of bytes received, or **ERROR** if the call fails.

SEE ALSO **sockLib**, **setsockopt( )**

recvmsg( )

NAME

recvmsg( ) – receive a message from a socket

SYNOPSIS

```
int recvmsg
    (
    int              sd,       /* socket to receive from */
    struct msghdr * mp,        /* scatter-gather message header */
    int              flags     /* flags to underlying protocols */
    )
```

DESCRIPTION

This routine receives a message from a datagram socket. It may be used in place of **recvfrom( )** to decrease the overhead of breaking down the message-header structure **msghdr** for each message.

For BSD 4.4 sockets a copy of the *mp*>msg_iov array will be made. This requires a cluster from the network stack system pool of size *mp*>msg_iovlen * sizeof (struct iovec) or 8 bytes.

RETURNS

The number of bytes received, or **ERROR** if the call fails.

SEE ALSO

sockLib

reld( )

NAME

reld( ) – reload an object module

SYNOPSIS

```
MODULE_ID reld
    (
    void * nameOrId,          /* name or ID of the object module file */
    int    options            /* options used for unloading */
    )
```

DESCRIPTION

This routine unloads a specified object module from the system, and then calls **loadModule( )** to load a new copy of the same name.

If the file was originally loaded using a complete pathname, then **reld( )** will use the complete name to locate the file. If the file was originally loaded using a partial pathname, then the current working directory must be changed to the working directory in use at the time of the original load.

Valid values for the options parameter are the same as those allowed for the function **unld( )**.

This routine is a **shell command**. That is, it is designed to be used only in the shell, and not in code running on the target. In future releases, calling **reld( )** directly from code may not be supported.

RETURNS A module ID (type **MODULE_ID**), or **NULL**.

SEE ALSO **unldLib**, **unld( )**

remCurIdGet( )

NAME **remCurIdGet( )** – get the current user name and password

SYNOPSIS

```
void remCurIdGet
    (
    char * user,              /* where to return current user name */
    char * passwd             /* where to return current password */
    )
```

DESCRIPTION This routine gets the user name and password currently used for remote host access privileges and copies them to *user* and *passwd*. Either parameter can be initialized to **NULL**, and the corresponding item will not be passed.

RETURNS N/A

SEE ALSO **remLib**, **iam( )**, **whoami( )**

remCurIdSet( )

NAME **remCurIdSet( )** – set the remote user name and password

SYNOPSIS

```
STATUS remCurIdSet
    (
    char * newUser,           /* user name to use on remote */
    char * newPasswd          /* password to use on remote (NULL = none) */
    )
```

DESCRIPTION

This routine specifies the user name that will have access privileges on the remote machine. The user name must exist in the remote machine's **/etc/passwd**, and if it has been assigned a password, the password must be specified in *newPasswd*.

Either parameter can be **NULL**, and the corresponding item will not be set.

The maximum length of the user name and the password is **MAX_IDENTITY_LEN**(defined in **remLib.h**).

NOTE: A more convenient version of this routine is **iam( )**, which is intended to be used from the shell.

RETURNS

OK, or **ERROR** if the name or password is too long.

SEE ALSO

remLib, **iam( )**, **whoami( )**

---

remove( )

NAME

remove( ) – remove a file (ANSI)

SYNOPSIS

```
STATUS remove
    (
    const char * name          /* name of the file to remove */
    )
```

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**.

SEE ALSO

ioLib, *American National Standard for Information Systems -Programming Language - C, ANSI X3.159-1989: Input/Output (**stdio.h**)*

R

rename( )

NAME

rename( ) – change the name of a file

SYNOPSIS

```
int rename
    (
    const char * oldname,     /* name of file to rename */
    const char * newname      /* name with which to rename file */
    )
```

DESCRIPTION

This routine changes the name of a file from *oldfile* to *newfile*.

NOTE: Only certain devices support **rename( )**. To confirm that your device supports it, consult the respective **xxDrv** or xxFs listings to verify that ioctl **FIORENAME** exists. For example, dosFs and rt11Fs support **rename( )**, but **netDrv** and **nfsDrv** do not.

RETURNS

OK, or **ERROR** if the file could not be opened or renamed.

SEE ALSO

ioLib

repeat( )

NAME

repeat( ) – spawn a task to call a function repeatedly

SYNOPSIS

```
int repeat
    (
    int     n,                /* no. of times to call func (0=forever) */
    FUNCPTR func,             /* function to call repeatedly */
    int     arg1,             /* first of eight args to pass to func */
    int     arg2,
    int     arg3,
    int     arg4,
    int     arg5,
    int     arg6,
    int     arg7,
    int     arg8
    )
```

DESCRIPTION

This command spawns a task that calls a specified function *n* times, with up to eight of its arguments. If *n* is 0, the routine is called endlessly, or until the spawned task is deleted.

NOTE: The task is spawned using **sp( )**. See the description of **sp( )** for details about priority, options, stack size, and task ID.

RETURNS

A task ID, or **ERROR** if the task cannot be spawned.

SEE ALSO

usrLib, **repeatRun( )**, **sp( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

---

repeatRun( )

NAME

repeatRun( ) – call a function repeatedly

SYNOPSIS

```
void repeatRun
    (
    int      n,                /* no. of times to call func (0=forever) */
    FUNCPTR func,              /* function to call repeatedly */
    int      arg1,             /* first of eight args to pass to func */
    int      arg2,
    int      arg3,
    int      arg4,
    int      arg5,
    int      arg6,
    int      arg7,
    int      arg8
    )
```

DESCRIPTION

This command calls a specified function *n* times, with up to eight of its arguments. If *n* is 0, the routine is called endlessly.

Normally, this routine is called only by **repeat( )**, which spawns it as a task.

RETURNS

N/A

SEE ALSO

usrLib, **repeat( )**, *VxWorks Programmer's Guide: Target Shell*

R

resolvDNComp( )

NAME **resolvDNComp( )** – compress a DNS name in a DNS packet

SYNOPSIS

```
int resolvDNComp
    (
    const u_char * exp_dn,   /* ptr to the expanded domain name */
    u_char *       comp_dn,  /* ptr to where to output the compressed name */
    int            length,   /* length of the buffer pointed by comp_dn */
    u_char * *     dnptrs,   /* ptr to a ptr list of compressed names */
    u_char * *     lastdnptr /* ptr to the last entry pointed by dnptrs */
    )
```

DESCRIPTION This routine takes the expanded domain name referenced in the *exp_dn* parameter, compresses it, and stores the compressed name in the location pointed to by the *comp_dn* parameter. The *length* parameter passes in the length of the buffer starting at *comp_dn*. The *dnptrs* parameter is a pointer to a list of pointers to previously compressed names. The *lastdnptr* parameter points to the last entry in the *dnptrs* array.

RETURNS The size of the compressed name, or **ERROR**.

SEE ALSO **resolvLib**, **resolvGetHostByName( )**, **resolvGetHostByAddr( )**, **resolvDNExpand( )**, **resolvInit( )**, **resolvSend( )**, **resolvParamsSet( )**, **resolvParamsGet( )**, **resolvMkQuery( )**, **resolvQuery( )**

resolvDNExpand( )

NAME **resolvDNExpand( )** – expand a DNS compressed name from a DNS packet

SYNOPSIS

```
int resolvDNExpand
    (
    const u_char * msg,       /* ptr to the start of the DNS packet */
    const u_char * eomorig,   /* ptr to the last location +1 of the DNS */
                              /* packet */
    const u_char * comp_dn,   /* ptr to the compressed domain name */
    u_char *       exp_dn,    /* ptr to where the expanded DN is output */
    int            length     /* length of the buffer pointed by expd_dn */
    )
```

DESCRIPTION This functions expands a compressed DNS name from a DNS packet. The *msg* parameter points to that start of the DNS packet. The *eomorig* parameter points to the last location of the DNS packet plus 1. The *comp_dn* parameter points to the compress domain name, and *exp_dn* parameter expects a pointer to a buffer. Upon function completion, this buffer contains the expanded domain name. Use the *length* parameter to pass in the size of the buffer referenced by the *exp_dn* parameter.

RETURNS The length of the expanded domain name, or **ERROR** on failure.

SEE ALSO **resolvLib**, **resolvGetHostByName( )**, **resolvGetHostByAddr( )**, **resolvInit( )**, **resolvDNComp( )**, **resolvSend( )**, **resolvParamsSet( )**, **resolvParamsGet( )**, **resolvMkQuery( )**, **resolvQuery( )**

---

resolvGetHostByAddr( )

NAME **resolvGetHostByAddr( )** – query the DNS server for the host name of an IP address

SYNOPSIS

```
struct hostent * resolvGetHostByAddr
    (
    const char * pInetAddr,
    char *       pHostBuf,
    int          bufLen
    )
```

DESCRIPTION This function returns a **hostent** structure, which is defined as follows:

```
struct   hostent
    {
    char *   h_name;            /* official name of host */
    char **  h_aliases;         /* alias list */
    int      h_addrtype;        /* address type */
    int      h_length;          /* length of address */
    char **  h_addr_list;       /* list of addresses from name server */
    unsigned int h_ttl;         /* Time to Live in Seconds for this entry */
    }
```

The **h_aliases** and **h_addr_list** vectors are **NULL**-terminated. For a locally resolved entry **h_ttl** is always 60 (an externally resolved entry may also have a TTL of 60 depending on its age but it is usually much higher).

The *pinetAddr* parameter passes in the IP address (in network byte order) for the host whose name you want to discover. The *pBuf* and *bufLen* parameters specify the location

R

and size (512 bytes or more) of the buffer that is to receive the **hostent** structure. **resolvGetHostByAddr( )** returns host addresses are returned in network byte order.

RETURNS

A pointer to a **hostent** structure if the host is found, or **NULL** if the parameters are invalid, host is not found, or the buffer is too small.

ERRNO

S_resolvLib_INVALID_PARAMETER
S_resolvLib_BUFFER_2_SMALL
S_resolvLib_TRY_AGAIN
S_resolvLib_HOST_NOT_FOUND
S_resolvLib_NO_DATA
S_resolvLib_NO_RECOVERY

SEE ALSO

resolvLib, **resolvGetHostByName( )**, **resolvInit( )**, **resolvDNExpand( )**, **resolvDNComp( )**, **resolvSend( )**, **resolvParamsSet( )**, **resolvParamsGet( )**, **resolvMkQuery( )**, **resolvQuery( )**

resolvGetHostByName( )

NAME

resolvGetHostByName( ) – query the DNS server for the IP address of a host

SYNOPSIS

```
struct hostent * resolvGetHostByName
    (
    char * pHostName, /* ptr to the name of the host */
    char * pHostBuf,  /* ptr to the buffer used by hostent structure */
    int    bufLen     /* length of the buffer */
    )
```

DESCRIPTION

This function returns a **hostent** structure. This structure is defined as follows:

```
    struct   hostent
    {
    char *   h_name;          /* official name of host */
    char **  h_aliases;       /* alias list */
    int      h_addrtype;      /* address type */
    int      h_length;        /* length of address */
    char **  h_addr_list;     /* list of addresses from name server */
    unsigned int h_ttl;       /* Time to Live in Seconds for this entry */
    }
```

The **h_aliases** and **h_addr_list** vectors are **NULL**-terminated. For a locally resolved entry **h_ttl** is always 60 (an externally resolved entry may also have a TTL of 60 depending on its age but it is usually much higher).

Specify the host you want to query in *pHostname*. Use *pBuf* and *bufLen* to specify the location and size of a buffer to receive the **hostent** structure and its associated contents. Host addresses are returned in network byte order. Given the information this routine retrieves, the *pBuf* buffer should be 512 bytes or larger.

RETURNS A pointer to a **hostent** structure if the host is found, or **NULL** if the parameters are invalid, the host is not found, or the buffer is too small.

ERRNO
S_resolvLib_INVALID_PARAMETER
S_resolvLib_BUFFER_2_SMALL
S_resolvLib_TRY_AGAIN
S_resolvLib_HOST_NOT_FOUND
S_resolvLib_NO_DATA
S_resolvLib_NO_RECOVERY

SEE ALSO **resolvLib**, **resolvInit( )**, **resolvGetHostByAddr( )**, **resolvDNExpand( )**, **resolvDNComp( )**, **resolvSend( )**, **resolvParamsSet( )**, **resolvParamsGet( )**, **resolvMkQuery( )**, **resolvQuery( )**

---

resolvInit( )

NAME **resolvInit( )** – initialize the resolver library

SYNOPSIS

```
STATUS resolvInit
    (
    char *   pNameServer,         /* pointer to Name server IP address */
    char *   pDefaultDomainName,  /* default domain name */
    FUNCPTR pdnsDebugRtn          /* function ptr to debug routine */
    )
```

DESCRIPTION This function initializes the resolver. *pNameServer* is a single IP address for a name server in dotted decimal notation. *pDefaultDomainName* is the default domain name to be appended to names without a dot. The function pointer *pdnsDebugRtn* is set to the resolver debug function. Additional name servers can be configured using the function **resolvParamsSet( )**.

RETURNS **OK** or **ERROR**.

SEE ALSO **resolvLib**, **resolvGetHostByName( )**, **resolvGetHostByAddr( )**, **resolvDNExpand( )**, **resolvDNComp( )**, **resolvSend( )**, **resolvParamsSet( )**, **resolvParamsGet( )**, **resolvQuery( )**

resolvMkQuery( )

NAME

resolvMkQuery( ) – create all types of DNS queries

SYNOPSIS

```
int resolvMkQuery
    (
    int          op,          /* set to desire query QUERY or IQUERY */
    const char * dname,       /* domain name to be use in the query */
    int          class,       /* query class for IP is C_IN */
    int          type,        /* type is T_A, T_PTR, ... */
    const char * data,        /* resource Record (RR) data */
    int          datalen,     /* length of the RR */
    const char * newrr_in,    /* not used always set to NULL */
    char *       buf,         /* out of the constructed query */
    int          buflen       /* length of the buffer for the query */
    )
```

DESCRIPTION

This routine uses the input parameters to create a domain name query. You can set the *op* parameter to QUERY or IQUERY. Specify the domain name in *dname*, the class in *class*, the query type in *type*. Valid values for type include **T_A**, **T_PTR**, and so on. Use *data* to add Resource Record data to the query. Use *datalen* to pass in the length of the data buffer. Set *newrr_in* to **NULL**. This parameter is reserved for future use. The *buf* parameter expects a pointer to the output buffer for the constructed query. Use *buflen* to pass in the length of the buffer referenced in *buf*.

RETURNS

The length of the constructed query or **ERROR**.

SEE ALSO

resolvLib, **resolvGetHostByName( )**, **resolvGetHostByAddr( )**, **resolvDNExpand( )**, **resolvDNComp( )**, **resolvSend( )**, **resolvParamsSet( )**, **resolvParamsGet( )**, **resolvInit( )**, **resolvQuery( )**

resolvParamsGet( )

NAME

resolvParamsGet( ) – get the parameters which control the resolver library

SYNOPSIS

```
void resolvParamsGet
    (
    RESOLV_PARAMS_S * pResolvParams /* ptr to resolver parameter struct */
    )
```

DESCRIPTION

This routine copies the resolver parameters to the **RESOLV_PARAMS_S** structure referenced in the *pResolvParms* parameter. The **RESOLV_PARAMS_S** structure is defined in **resolvLib.h** as follows:

```
typedef struct
   {
   char   queryOrder;
   char   domainName [MAXDNAME];
   char   nameServersAddr [MAXNS][MAXIPADDRLEN];
   } RESOLV_PARAMS_S;
```

Typically, you call this function just before calling **resolvParamsSet( )**. The **resolvParamsGet( )** call populates the **RESOLV_PARAMS_S** structure. You can then modify the default values just before calling **resolvParamsSet( )**.

RETURNS

N/A

SEE ALSO

resolvLib, **resolvGetHostByName( )**, **resolvGetHostByAddr( )**, **resolvDNExpand( )**, **resolvDNComp( )**, **resolvSend( )**, **resolvParamsSet( )**, **resolvInit( )**, **resolvMkQuery( )**, **resolvQuery( )**

resolvParamsSet( )

NAME

resolvParamsSet( ) – set the parameters which control the resolver library

SYNOPSIS

```
STATUS resolvParamsSet
    (
    RESOLV_PARAMS_S * pResolvParams /* ptr to resolver parameter struct */
    )
```

DESCRIPTION

This routine sets the resolver parameters. *pResolvParams* passes in a pointer to a **RESOLV_PARAMS_S** structure, which is defined as follows:

```
typedef struct
   {
   char   queryOrder;
   char   domainName [MAXDNAME];
   char   nameServersAddr [MAXNS][MAXIPADDRLEN];
   } RESOLV_PARAMS_S;
```

Use the members of this structure to specify the settings you want to apply to the resolver. It is important to remember that multiple tasks can use the resolver library and that the settings specified in this **RESOLV_PARAMS_S** structure affect all queries from all tasks. In

R

addition, you should set resolver parameters at initialization and not while queries could be in progress. Otherwise, the results of the query are unpredictable.

Before calling **resolvParamsSet( )**, you should first call **resolvParamsGet( )** to populate a **RESOLV_PARAMS_S** structure with the current settings. Then you change the values of the members that interest you.

Valid values for the **queryOrder** member of **RESOLV_PARAMS_S** structure are defined in **resolvLib.h**. Set the **domainName** member to the domain to which this resolver belongs. Set the **nameServersAddr** member to the IP addresses of the DNS server that the resolver can query. You must specify the IP addresses in standard dotted decimal notation. This function tries to validate the values in the **queryOrder** and **nameServerAddr** members. This function does not try to validate the domain name.

RETURNS

OK if the parameters are valid, **ERROR** otherwise.

SEE ALSO

resolvLib, **resolvGetHostByName( )**, **resolvGetHostByAddr( )**, **resolvDNExpand( )**, **resolvDNComp( )**, **resolvSend( )**, **resolvInit( )**, **resolvParamsGet( )**, **resolvMkQuery( )**, **resolvQuery( )**

resolvQuery( )

NAME

resolvQuery( ) – construct a query, send it, wait for a response

SYNOPSIS

```
int resolvQuery
    (
    char *   name,           /* domain name */
    int      class,          /* query class for IP is C_IN */
    int      type,           /* type is T_A, T_PTR, ... */
    u_char * answer,         /* buffer to put answer */
    int      anslen          /* length of answer buffer */
    )
```

DESCRIPTION

This routine constructs a query for the domain specified in the *name* parameter. The *class* parameter specifies the class of the query. The *type* parameter specifies the type of query. The routine then sends the query to the DNS server. When the server responds, the response is validated and copied to the buffer you supplied in the *answer* parameter. Use the *anslen* parameter to pass in the size of the buffer referenced in *answer*.

RETURNS

The length of the response or **ERROR**.

ERRNO
S_resolvLib_TRY_AGAIN
S_resolvLib_HOST_NOT_FOUND
S_resolvLib_NO_DATA
S_resolvLib_NO_RECOVERY

SEE ALSO
resolvLib, **resolvGetHostByName( )**, **resolvGetHostByAddr( )**, **resolvDNExpand( )**, **resolvDNComp( )**, **resolvInit( )**, **resolvParamsSet( )**, **resolvParamsGet( )**, **resolvMkQuery( )**

resolvSend( )

NAME
resolvSend( ) – send a pre-formatted query and return the answer

SYNOPSIS

```
int resolvSend
    (
    const char * buf,         /* pre-formatted query */
    int          buflen,      /* length of query */
    char *       answer,      /* buffer for answer */
    int          anslen       /* length of answer */
    )
```

DESCRIPTION
This routine takes a pre-formatted DNS query and sends it to the domain server. Use *buf* to pass in a pointer to the query. Use *buflen* to pass in the size of the buffer referenced in *buf*. The *answer* parameter expects a pointer to a buffer into which this routine can write the answer retrieved from the server. Use *anslen* to pass in the size of the buffer you have provided in *anslen*.

RETURNS
The length of the response or **ERROR**.

ERRNO
S_resolvLib_TRY_AGAIN
ECONNREFUSE
ETIMEDOU

SEE ALSO
resolvLib, **resolvGetHostByName( )**, **resolvGetHostByAddr( )**, **resolvDNExpand( )**, **resolvDNComp( )**, **resolvInit( )**, **resolvParamsSet( )**, **resolvParamsGet( )**, **resolvMkQuery( )**, **resolvQuery( )**

R

rewind( )

NAME

rewind( ) – set the file position indicator to the beginning of a file (ANSI)

SYNOPSIS

```
void rewind
    (
    FILE * fp                /* stream */
    )
```

DESCRIPTION

This routine sets the file position indicator for a specified stream to the beginning of the file.

It is equivalent to:

```
(void) fseek (fp, 0L, SEEK_SET);
```

except that the error indicator for the stream is cleared.

INCLUDE FILES

stdio.h

RETURNS

N/A

SEE ALSO

ansiStdio, **fseek( )**, **ftell( )**

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

SEE ALSO **dirLib**, **opendir( )**, **readdir( )**, **closedir( )**

rindex( )

NAME **rindex( )** – find the last occurrence of a character in a string

SYNOPSIS

```
char *rindex
    (
    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.

SEE ALSO **bLib**

ripAddrsXtract( )

NAME **ripAddrsXtract( )** – extract socket address pointers from the route message

SYNOPSIS

```
void ripAddrsXtract
    (
    ROUTE_INFO *        pRtInfo,    /* Route information message */
    struct sockaddr * * pDstAddr,   /* Where to store the Destination */
                                    /* addr pointer */
    struct sockaddr * * pNetmask,   /* Where to store the netmask pointer*/
    struct sockaddr * * pGateway,   /* Where to store the Gateway addr */
                                    /* pointer */
    struct sockaddr * * pOldGateway /* Where to store the Old gateway */
                                    /* addr (if any) pointer */
    )
```

R

DESCRIPTION This routine extracts the socket addresses from the route message in *pRtInfo* and uses the other parameters to return pointers to the extracted messages.

pRtInfo
Passes in a pointer to a route information message.

pDstAddr
Returns a pointer to the destination address.

pNetmask
Returns a pointer to the netmask.

pGateway
Returns a pointer to the gateway address.

pOldGateway
Returns a pointer to the OLD gateway address if it exists.

If the route message doesn't specify an address, the corresponding address pointer is set to **NULL**

RETURNS N/A

ERRNO N/A

SEE ALSO **ripLib**

ripAuthHook( )

NAME **ripAuthHook( )** – sample authentication hook

SYNOPSIS

```
STATUS ripAuthHook
    (
    char *    pKey,         /* rip2IfConfAuthKey entry from MIB-II family */
    RIP_PKT * pRip          /* received RIP message */
    )
```

DESCRIPTION This hook demonstrates one possible authentication mechanism. It rejects all RIP-2 messages that used simple password authentication since they did not match the key contained in the MIB variable. All other RIP-2 messages are also rejected since no other authentication type is supported and all RIP-1 messages are also rejected, as recommended by the RFC specification. This behavior is the same as if no hook were installed.

RETURNS **OK**, if message is acceptable; or **ERROR** otherwise.

SEE ALSO **ripLib**

ripAuthHookAdd( )

NAME **ripAuthHookAdd( )** – add an authentication hook to a RIP interface

SYNOPSIS

```
STATUS ripAuthHookAdd
    (
    char*   pIpAddr,          /* IP address in dotted decimal notation */
    FUNCPTR pAuthHook         /* routine to handle message authentication */
    )
```

DESCRIPTION This routine installs a hook routine to validate incoming RIP messages for a registered interface given by *pIpAddr*. (Interfaces created or changed after a RIP session has started may be installed/updated with the **ripIfSearch( )** and **ripIfReset( )** routines). The hook is only called if an SNMP agent enables authentication for the corresponding interface. It uses the following prototype:

```
STATUS ripAuthHookRtn (char *pKey, RIP_PKT *pRip);
```

The first argument contains the authentication key for the message stored in the rip2IfConfAuthKey MIB variable and the second argument uses the **RIP_PKT** structure (defined in **rip/ripLib.h**) to access the message body. The routine must return **OK** if the message is acceptable, or **ERROR** otherwise. All RIP-2 messages sent to that routine already contain an authentication entry, but have not been verified. (Any unauthenticated RIP-2 messages have already been discarded as required by the RFC specification). RIP-1 messages may be accepted or rejected. RIP-2 messages requesting simple password authentication that match the key are accepted automatically before the hook is called. The remaining RIP-2 messages either did not match that key or are using an unknown authentication type. If any messages are rejected, the MIB-II counters are updated appropriately outside of the hook routine.

The current RIP implementation contains a sample authentication hook that you may add as follows:

```
if (ripAuthHookAdd ("90.0.0.1", ripAuthHook) == ERROR)
    logMsg ("Unable to add authorization hook.\n", 0, 0, 0, 0, 0, 0);
```

The sample routine supports only simple password authentication against the key included in the MIB variable. Since all such messages have already been accepted, all RIP-2 messages received by the routine are discarded. All RIP-1 messages are also discarded, so the hook actually has no effect. The body of that routine is:

```
STATUS ripAuthHook
   (
   char *     pKey,    /* rip2IfConfAuthKey entry from MIB-II family */
   RIP_PKT *  pRip     /* received RIP message */
   )
   {
   if (pRip->rip_vers == 1)
       {
       /*
        * The RFC specification recommends, but does not require, rejecting
        @ version 1 packets when authentication is enabled.
        */
       return (ERROR);
       }
   /*
    @ The authentication type field in the RIP message corresponds to
    @ the first two bytes of the sa_data field overlayed on that
    @ message by the sockaddr structure contained within the RIP_PKT
    @ structure (see rip/ripLib.h).
    */
   if ( (pRip->rip_nets[0].rip_dst.sa_data[0] != 0) ||
       (pRip->rip_nets[0].rip_dst.sa_data[1] !=
       M2_rip2IfConfAuthType_simplePassword))
       {
       /* Unrecognized authentication type. */
       return (ERROR);
       }
   /*
    * Discard version 2 packets requesting simple password authentication
    @ which did not match the MIB variable.
    */
   return (ERROR);
   }
```

A comparison against a different key could be performed as follows:

```
 bzero ( (char *)&key, AUTHKEYLEN);    /* AUTHKEYLEN from rip/m2RipLib.h */
  /*
   @ The start of the authorization key corresponds to the third byte
   @ of the sa_data field in the sockaddr structure overlayed on the
   @ body of the RIP message by the RIP_PKT structure. It continues
   @ for the final 14 bytes of that structure and the first two bytes
   @ of the following rip_metric field.
   */
 bcopy ( (char *)(pRip->rip_nets[0].rip_dst.sa_data + 2),
        (char *)&key, AUTHKEYLEN);
 if (bcmp ( (char *) key, privateKey, AUTHKEYLEN) != 0)
```

```
    {
    /* Key does not match: reject message. */
    return (ERROR);
    }
return (OK);
```

The **ripAuthHookDelete( )** routine will remove the installed function. If authentication is still enabled for the interface, all incoming messages that do not use simple password authentication will be rejected until a routine is provided.

RETURNS **OK**, if hook added; or **ERROR** otherwise.

ERRNO **S_m2Lib_INVALID_PARAMETER**
S_m2Lib_ENTRY_NOT_FOUND

SEE ALSO **ripLib**

ripAuthHookDelete( )

NAME **ripAuthHookDelete( )** – remove an authentication hook from a RIP interface

SYNOPSIS

```
STATUS ripAuthHookDelete
    (
    char* pIpAddr            /* IP address in dotted decimal notation */
    )
```

DESCRIPTION This routine removes an assigned authentication hook from a registered interface indicated by *pIpAddr*. (Interfaces created or changed after a RIP session has started may be installed/updated with the **ripIfSearch( )** and **ripIfReset( )** routines). If authentication is still enabled for the interface, RIP-2 messages using simple password authentication will be accepted if they match the key in the MIB variable, but all other incoming messages will be rejected until a routine is provided.

RETURNS **OK**; or **ERROR**, if the interface could not be found.

ERRNO **S_m2Lib_INVALID_PARAMETER**
S_m2Lib_ENTRY_NOT_FOUND

SEE ALSO **ripLib**

R

ripAuthKeyAdd( )

NAME

ripAuthKeyAdd( ) – add a new RIP authentication key

SYNOPSIS

```
STATUS ripAuthKeyAdd
    (
    char * pInterfaceName,    /* interface to add a key */
    UINT16 keyId,             /* the keyId for this new key */
    char * pKey,              /* the secret key */
    UINT   keyLen,            /* length of the secret key */
    UINT   authProto,         /* auth protocol to use (1 = MD5) */
    ULONG  timeValid          /* number of seconds until key expires */
    )
```

DESCRIPTION

This routine is used to add a new RIP authentication key for a specific interface.

RETURNS

ERROR, if the interface does not exist, or the *keyId* already exists, or if the protocol is not supported; **OK**, if key was entered.

SEE ALSO

ripLib

ripAuthKeyDelete( )

NAME

ripAuthKeyDelete( ) – delete an existing RIP authentication key

SYNOPSIS

```
STATUS ripAuthKeyDelete
    (
    char * pInterfaceName,    /* interface to delete a key from */
    UINT16 keyId              /* the keyId of the key to delete */
    )
```

DESCRIPTION

This routine is used to delete a RIP authentication key for a specific interface.

RETURNS

ERROR, if the interface does not exist, or the *keyId* does not exist; **OK**, if key was deleted.

SEE ALSO

ripLib

ripAuthKeyFind( )

NAME

ripAuthKeyFind( ) – find a RIP authentication key

SYNOPSIS

```
STATUS ripAuthKeyFind
    (
    struct interface * ifp,   /* interface to search for key */
    UINT16             keyId, /* the keyId of the key to search for */
    RIP_AUTH_KEY * *   pKey   /* storage for the key data */
    )
```

DESCRIPTION

This routine is used to find a RIP authentication key based on a specified interface and *keyId*. When a key is found, a pointer to the **RIP_AUTH_KEY** struct for the key is stored in *pKey*.

RETURNS

ERROR, if the key is not found; **OK** if the key was found.

SEE ALSO

ripLib

ripAuthKeyFindFirst( )

NAME

ripAuthKeyFindFirst( ) – find a RIP authentication key

SYNOPSIS

```
STATUS ripAuthKeyFindFirst
    (
    struct interface * ifp,   /* interface to search for key */
    RIP_AUTH_KEY * *   pKey   /* storage for the key data */
    )
```

DESCRIPTION

This routine is used to find a RIP authentication key based on a specified interface. Because a *keyId* is not specified, this routine returns the first non-expired key found for the interface. When a key is found, a pointer to the **RIP_AUTH_KEY** structure for the key is returned in *pKey*.

RETURNS

ERROR, if a key is not found; **OK**, if a key was found.

SEE ALSO

ripLib

R

ripAuthKeyInMD5( )

NAME

ripAuthKeyInMD5( ) – authenticate an incoming RIP-2 message using MD5

SYNOPSIS

```
STATUS ripAuthKeyInMD5
    (
    struct interface * ifp,   /* interface message received on */
    RIP_PKT *          pRip,  /* received RIP message */
    UINT               size   /* length of the RIP message */
    )
```

DESCRIPTION

This routine is used to authenticate an incoming RIP-2 message using the MD5 digest protocol. This authentication scheme is described in RFC 2082.

RETURNS

ERROR, if could not authenticate; **OK**, if authenticated.

SEE ALSO

ripLib

ripAuthKeyOut1MD5( )

NAME

ripAuthKeyOut1MD5( ) – start MD5 authentication of an outgoing RIP-2 message

SYNOPSIS

```
STATUS ripAuthKeyOut1MD5
    (
    struct interface *  pIfp,      /* interface message being sent on */
    struct netinfo *    pNetinfo,  /* pointer to next RIP entry to fill in */
    RIP2_AUTH_PKT_HDR * * ppAuthHdr, /* stores the authentication header */
    RIP_AUTH_KEY * *    ppAuthKey  /* stores the authentication key to use */
    )
```

DESCRIPTION

This routine is used to start the authentication of an outgoing RIP-2 message by adding the authentication header used for MD5 authentication. This authentication scheme is described in RFC 2082. This function returns a pointer the authentication header and a pointer to the looked up authentication key.

RETURNS

ERROR, if a key could not be found; **OK**, if the header was added.

SEE ALSO

ripLib

ripAuthKeyOut2MD5( )

NAME **ripAuthKeyOut2MD5( )** – authenticate an outgoing RIP-2 message using MD5

SYNOPSIS

```
void ripAuthKeyOut2MD5
    (
    RIP_PKT *           pRip,     /* RIP message to authenticate */
    UINT *              pSize,    /* length of the RIP message */
    struct netinfo *    pNetinfo, /* pointer to next RIP entry to fill in */
    RIP2_AUTH_PKT_HDR * pAuthHdr, /* pointer to auth header in the message */
    RIP_AUTH_KEY *      pAuthKey  /* the auth key data to use */
    )
```

DESCRIPTION This routine is used to authenticate an outgoing RIP-2 message using the MD5 digest protocol. This authentication scheme is described in RFC 2082. This function modifies the size given in *pSize* to account for the extra **auth** trailer data. The **auth** trailer is appended to the given **RIP_PKT** and the authentication digest is filled in.

RETURNS N/A

SEE ALSO **ripLib**

ripAuthKeyShow( )

NAME **ripAuthKeyShow( )** – show current authentication configuration

SYNOPSIS

```
void ripAuthKeyShow
    (
    UINT showKey            /* if non-zero then key values are shown */
    )
```

DESCRIPTION This routines shows the current configuration of the authentication keys for each interface.

RETURNS N/A

SEE ALSO **ripLib**

ripDebugLevelSet( )

NAME
ripDebugLevelSet( ) – specify amount of debugging output

SYNOPSIS

```
void ripDebugLevelSet
    (
    int level                 /* verbosity level (0 - 3) */
    )
```

DESCRIPTION
This routine determines the amount of debugging information sent to standard output during the RIP session. Higher values of the *level* parameter result in increasingly verbose output. A *level* of zero restores the default behavior by disabling all debugging output.

RETURNS
N/A

ERRNO
N/A

SEE ALSO
ripLib

ripFilterDisable( )

NAME
ripFilterDisable( ) – prevent strict border gateway filtering

SYNOPSIS

```
void ripFilterDisable (void)
```

DESCRIPTION
This routine configures an active RIP session to ignore the restrictions necessary for RIP-1 and RIP-2 routers to operate correctly in the same network. All border gateway filtering is ignored and all routes to subnets, supernets, and specific hosts will be sent over any available interface. This operation is only correct if no RIP-1 routers are present anywhere on the network. Results are unpredictable if that condition is not met, but high rates of packet loss and widespread routing failures are likely.

The border gateway filtering rules are in force by default.

RETURNS
N/A

ERRNO
N/A

SEE ALSO
ripLib

ripFilterEnable( )

NAME **ripFilterEnable( )** – activate strict border gateway filtering

SYNOPSIS

```
void ripFilterEnable (void)
```

DESCRIPTION This routine configures an active RIP session to enforce the restrictions necessary for RIP-1 and RIP-2 routers to operate correctly in the same network as described in section 3.2 of RFC 1058 and section 3.3 of RFC 1723. When enabled, routes to portions of a logical network (including host routes) are limited to routers within that network. Updates sent outside that network include only a single entry representing the entire network. That entry subsumes all subnets and host-specific routes. If supernets are used, the entry advertises the largest class-based portion of the supernet reachable through the connected interface.

RETURNS N/A

ERRNO N/A

SEE ALSO **ripLib**

ripIfExcludeListAdd( )

NAME **ripIfExcludeListAdd( )** – add an interface to the RIP exclusion list

SYNOPSIS

```
STATUS ripIfExcludeListAdd
    (
    char * pIfName            /* name of interface to be excluded */
    )
```

DESCRIPTION This function adds the interface specified by *ifName* to a list of interfaces on which RIP will not be started. This can be used to prevent RIP from starting on an interface.

RETURNS **OK** if the interface was successfully added to the list; **ERROR** otherwise.

NOTE: This command must be issued prior to the interface being added to the system, as RIP starts on an interface, unless it has been excluded, as soon as an interface comes up. If RIP was already running on the interface which is now desired to be excluded from RIP, the **ripIfReset( )** command should be used after the **ripIfExcludeListAdd( )** command.

SEE ALSO **ripLib**

ripIfExcludeListDelete( )

NAME **ripIfExcludeListDelete( )** – delete an interface from RIP exclusion list

SYNOPSIS

```
STATUS ripIfExcludeListDelete
    (
    char * pIfName            /* name of un-excluded interface */
    )
```

DESCRIPTION This function deletes the interface specified by *ifName* from the list of interfaces on which RIP will not be started. That is, RIP will start on the interface when it is added or comes up.

RETURNS **OK** if the interface was successfully removed from the list;
ERROR otherwise.

NOTE: RIP will not automatically start on the interface. The **ripIfSearch( )** call will need to be made after this call to cause RIP to start on this interface.

SEE ALSO **ripLib**

ripIfExcludeListShow( )

NAME **ripIfExcludeListShow( )** – show the RIP interface exclusion list

SYNOPSIS

```
void ripIfExcludeListShow (void)
```

DESCRIPTION This function prints out the interfaces on which RIP will not be started.

RETURNS Nothing

SEE ALSO **ripLib**

ripIfReset( )

NAME **ripIfReset( )** – alter the RIP configuration after an interface changes

SYNOPSIS

```
STATUS ripIfReset
    (
    char * pIfName            /* name of changed interface */
    )
```

DESCRIPTION This routine updates the interface list and routing tables to reflect address and/or netmask changes for the device indicated by *pIfName*. To accommodate possible changes in the network number, all routes using the named interface are removed from the routing tables, but will be added in the next route update if appropriate. None of the removed routes are poisoned, so it may take some time for the routing tables of all the RIP participants to stabilize if the network number has changed. This routine replaces the existing interface structure with a new one. Thus, any interface specific MIB2 changes that were made to the interface being reset will be lost

RETURNS **OK**, or **ERROR** if named interface not found or not added to list.

ERRNO N/A

SEE ALSO **ripLib**

R

ripIfSearch( )

NAME **ripIfSearch( )** – add new interfaces to the internal list

SYNOPSIS

```
void ripIfSearch (void)
```

DESCRIPTION By default, a RIP session will not recognize any interfaces initialized after it has started. This routine schedules a search for additional interfaces that will occur during the next update of the internal routing table. Once completed, the session will accept and send RIP messages over the new interfaces.

RETURNS N/A

ERRNO N/A

SEE ALSO **ripLib**

ripIfShow( )

NAME

ripIfShow( ) – display the internal interface table maintained by RIP

SYNOPSIS

```
void ripIfShow (void)
```

DESCRIPTION

This routine prints every entry in the local RIP interface table. The interface name, interface index, the UP/DOWN status and the interface address and netmask are displayed.

RETURNS

N/A

ERRNO

N/A

SEE ALSO

ripLib

ripLeakHookAdd( )

NAME

ripLeakHookAdd( ) – add a hook to bypass the RIP and kernel routing tables

SYNOPSIS

```
STATUS ripLeakHookAdd
    (
    char *  pIpAddr,          /* IP address in dotted decimal notation */
    FUNCPTR pLeakHook         /* function pointer to hook */
    )
```

DESCRIPTION

This routine installs a hook routine to support alternative routing protocols for the registered interface given by *pIpAddr*. (Interfaces created or changed after a RIP session has started may be installed/updated with the **ripIfSearch( )** and **ripIfReset( )** routines).

The hook uses the following interface:

```
    STATUS ripLeakHookRtn (long dest, long gateway, long netmask)
```

The RIP session will not add the given route to any tables if the hook routine returns **OK**, but will create a route entry otherwise.

The **ripLeakHookDelete( )** will allow the RIP session to add new routes unconditionally.

RETURNS

OK; or **ERROR**, if the interface could not be found.

ERRNO S_m2Lib_INVALID_PARAMETER
S_m2Lib_ENTRY_NOT_FOUND

SEE ALSO **ripLib**

ripLeakHookDelete( )

NAME **ripLeakHookDelete( )** – remove a table bypass hook from a RIP interface

SYNOPSIS

```
STATUS ripLeakHookDelete
    (
    char* pIpAddr             /* IP address in dotted decimal notation */
    )
```

DESCRIPTION This routine removes the assigned bypass hook from a registered interface indicated by *pIpAddr*. (Interfaces created or changed after a RIP session has started may be installed/updated with the **ripIfSearch( )** and **ripIfReset( )** routines). The RIP session will return to the default behavior and add entries to the internal RIP table and kernel routing table unconditionally.

RETURNS **OK**; or **ERROR**, if the interface could not be found.

ERRNO S_m2Lib_INVALID_PARAMETER
S_m2Lib_ENTRY_NOT_FOUND

SEE ALSO **ripLib**

ripLibInit( )

NAME **ripLibInit( )** – initialize the RIP routing library

SYNOPSIS

```
STATUS ripLibInit
    (
    BOOL supplier,          /* operate in silent mode? */
    BOOL gateway,           /* act as gateway to the Internet? */
    BOOL multicast,         /* use multicast or broadcast addresses? */
    int  version,           /* 1 or 2: selects format of outgoing messages */
    int  timerRate,         /* update frequency for internal routing table */
```

R

```
    int  supplyInterval,  /* update frequency for neighboring routers */
    int  expire,          /* maximum interval for renewing learned routes */
    int  garbage,         /* elapsed time before deleting stale route */
    int  authType         /* default authentication type to use */
    )
```

DESCRIPTION

This routine creates and initializes the global data structures used by the RIP routing library and starts a RIP session to maintain routing tables for a host. You must call **ripLibInit( )** before you can use any other **ripLib** routines. A VxWorks image automatically invokes **ripLibInit( )** if **INCLUDE_RIP** was defined when the image was built.

The resulting RIP session will monitor all network interfaces that are currently available for messages from other RIP routers. If the *supplier* parameter is true, it will also respond to specific requests from other routers and transmit route updates over every known interface at the interval specified by *supplyInterval*.

Specifying a *gateway* setting of true establishes this router as a gateway to the wider Internet, capable of routing packets anywhere within the local networks. The final *multicast* flag indicates whether the RIP messages are sent to the pre-defined multicast address of 224.0.0.9 (which requires a *version* setting of 2) or to the broadcast address of the interfaces.

The *version* parameter determines the format used for outgoing RIP messages, and also sets the initial settings of the MIB-II compatibility switches in combination with the *multicast* flag. A *version* of 1 will restrict all incoming traffic to that older message type. A *version* of 2 will set the receive switch to accept either type unless *multicast* is true, which limits reception to version 2 messages only. SNMP agents may alter those settings on a per-interface basis once startup is complete.

The remaining parameters set various system timers used to maintain the routing table. All of the values are expressed in seconds, and must be greater than or equal to 1. The *timerRate* determines how often the routing table is examined for changes and expired routes. The *supplyInterval* must be an exact multiple of that value. The *expire* parameter specifies the maximum time between updates before a route is invalidated and removed from the kernel table. Expired routes are then deleted from the internal RIP routing table if no update has been received within the time set by the *garbage* parameter.

The following configuration parameters determine the initial values for all these settings. The default timer values match the settings indicated in the RFC specification.

Parameter Name	Default Value	Configuration Parameter
supplier	0 (**FALSE**)	**RIP_SUPPLIER**
gateway	0 (**FALSE**)	**RIP_GATEWAY**
multicast	0 (**FALSE**)	**RIP_MULTICAST**
version	1	**RIP_VERSION**
timerRate	1	**RIP_TIMER_RATE**

Parameter Name	Default Value	Configuration Parameter
supplyInterval	30	**RIP_SUPPLY_INTERVAL**
expire	180	**RIP_EXPIRE_TIME**
garbage	300	**RIP_GARBAGE_TIME**
authType	1	**RIP_AUTH_TYPE**

RETURNS **OK**; or **ERROR**, if configuration fails.

SEE ALSO **ripLib**

ripRouteHookAdd( )

NAME **ripRouteHookAdd( )** – add a hook to install static and non-RIP routes into RIP

SYNOPSIS

```
STATUS ripRouteHookAdd
    (
    FUNCPTR pRouteHook         /* function pointer to hook */
    )
```

DESCRIPTION This routine installs a hook routine that you can use to give RIP the ability to respond to route-add events generated by non-RIP agents. By design, RIP is not interested in the routes generated by non-RIP agents. If you do not install a route hook function, RIP continues this default behavior. However, if you want RIP to add these non-RIP routes to its internal routing database and even propagate routes added by other agents, you must use **ripRouteHookAdd( )** to register a function of the form:

```
STATUS YourRipRouteHookRtn
    (
    struct ROUTE_INFO * pRouteInfo,
    int  protoId,
    BOOL primaryRoute,
    int  flags
    )
```

RIP invokes this function in response to the following events:

1. A non-RIP non-system route was added to the routing table.
2. A route change message arrived.
3. An ICMP redirect message arrived.

The returned function value of the route hook routine tells rip how to respond to the event. In the first case, the returned function value tells RIP whether to add or ignore the new route. In the second case, the returned function tells RIP whether to delete the specified route or change its metric. In the third case, the event is of no direct importance for RIP, so RIP ignores the returned value of the route hook function.

pRouteInfo
This parameter passes in a pointer to a route information structure that stores the routing message. You should not access the contents of this structure directly. Instead, use **ripAddrsXtract( )** to extract the following information:

- destination address
- netmask
- gateway address
- old gateway address (if available)

protoId
This parameter passes in the ID of the protocol that generated the event. Valid protocol IDs are defined in **m2Lib.h** as follows:

M2_ipRouteProto_other (static routes)
M2_ipRouteProto_local
M2_ipRouteProto_netmgmt
M2_ipRouteProto_icmp
M2_ipRouteProto_egp
M2_ipRouteProto_ggp
M2_ipRouteProto_hello
M2_ipRouteProto_rip
M2_ipRouteProto_is_is
M2_ipRouteProto_es_is
M2_ipRouteProto_ciscoIgrp
M2_ipRouteProto_bbnSpfIgp
M2_ipRouteProto_ospf
M2_ipRouteProto_bgp

primaryRoute
This parameter passes in a boolean value that indicates whether the route is a primary route. **TRUE** indicates a primary route. **FALSE** indicates a duplicate route.

flags
This parameter passes in a value that indicates which event occurred:

0 (zero)
This indicates a route added to the routing table by a non-RIP agent.

RIP_ROUTE_CHANGE_RECD
This indicates a route change message.

RIP_REDIRECT_RECD
This indicates and ICMP redirect message.

A New Non-RIP Non-System Route was Added to the Routing Table

In response to this event, RIP needs to be told whether to ignore or add the route. RIP does this on the basis of the returned function value of the route hook routine. In the case of route-add event, RIP interprets the returned function value of the route hook routine as the metric for the route.

If the metric is **HOPCNT_INFINITY**, RIP ignores the route. If the metric is greater than zero but less than **HOPCNT_INFINITY**, RIP considers the route for inclusion. If the route is new to RIP, RIP adds the new route to its internal database, and then propagates the route in its subsequent update messages. If RIP already stores a route for that destination, RIP compares the metric of the new route and the stored route. If the new route has a better (lower) metric, RIP adds the new route. Otherwise, RIP ignores the new route.

When generating its returned function value, your route hook routine can use the creator of the event (*protoID*) as a factor in the decision on whether to include the route. For example, if you wanted the route hook to tell RIP to ignore all non-RIP routes except static routes, your route hook would return **HOPCNT_INFINITY** if the *protoID* were anything other than **M2_ipRouteProto_other**. Thus, your route hook routine is a vehicle through which you can implement a policy for including non-RIP routes in the RIP internal route data base.

When designing your policy, you should keep in mind how RIP prioritizes these non-RIP routes and when it deletes these non-RIP routes. For example, non-RIP routes never time out. They remain in the RIP database until one of the following events occurs:

1. An agent deletes the route from the system routing table.
2. An agent deletes the interface through which the route passes.
3. A route change message for the route arrives.

Also, these non-RIP routes take precedence over RIP routes to the same destination. RIP ignores routes learned from RIP peers if a route to the same destination was recommended by the hook routine. This non-RIP route takes precedence over the RIP route without regard of the route metric. However, if the route hook routine adds multiple same-destination routes, the route with the lowest metric takes precedence. If the route hook route approves multiple same-metric same-destination routes, the most recently added route is installed.

A Route Change Notification Arrived

In response to this event, RIP needs to be told whether to delete the route or change its metric. If the hook returns a value greater than or equal to **HOPCNT_INFINITY**, RIP deletes the route from its internal routing data base. If the hook routine returns a valid metric (a value greater than zero but less than **HOPCNT_INFINITY**), RIP reassigns the routes metric to equal the returned value of the route hook routine. If the returned value of the route hook route is invalid (less than zero) RIP ignores the event. RIP also ignores the event if the route specified in *pRouteInfo* is not one stored in its internal data base.

An ICMP Redirect Message Arrived

In response to this event, RIP never needs to make any changes to its internal routing database. Thus, RIP ignores the returned function value of the route hook routine called in response to an ICMP redirect message. However, if the event is of interest to your particular environment, and it makes sense to catch the event in the context of the RIP task, you can use the route hook routine to do so.

Within your route hook routine, your can recognize an ICMP event by checking whether the flags parameter value sets the **RIP_REDIRECT_RECD** bit. The *primaryRoute* parameter passes in a boolean value that indicates whether the route is primary route. If the *primaryRoute* passes in **FALSE**, the route hook routine need will most likely need to do nothing more. If this parameter passes in **TRUE**, take whatever action (if any) that you know to be appropriate to your particular environment.

RETURNS **OK**; or **ERROR**, if RIP is not initialized.

SEE ALSO **ripLib**

ripRouteHookDelete( )

NAME **ripRouteHookDelete( )** – remove the route hook

SYNOPSIS

```
STATUS ripRouteHookDelete (void)
```

DESCRIPTION This routine removes the route hook installed earlier by the **ripRouteHookAdd( )** routine. This will cause RIP to ignore any routes added to the system Routing database.

RETURNS **OK**; or **ERROR**, if RIP is not initialized.

SEE ALSO **ripLib**

ripRouteShow( )

NAME **ripRouteShow( )** – display the internal routing table maintained by RIP

SYNOPSIS

```
void ripRouteShow (void)
```

DESCRIPTION This routine prints every entry in the local RIP routing table. The flags displayed below the destination, gateway, and netmask addresses indicate the current route status. Entries with the **RTS_INTERFACE** flag indicate locally generated routes to directly connected networks. If **RTS_SUBNET** is set for an entry, it is subject to border gateway filtering (if enabled). When **RTS_INTERNAL** is also present, the corresponding entry is an "artificial" route created to supply distant networks with legitimate destinations if border filtering excludes the actual entry. Those entries are not copied to the kernel routing table. The **RTS_CHANGED** flag marks entries added or modified in the last timer interval that will be included in a triggered update. The **RTS_OTHER** flag is set for routes learnt from other sources. The **RTS_PRIMARY** flag (set only if the **RTS_OTHER** flag is also set) indicates that the route is a primary route, visible to the IP forwarding process. The DOWN flag indicates that the interface through which the gateway is reachable is down.

SEE ALSO **ripLib**

ripSendHookAdd( )

NAME **ripSendHookAdd( )** – add an update filter to a RIP interface

SYNOPSIS

```
STATUS ripSendHookAdd
    (
    char* pIpAddr,            /* IP address in dotted decimal notation */
    BOOL (* ripSendHook) (struct rt_entry* pRt)
                              /* Routine to use. */
    )
```

DESCRIPTION This routine installs a hook routine to screen individual route entries for inclusion in a periodic update. The routine is installed for the registered interface given by *pIpAddr*. (Interfaces created or changed after a RIP session has started may be installed/updated with the **ripIfSearch( )** and **ripIfReset( )** routines).

The hook uses the following prototype:

```
BOOL ripSendHookRtn (struct rt_entry* pRt);
```

If the hook returns **FALSE**, the route is not included in the update. Otherwise, it is included if it meets the other restrictions, such as simple split horizon and border gateway filtering. The **ripSendHookDelete( )** routine removes this additional filter from the output processing.

RETURNS **OK**; or **ERROR**, if the interface could not be found.

ERRNO **S_m2Lib_INVALID_PARAMETER**
S_m2Lib_ENTRY_NOT_FOUND

SEE ALSO **ripLib**

R

ripSendHookDelete( )

NAME **ripSendHookDelete( )** – remove an update filter from a RIP interface

SYNOPSIS

```
STATUS ripSendHookDelete
    (
    char* pIpAddr            /* IP address in dotted decimal notation */
    )
```

DESCRIPTION This routine removes the hook routine that allowed additional screening of route entries in periodic updates from the registered interface indicated by *pIpAddr*. (Interfaces created or changed after a RIP session has started may be installed/updated with the **ripIfSearch( )** and **ripIfReset( )** routines). The RIP session will return to the default behavior and include any entries that meet the other restrictions (such as simple split horizon).

RETURNS **OK**; or **ERROR**, if the interface could not be found.

ERRNO **S_m2Lib_INVALID_PARAMETER**
S_m2Lib_ENTRY_NOT_FOUND

SEE ALSO **ripLib**

ripShutdown( )

NAME **ripShutdown( )** – terminate all RIP processing

SYNOPSIS

```
STATUS ripShutdown (void)
```

DESCRIPTION This routine "poisons" all routes in the current table by transmitting updates with an infinite metric for each entry over all available interfaces. It then halts all RIP processing and removes the associated tasks and data structures. When completed successfully, the RIP services are unavailable until restarted with the **ripLibInit( )** routine.

RETURNS **OK** if shutdown completed, or **ERROR** otherwise.

ERRNO N/A

SEE ALSO **ripLib**

rlogin( )

NAME **rlogin( )** – log in to a remote host

SYNOPSIS

```
STATUS rlogin
    (
    char * host             /* name of host to connect to */
    )
```

DESCRIPTION This routine allows users to log in to a remote host. It may be called from the VxWorks shell as follows:

```
-> rlogin "remoteSystem"
```

where *remoteSystem* is either a host name, which has been previously added to the remote host table by a call to **hostAdd( )**, or an Internet address in dot notation (*e.g.*, "90.0.0.2"). The remote system will be logged into with the current user name as set by a call to **iam( )**.

The user disconnects from the remote system by typing:

```
~.
```

as the only characters on the line, or by simply logging out from the remote system using **logout( )**.

RETURNS **OK**, or **ERROR** if the host is unknown, no privileged ports are available, the routine is unable to connect to the host, or the child process cannot be spawned.

SEE ALSO **rlogLib**, **iam( )**, **logout( )**

R

rlogind( )

NAME **rlogind( )** – the VxWorks remote login daemon

SYNOPSIS

```
void rlogind (void)
```

DESCRIPTION This routine provides a facility for remote users to log in to VxWorks over the network. If **INCLUDE_RLOGIN** is defined, **rlogind( )** is spawned by **rlogInit( )** at boot time.

Remote login requests will cause **stdin**, **stdout**, and **stderr** to be directed away from the console. When the remote user disconnects, **stdin**, **stdout**, and **stderr** are restored, and the shell is restarted. The **rlogind( )** routine uses the remote user verification protocol specified by the UNIX remote shell daemon documentation, but ignores all the information except the user name, which is used to set the VxWorks remote identity (see the manual entry for **iam( )**).

The remote login daemon requires the existence of a pseudo-terminal device, which is created by **rlogInit( )** before **rlogind( )** is spawned. The **rlogind( )** routine creates two child processes, **tRlogInTask** and **tRlogOutTask**, whenever a remote user is logged in. These processes exit when the remote connection is terminated.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS N/A

SEE ALSO **rlogLib**, **rlogInit( )**, **iam( )**

rlogInit( )

NAME **rlogInit( )** – initialize the remote login facility

SYNOPSIS

```
STATUS rlogInit (void)
```

DESCRIPTION This routine initializes the remote login facility. It creates a pty (pseudo tty) device and spawns **rlogind( )**. If **INCLUDE_RLOGIN** is included, **rlogInit( )** is called automatically at boot time.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS **OK** or **ERROR**.

SEE ALSO **rlogLib**, **ptyDrv**

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.

SEE ALSO

usrFsLib, **remove( )**, *VxWorks Programmer's Guide: Target Shell*

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.

SEE ALSO

usrFsLib, **mkdir( )**, *VxWorks Programmer's Guide: Target Shell*

rngBufGet( )

NAME

rngBufGet( ) – get characters from a ring buffer

SYNOPSIS

```
int rngBufGet
    (
    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.

SEE ALSO

rngLib

rngBufPut( )

NAME

rngBufPut( ) – put bytes into a ring buffer

SYNOPSIS

```
int rngBufPut
    (
    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.

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.

SEE ALSO

rngLib

rngDelete( )

NAME

rngDelete( ) – delete a ring buffer

SYNOPSIS

```
void rngDelete
    (
    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

SEE ALSO

rngLib

rngFlush( )

NAME **rngFlush( )** – make a ring buffer empty

SYNOPSIS

```
void rngFlush
    (
    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

SEE ALSO **rngLib**

rngFreeBytes( )

NAME **rngFreeBytes( )** – determine the number of free bytes in a ring buffer

SYNOPSIS

```
int rngFreeBytes
    (
    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.

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.

SEE ALSO

rngLib

rngIsFull( )

NAME

rngIsFull( ) – test if a ring buffer is full (no more room)

SYNOPSIS

```
BOOL rngIsFull
    (
    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.

SEE ALSO

rngLib

rngMoveAhead( )

NAME **rngMoveAhead( )** – advance a ring pointer by *n* bytes

SYNOPSIS

```
void rngMoveAhead
    (
    RING_ID ringId,         /* ring buffer to be advanced */
    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

SEE ALSO **rngLib**

rngNBytes( )

NAME **rngNBytes( )** – determine the number of bytes in a ring buffer

SYNOPSIS

```
int rngNBytes
    (
    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.

SEE ALSO **rngLib**

rngPutAhead( )

NAME **rngPutAhead( )** – put a byte ahead in a ring buffer without moving ring pointers

SYNOPSIS

```
void rngPutAhead
    (
    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 **RNG_ELEM_PUT** would, 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

SEE ALSO **rngLib**

R

romStart( )

NAME **romStart( )** – generic ROM initialization

SYNOPSIS

```
void romStart
    (
    int startType             /* start type */
    )
```

DESCRIPTION This is the first C code executed after reset.

This routine is called by the assembly start-up code in **romInit( )**. It clears memory, copies ROM to RAM, and possibly invokes the uncompresser. It then jumps to the entry point of the uncompressed object code.

RETURNS N/A

SEE ALSO **bootInit**

round( )

NAME **round( )** – round a number to the nearest integer

SYNOPSIS

```
double round
    (
    double x  /* value to round */
    )
```

DESCRIPTION This routine rounds a double-precision value *x* to the nearest integral value.

INCLUDE FILES **math.h**

RETURNS The double-precision representation of *x* rounded to the nearest integral value.

SEE ALSO **mathALib**

roundf( )

NAME **roundf( )** – round a number to the nearest integer

SYNOPSIS

```
float roundf
    (
    float x   /* argument */
    )
```

DESCRIPTION This routine rounds a single-precision value *x* to the nearest integral value.

INCLUDE FILES **math.h**

RETURNS The single-precision representation of *x* rounded to the nearest integral value.

SEE ALSO **mathALib**

routeAdd( )

NAME

routeAdd( ) – add a route

SYNOPSIS

```
STATUS routeAdd
    (
    char * destination,  /* inet addr or name of route destination */
    char * gateway       /* inet addr or name of gateway to destination */
    )
```

DESCRIPTION

This routine adds gateways to the network routing tables. It is called from a VxWorks machine that needs to establish a gateway to a destination network (or machine).

You can specify both *destination* and *gateway* in standard Internet address format (for example, 90.0.0.2), or you can specify them using their host names, as specified with **hostAdd( )**.

This routine can be used to add multiple routes to the same destination differing by the gateway.

EXAMPLE

Consider the following example:

```
-> routeAdd "90.0.0.0", "gate"
```

This call tells VxWorks that the machine with the host name "gate" is the gateway to network 90.0.0.0. The host "gate" must already have been created by **hostAdd( )**.

Consider the following example:

```
-> routeAdd "90.0.0.0", "91.0.0.3"
```

This call tells VxWorks that the machine with the Internet address 91.0.0.3 is the gateway to network 90.0.0.0.

Consider the following example:

```
-> routeAdd "destination", "gate"
```

This call tells VxWorks that the machine with the host name "gate" is the gateway to the machine named "destination". The host names "gate" and "destination" must already have been created by **hostAdd( )**.

Consider the following example:

```
-> routeAdd "0", "gate"
```

This call tells VxWorks that the machine with the host name "gate" is the default gateway. The host "gate" must already have been created by **hostAdd( )**. A default gateway is where Internet Protocol (IP) datagrams are routed when there is no specific routing table entry available for the destination IP network or host.

RETURNS **OK** or **ERROR**.

SEE ALSO **routeLib**

routeDelete( )

NAME **routeDelete( )** – delete a route

SYNOPSIS

```
STATUS routeDelete
    (
    char * destination,    /* inet addr or name of route destination */
    char * gateway         /* inet addr or name of gateway to destination */
    )
```

DESCRIPTION This routine deletes a specified route from the network routing tables.

RETURNS **OK** or **ERROR**.

SEE ALSO **routeLib**, **routeAdd( )**

routeEntryAdd( )

NAME **routeEntryAdd( )** – insert a route in the routing table

SYNOPSIS

```
STATUS routeEntryAdd
    (
    ROUTE_DESC * pRouteDesc   /* information for new route entry */
    )
```

DESCRIPTION This routine adds a route to the routing table. The *pRouteDesc* argument must include a destination address, gateway, and protocol identifier. If that argument does not include a netmask or specifies a netmask value of 0, the system creates a host-specific route entry. The *value1* through *value5*, and *routeTag* fields store arbitrary values for the new entry. The required *weight* field indicates the relative priority of the route (from 1 to 255) in case other entries to the same destination exist. The route with the lowest weight is visible to the IP forwarding process. A value of 0 will create an entry with the default weight value.

This routine ignores any values in the *flags*, *pIf*, and *pData* fields in the provided structure. If the add attempt is successful, the system sends callback messages and routing socket messages announcing the existence of the new route.

RETURNS **OK** on success and **ERROR** on failure.

SEE ALSO **routeEntryLib**

routeEntryDel( )

NAME **routeEntryDel( )** – remove a route from the routing table

SYNOPSIS

```
STATUS routeEntryDel
    (
    ROUTE_DESC * pRouteDesc   /* information for deleted route */
    )
```

DESCRIPTION This routine deletes a route in the routing table. The *pRouteDesc* argument must include a destination address. If that argument does not include a netmask or specifies a netmask value of 0, the system attempts to delete a host-specific route to the destination. If a route which matches the destination and netmask exists, a protocol ID of zero attempts to delete that entry (which is visible to the IP forwarding process) if the gateway value is not equal to zero. Otherwise, the system attempts to remove the first (lowest weight) entry which matches the provided protocol, or a specific entry within the first protocol group which also matches the supplied gateway address.

NOTE: This routine stores the actual gateway value in the *pRouteDesc* structure, so the corresponding buffer must be supplied even if no specific value is assigned. This routine does not use any fields in the *pRouteDesc* structure except the destination, gateway, netmask and protocol ID.

RETURNS **OK** on success, **ERROR** on failure

SEE ALSO **routeEntryLib**

routeEntryLookup( )

NAME

routeEntryLookup( ) – find a matching route for a destination

SYNOPSIS

```
STATUS routeEntryLookup
    (
    struct sockaddr * pDest,   /* IP address reachable with matching route */
    ULONG *           pMask,   /* netmask value, in network byte order */
    int               protoId, /* route source from m2Lib.h, or 0 for any. */
    ROUTE_DESC *    pRouteDesc /* information for matching route */
    )
```

DESCRIPTION

This routine searches the routing table for an entry which covers the specified destination address. It provides four types of searches based on the values of the *protoId* and *pMask* arguments.

If no mask is present (*pMask* is **NULL**) the search finds the matching entry with the longest netmask. Otherwise, the search ignores entries whose netmasks permit a match against the destination but do not equal the given value. Likewise, if *protoId* is not zero, the search restricts the possible matches to the specified route source.

Mask values of zero and 0xffffffff both indicate a host-specific route.

If neither value is specified, the search duplicates the results of the IP forwarding process for the destination (assuming no type-of-service match is required). It retrieves the matching entry with the longest netmask, regardless of the source which created it.

In all cases, if multiple entries match the search criteria, this routine selects the oldest one.

The chosen entry is copied into the supplied *pRouteDesc* structure, which is not modified if the search fails.

RETURNS

OK if a route is found, or **ERROR** otherwise.

SEE ALSO

routeEntryLib

routeModify( )

NAME

routeModify( ) – change an entry in the routing table

SYNOPSIS

```
STATUS routeModify
    (
    ROUTE_DESC *      pRouteDesc, /* information for matching route */
    struct sockaddr * pNewGateway /* new gateway, NULL if unchanged */
    )
```

DESCRIPTION

This routine searches the routing table for an entry which matches the destination address and netmask in the *pRouteDesc* structure. If the route descriptor structure does not include a netmask, it selects the longest netmask for the matching destination. A netmask value of zero searches for a host-specific route to the destination. A protocol ID of zero selects the first entry which matches the destination address and any netmask value. Otherwise, the search finds the route with the specified protocol. The retrieved route also matches any specified gateway value.

The *pNewGateway* argument supplies an optional replacement gateway address. The new address must be reachable through one of the local interfaces or the modification fails. The modification also fails if the destination address is not specified or if no route which matches the search criteria is found.

Once a route is chosen, this routine replaces the current metric values, route weight, and route tag with the corresponding entries in the *pRouteDesc* structure. The pointers for the interface and additional data in the *pRouteDesc* argument are not used. The route flags are also not changed.

NOTE: Changing the weight of a route will reorganize any duplicate routes and may alter which entry is visible to the IP forwarding process.

RETURNS

OK if a route is found and changed, or **ERROR** otherwise.

SEE ALSO

routeEntryLib

routeNetAdd( )

NAME

routeNetAdd( ) – add a route to a destination that is a network

SYNOPSIS

```
STATUS routeNetAdd
    (
    char * destination,    /* inet addr or name of network destination */
    char * gateway         /* inet addr or name of gateway to destination */
    )
```

DESCRIPTION

This routine is equivalent to **routeAdd( )**, except that the destination address is assumed to be a network. This is useful for adding a route to a sub-network that is not on the same overall network as the local network.

This routine can be used to add multiple routes to the same destination differing by the gateway.

RETURNS

OK or **ERROR**.

SEE ALSO

routeLib

routeShow( )

NAME

routeShow( ) – display all IP routes (summary information)

SYNOPSIS

```
void routeShow (void)
```

DESCRIPTION

This routine displays the list of destinations in the routing table along with the next-hop gateway and associated interface for each entry. It separates the routes into network routes and host-specific entries, but does not display the netmask for a route since it was created for class-based routes which used predetermined values for that field.

The IP forwarding process will only use the first route entry to a destination. When multiple routes exist to the same destination with the same netmask (which is not shown), the first route entry uses the lowest administrative weight. The remaining entries (listed as additional routes) use the same destination address. One of those entries will replace the primary route if it is deleted.

EXAMPLE

```
-> routeShow
ROUTE NET TABLE
Destination      Gateway          Flags  Refcnt Use        Interface
```

```
--------------------------------------------------------------------
90.0.0.0          90.0.0.63         0x1    1      142        enp0
10.0.0.0          90.0.0.70         0x1    1      142        enp0
  Additional routes to 10.0.0.0:
                  80.0.0.70         0x1    0      120        enp1
--------------------------------------------------------------------
ROUTE HOST TABLE
Destination       Gateway           Flags  Refcnt Use        Interface
--------------------------------------------------------------------
127.0.0.1         127.0.0.1         0x101  0      82         lo0
--------------------------------------------------------------------
```

The flags field represents a decimal value of the flags specified for a given route. The following is a list of currently available flag values:

0x1	- route is usable (that is, "up")
0x2	- destination is a gateway
0x4	- host specific routing entry
0x8	- host or net unreachable
0x10	- created dynamically (by redirect)
0x20	- modified dynamically (by redirect)
0x40	- message confirmed
0x80	- subnet mask present
0x100	- generate new routes on use
0x200	- external daemon resolves name
0x400	- generated by ARP
0x800	- manually added (static)
0x1000	- just discard packets (during updates)
0x2000	- modified by management protocol
0x4000	- protocol specific routing flag
0x8000	- protocol specific routing flag

In the above display example, the entry in the ROUTE NET TABLE has a flag value of 1, which indicates that this route is "up" and usable and network specific (the 0x4 bit is turned off). The entry in the ROUTE HOST TABLE has a flag value of 5 (0x1 OR'ed with 0x4), which indicates that this route is "up" and usable and host-specific.

Some configuration is required when this routine is to be used remotely over the network, *e.g.*, through a **telnet** session or through the host shell using **WDB_COMM_NETWORK**. If, more than 5 routes are expected in the table the parameter **RT_BUFFERED_DISPLAY** should be set to **TRUE** to prevent a possible deadlock. This requires a buffer whose size can be set with **RT_DISPLAY_MEMORY**. It will limit the number of routes that can be displayed (each route requires approx. 70 bytes).

RETURNS N/A

SEE ALSO **netShow**

R

routestatShow( )

NAME **routestatShow( )** – display routing statistics

SYNOPSIS

```
void routestatShow (void)
```

DESCRIPTION This routine displays routing statistics.

RETURNS N/A

SEE ALSO **netShow**

routeStorageUnbind( )

NAME **routeStorageUnbind( )** – remove a registered handler from the routing system

SYNOPSIS

```
STATUS routeStorageUnbind
    (
    void * pCookie          /* identifier from routeStorageBind() routine */
    )
```

DESCRIPTION A routing protocol uses this routine to prevent a registered function from receiving any callback messages. Any data accessible with the extra argument to that function must be maintained until this routine completes successfully.

RETURNS **OK** if removal succeeds, or **ERROR** otherwise.

SEE ALSO **routeMessageLib**

routeTableWalk( )

NAME

routeTableWalk( ) – traverse the IP routing table

SYNOPSIS

```
STATUS routeTableWalk
    (
    struct sockaddr * pDest,   /* destination address, or NULL if none. */
    int               protoId, /* route source, or 0 for any. */
    VOIDFUNCPTR       pFunc,   /* callback function */
    void *            pArg     /* optional callback function argument */
    )
```

DESCRIPTION

This routine applies the provided function to every entry in the IP routing table which meets the criteria indicated by the *pDest*and *protoId* arguments. If a destination address is specified, the given function executes for each route table entry which matches the destination. If a protocol identifier is supplied, the function executes for each entry created by the protocol instead. If no value is specified, the routine displays every entry in the table. The supplied argument *pArg* is passed back to callback function.

RETURNS

OK if traversal completes, or **ERROR** otherwise.

NOTE: Only one of the two values *pDest* and *protoId* should be specified. Specifying both results in **ERROR** being returned.

NOTE: The provided routine executes while the system holds internal locks which restrict all network stack activity and any routing operations to the calling task. That routine MUST NOT perform any operations which alter the existing routing table. This walk routine relies on a fixed order of all route entries to complete. Creating or removing route entries could corrupt the table, causing the calling task to enter an endless loop or halt completely. That behavior would deadlock the entire network system, since other tasks would wait indefinitely for the unavailable locks.

SEE ALSO

routeEntryLib

rpcInit( )

NAME **rpcInit( )** – initialize the RPC package

SYNOPSIS

```
STATUS rpcInit (void)
```

DESCRIPTION This routine must be called before any task can use the RPC facility; it spawns the **portmap** daemon. It is called automatically if **INCLUDE_RPC** is defined.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS **OK**, or **ERROR** if the portmap daemon cannot be spawned.

SEE ALSO **rpcLib**

rpcTaskInit( )

NAME **rpcTaskInit( )** – initialize a task's access to the RPC package

SYNOPSIS

```
STATUS rpcTaskInit (void)
```

DESCRIPTION This routine must be called by a task before it makes any calls to other routines in the RPC package.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS **OK**, or **ERROR** if there is insufficient memory or the routine is unable to add a task delete hook.

SEE ALSO **rpcLib**

rresvport( )

NAME **rresvport( )** – open a socket with a privileged port bound to it

SYNOPSIS

```
int rresvport
    (
    int * alport             /* port number to initially try */
    )
```

DESCRIPTION This routine opens a socket with a privileged port bound to it. It is analogous to the UNIX routine **rresvport( )**.

RETURNS A socket descriptor, or **ERROR** if either the socket cannot be opened or all ports are in use.

SEE ALSO **remLib**, UNIX BSD 4.3 manual entry for **rresvport( )**

rt11FsDateSet( )

NAME **rt11FsDateSet( )** – set the rt11Fs file system date

SYNOPSIS

```
void rt11FsDateSet
    (
    int year,          /* year (72...03 (RT-11's days are numbered)) */
    int month,         /* month (0, or 1...12) */
    int day            /* day (0, or 1...31) */
    )
```

DESCRIPTION This routine sets the date for the rt11Fs file system, which remains in effect until changed. All files created are assigned this creation date.

To set a blank date, invoke the command:

```
rt11FsDateSet (72, 0, 0);    /* a date outside RT-11's epoch */
```

NOTE: No automatic incrementing of the date is performed; each new date must be set with a call to this routine.

RETURNS N/A

SEE ALSO **rt11FsLib**

R

rt11FsDevInit( )

NAME

rt11FsDevInit( ) – initialize the rt11Fs device descriptor

SYNOPSIS

```
RT_VOL_DESC *rt11FsDevInit
    (
    char *    devName,        /* device name */
    BLK_DEV * pBlkDev,        /* pointer to block device info */
    BOOL      rt11Fmt,        /* TRUE if RT-11 skew & interleave */
    int       nEntries,       /* no. of dir entries incl term entry */
    BOOL      changeNoWarn    /* TRUE if no disk change warning */
    )
```

DESCRIPTION

This routine initializes the device descriptor. The *pBlkDev* parameter is a pointer to an already-created **BLK_DEV** device structure. This structure contains definitions for various aspects of the physical device format, as well as pointers to the sector read, sector write, **ioctl( )**, status check, and reset functions for the device.

The *rt11Fmt* parameter is **TRUE** if the device is to be accessed using standard RT-11 skew and interleave.

The device directory will consist of one segment able to contain at least as many files as specified by *nEntries*. If *nEntries* is equal to **RT_FILES_FOR_2_BLOCK_SEG**, strict RT-11 compatibility is maintained.

The *changeNoWarn* parameter is **TRUE** if the disk may be changed without announcing the change via **rt11FsReadyChange( )**. Setting *changeNoWarn* to **TRUE** causes the disk to be regularly remounted, in case it has been changed. This results in a significant performance penalty.

NOTE: An **ERROR** is returned if *rt11Fmt* is **TRUE** and the **bd_blksPerTrack** (sectors per track) field in the **BLK_DEV** structure is odd. This is because an odd number of sectors per track is incompatible with the RT-11 interleaving algorithm.

RETURNS

A pointer to the volume descriptor (**RT_VOL_DESC**), or **NULL** if invalid device parameters were specified, or the routine runs out of memory.

SEE ALSO

rt11FsLib

rt11FsInit( )

NAME

rt11FsInit( ) – prepare to use the rt11Fs library

SYNOPSIS

```
STATUS rt11FsInit
    (
    int maxFiles           /* max no. of simultaneously open rt11Fs files */
    )
```

DESCRIPTION

This routine initializes the rt11Fs library. It must be called exactly once, before any other routine in the library. The *maxFiles* parameter specifies the number of rt11Fs files that may be open at once. This routine initializes the necessary memory structures and semaphores.

This routine is called automatically from the root task, **usrRoot( )**, in **usrConfig.c** when the configuration macro **INCLUDE_RT11FS** is defined.

RETURNS

OK, or **ERROR** if memory is insufficient.

SEE ALSO

rt11FsLib

rt11FsMkfs( )

NAME

rt11FsMkfs( ) – initialize a device and create an rt11Fs file system

SYNOPSIS

```
RT_VOL_DESC *rt11FsMkfs
    (
    char *    volName,        /* volume name to use */
    BLK_DEV * pBlkDev         /* pointer to block device struct */
    )
```

DESCRIPTION

This routine provides a quick method of creating an rt11Fs file system on a device. It is used instead of the two-step procedure of calling **rt11FsDevInit( )** followed by an **ioctl( )** call with an **FIODISKINIT** function code.

This routine provides defaults for the rt11Fs parameters expected by **rt11FsDevInit( )**. The directory size is set to **RT_FILES_FOR_2_BLOCK_SEG**(defined in **rt11FsLib.h**). No standard disk format is assumed; this allows the use of rt11Fs on block devices with an odd number of sectors per track. The *changeNoWarn* parameter is defined as **FALSE**, indicating that the disk will not be replaced without **rt11FsReadyChange( )** being called first.

R

If different values are needed for any of these parameters, the routine **rt11FsDevInit( )** must be used instead of this routine, followed by a request for disk initialization using the **ioctl( )** function **FIODISKINIT**.

RETURNS

A pointer to an rt11Fs volume descriptor (**RT_VOL_DESC**), or **NULL** if there is an error.

SEE ALSO

rt11FsLib, **rt11FsDevInit( )**

rt11FsModeChange( )

NAME

rt11FsModeChange( ) – modify the mode of an rt11Fs volume

SYNOPSIS

```
void rt11FsModeChange
    (
    RT_VOL_DESC * vdptr,      /* pointer to volume descriptor */
    int           newMode     /* O_RDONLY, O_WRONLY, or O_RDWR (both) */
    )
```

DESCRIPTION

This routine sets the volume descriptor mode to *newMode*. It should be called whenever the read and write capabilities are determined, usually after a ready change. See the manual entry for **rt11FsReadyChange( )**.

The **rt11FsDevInit( )** routine initially sets the mode to **O_RDWR**, (*e.g.*, both **O_RDONLY** and **O_WRONLY**).

RETURNS

N/A

SEE ALSO

rt11FsLib, **rt11FsDevInit( )**, **rt11FsReadyChange( )**

rt11FsReadyChange( )

NAME

rt11FsReadyChange( ) – notify rt11Fs of a change in ready status

SYNOPSIS

```
void rt11FsReadyChange
    (
    RT_VOL_DESC * vdptr       /* pointer to device descriptor */
    )
```

DESCRIPTION

This routine sets the volume descriptor state to **RT_VD_READY_CHANGED**. It should be called whenever a driver senses that a device has come on-line or gone off-line (*e.g.*, a disk has been inserted or removed).

RETURNS

N/A

SEE ALSO

rt11FsLib

s( )

NAME

s( ) – single-step a task

SYNOPSIS

```
STATUS s
    (
    int      taskNameOrId,      /* task to step; 0 = use default */
    INSTR * addr,               /* address to step to; 0 = next instruction */
    )
```

DESCRIPTION

This routine single-steps a task that is stopped at a breakpoint.

To execute, enter:

```
-> s [task[,addr[,addr1]]]
```

If *task* is omitted or zero, the last task referenced is assumed. If *addr* is non-zero, then the program counter is changed to *addr*; if *addr1* is non-zero, the next program counter is changed to *addr1*, and the task is stepped.

WARNING: When a task is continued, **s( )** does not distinguish between a suspended task or a task suspended by the debugger. Therefore, its use should be restricted to only those tasks being debugged.

RETURNS

OK, or **ERROR** if the debugging package is not installed, the task cannot be found, or the task is not suspended.

SEE ALSO

dbgLib, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

scanf( )

NAME

scanf( ) – read and convert characters from the standard input stream (ANSI)

SYNOPSIS

```
int scanf
    (
    char const * fmt,           /* format string */
                 ...            /* arguments to format string */
    )
```

DESCRIPTION

This routine reads input from the standard input stream under the control of the string *fmt*. It is equivalent to **fscanf( )** with an *fp* argument of **stdin**.

INCLUDE FILES **stdio.h**

RETURNS The number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure; or EOF if an input failure occurs before any conversion.

SEE ALSO **ansiStdio**, **fscanf( )**, **sscanf( )**

sched_get_priority_max( )

NAME **sched_get_priority_max( )** – get the maximum priority (POSIX)

SYNOPSIS

```
int sched_get_priority_max
    (
    int policy                /* scheduling policy */
    )
```

DESCRIPTION This routine returns the value of the highest possible task priority for a specified scheduling policy (**SCHED_FIFO** or **SCHED_RR**).

NOTE: If the global variable **posixPriorityNumbering** is **FALSE**, the VxWorks native priority numbering scheme is used, in which higher priorities are indicated by smaller numbers. This is different than the priority numbering scheme specified by POSIX, in which higher priorities are indicated by larger numbers.

RETURNS Maximum priority value, or -1 (**ERROR**) on error.

ERRNO **EINVAL**
- invalid scheduling policy.

SEE ALSO **schedPxLib**

S

sched_get_priority_min( )

NAME

sched_get_priority_min( ) – get the minimum priority (POSIX)

SYNOPSIS

```
int sched_get_priority_min
    (
    int policy                /* scheduling policy */
    )
```

DESCRIPTION

This routine returns the value of the lowest possible task priority for a specified scheduling policy (**SCHED_FIFO** or **SCHED_RR**).

NOTE: If the global variable **posixPriorityNumbering** is **FALSE**, the VxWorks native priority numbering scheme is used, in which higher priorities are indicated by smaller numbers. This is different than the priority numbering scheme specified by POSIX, in which higher priorities are indicated by larger numbers.

RETURNS

Minimum priority value, or -1 (**ERROR**) on error.

ERRNO

EINVAL
- invalid scheduling policy.

SEE ALSO

schedPxLib

sched_getparam( )

NAME

sched_getparam( ) – get the scheduling parameters for a specified task (POSIX)

SYNOPSIS

```
int sched_getparam
    (
    pid_t                tid,  /* task ID */
    struct sched_param * param /* scheduling param to store priority */
    )
```

DESCRIPTION

This routine gets the scheduling priority for a specified task, *tid*. If *tid* is 0, it gets the priority of the calling task. The task's priority is copied to the **sched_param** structure pointed to by *param*.

NOTE: If the global variable **posixPriorityNumbering** is **FALSE**, the VxWorks native priority numbering scheme is used, in which higher priorities are indicated by smaller numbers. This is different than the priority numbering scheme specified by POSIX, in which higher priorities are indicated by larger numbers.

RETURNS 0 (**OK**) if successful, or -1 (**ERROR**) on error.

ERRNO **ESRCH**
- invalid task ID.

SEE ALSO **schedPxLib**

sched_getscheduler( )

NAME **sched_getscheduler( )** – get the current scheduling policy (POSIX)

SYNOPSIS

```
int sched_getscheduler
    (
    pid_t tid                 /* task ID */
    )
```

DESCRIPTION This routine returns the currents scheduling policy (*i.e.*, **SCHED_FIFO** or **SCHED_RR**).

RETURNS Current scheduling policy (**SCHED_FIFO** or **SCHED_RR**), or -1 (**ERROR**) on error.

ERRNO **ESRCH**
- invalid task ID.

SEE ALSO **schedPxLib**

S

sched_rr_get_interval( )

NAME **sched_rr_get_interval( )** – get the current time slice (POSIX)

SYNOPSIS

```
int sched_rr_get_interval
    (
    pid_t              tid,      /* task ID */
    struct timespec * interval /* struct to store time slice */
    )
```

DESCRIPTION This routine sets *interval* to the current time slice period if round-robin scheduling is currently enabled.

RETURNS 0 (**OK**) if successful, -1 (**ERROR**) on error.

ERRNO

EINVAL
- round-robin scheduling is not currently enabled.

ESRCH
- invalid task ID.

SEE ALSO **schedPxLib**

sched_setparam( )

NAME **sched_setparam( )** – set a task's priority (POSIX)

SYNOPSIS

```
int sched_setparam
    (
    pid_t                          tid,   /* task ID */
    const struct sched_param * param /* scheduling parameter */
    )
```

DESCRIPTION This routine sets the priority of a specified task, *tid*. If *tid* is 0, it sets the priority of the calling task. Valid priority numbers are 0 through 255.

The *param* argument is a structure whose member **sched_priority** is the integer priority value. For example, the following program fragment sets the calling task's priority to 13 using POSIX interfaces:

```
#include "sched.h"
 ...
```

```
struct sched_param AppSchedPrio;
 ...
AppSchedPrio.sched_priority = 13;
if ( sched_setparam (0, &AppSchedPrio) != OK )
    {
    ... /* recovery attempt or abort message */
    }
 ...
```

NOTE: If the global variable **posixPriorityNumbering** is **FALSE**, the VxWorks native priority numbering scheme is used, in which higher priorities are indicated by smaller numbers. This is different than the priority numbering scheme specified by POSIX, in which higher priorities are indicated by larger numbers.

RETURNS 0 (**OK**) if successful, or -1 (**ERROR**) on error.

ERRNO

EINVAL
- scheduling priority is outside valid range.
ESRCH
- task ID is invalid.

SEE ALSO **schedPxLib**

sched_setscheduler( )

NAME **sched_setscheduler( )** – set scheduling policy and scheduling parameters (POSIX)

SYNOPSIS

```
int sched_setscheduler
    (
    pid_t                      tid,     /* task ID */
    int                        policy,  /* scheduling policy requested */
    const struct sched_param * param    /* scheduling parameters requested */
    )
```

DESCRIPTION This routine sets the scheduling policy and scheduling parameters for a specified task, *tid*. If *tid* is 0, it sets the scheduling policy and scheduling parameters for the calling task.

Because VxWorks does not set scheduling policies (*e.g.*, round-robin scheduling) on a task-by-task basis, setting a scheduling policy that conflicts with the current system policy simply fails and **errno** is set to **EINVAL**. If the requested scheduling policy is the same as the current system policy, then this routine acts just like **sched_setparam( )**.

S

NOTE: If the global variable **posixPriorityNumbering** is **FALSE**, the VxWorks native priority numbering scheme is used, in which higher priorities are indicated by smaller numbers. This is different than the priority numbering scheme specified by POSIX, in which higher priorities are indicated by larger numbers.

RETURNS
The previous scheduling policy (**SCHED_FIFO** or **SCHED_RR**), or -1 (**ERROR**) on error.

ERRNO
EINVAL
- scheduling priority is outside valid range, or it is impossible to set the specified scheduling policy.

ESRCH
- invalid task ID.

SEE ALSO
schedPxLib

sched_yield( )

NAME
sched_yield( ) – relinquish the CPU (POSIX)

SYNOPSIS
```
int sched_yield (void)
```

DESCRIPTION
This routine forces the running task to give up the CPU.

RETURNS
0 (**OK**) if successful, or -1 (**ERROR**) on error.

SEE ALSO
schedPxLib

scsi2IfInit( )

NAME
scsi2IfInit( ) – initialize the SCSI-2 interface to **scsiLib**

SYNOPSIS
```
void scsi2IfInit ()
```

DESCRIPTION
This routine initializes the SCSI-2 function interface by adding all the routines in **scsi2Lib** plus those in **scsiDirectLib** and **scsiCommonLib**. It is invoked by **usrConfig.c** if the macro **INCLUDE_SCSI2** is defined in **config.h**. The calling interface remains the same

between SCSI-1 and SCSI-2; this routine simply sets the calling interface function pointers to the SCSI-2 functions.

RETURNS N/A

SEE ALSO **scsi2Lib**

scsiAutoConfig( )

NAME **scsiAutoConfig( )** – configure all devices connected to a SCSI controller

SYNOPSIS

```
STATUS scsiAutoConfig
    (
    SCSI_CTRL * pScsiCtrl     /* ptr to SCSI controller info */
    )
```

DESCRIPTION This routine cycles through all valid SCSI bus IDs and logical unit numbers (LUNs), attempting a **scsiPhysDevCreate( )** with default parameters on each. All devices which support the INQUIRY command are configured. The **scsiShow( )** routine can be used to find the system table of SCSI physical devices attached to a specified SCSI controller. In addition, **scsiPhysDevIdGet( )** can be used programmatically to get a pointer to the **SCSI_PHYS_DEV** structure associated with the device at a specified SCSI bus ID and LUN.

RETURNS **OK**, or **ERROR** if *pScsiCtrl* and the global variable **pSysScsiCtrl**are both **NULL**.

SEE ALSO **scsiLib**

S

scsiBlkDevCreate( )

NAME **scsiBlkDevCreate( )** – define a logical partition on a SCSI block device

SYNOPSIS

```
BLK_DEV * scsiBlkDevCreate
    (
    SCSI_PHYS_DEV * pScsiPhysDev, /* ptr to SCSI physical device info */
    int             numBlocks,    /* number of blocks in block device */
    int             blockOffset   /* address of first block in volume */
    )
```

DESCRIPTION

This routine creates and initializes a **BLK_DEV** structure, which describes a logical partition on a SCSI physical-block device. A logical partition is an array of contiguously addressed blocks; it can be completely described by the number of blocks and the address of the first block in the partition. In normal configurations partitions do not overlap, although such a condition is not an error.

NOTE: If *numBlocks* is 0, the rest of device is used.

RETURNS

A pointer to the created **BLK_DEV**, or **NULL** if parameters exceed physical device boundaries, if the physical device is not a block device, or if memory is insufficient for the structures.

SEE ALSO

scsiLib

scsiBlkDevInit( )

NAME

scsiBlkDevInit( ) – initialize fields in a SCSI logical partition

SYNOPSIS

```
void scsiBlkDevInit
    (
    SCSI_BLK_DEV * pScsiBlkDev,  /* ptr to SCSI block dev. struct */
    int            blksPerTrack, /* blocks per track */
    int            nHeads        /* number of heads */
    )
```

DESCRIPTION

This routine specifies the disk-geometry parameters required by certain file systems (for example, dosFs). It is called after a **SCSI_BLK_DEV** structure is created with **scsiBlkDevCreate( )**, but before calling a file system initialization routine. It is generally required only for removable-media devices.

RETURNS

N/A

SEE ALSO

scsiLib

scsiBlkDevShow( )

NAME **scsiBlkDevShow( )** – show the **BLK_DEV** structures on a specified physical device

SYNOPSIS

```
void scsiBlkDevShow
    (
    SCSI_PHYS_DEV * pScsiPhysDev /* ptr to SCSI physical device info */
    )
```

DESCRIPTION This routine displays all of the **BLK_DEV** structures created on a specified physical device. This routine is called by **scsiShow( )** but may also be invoked directly, usually from the shell.

RETURNS N/A

SEE ALSO **scsiLib**, **scsiShow( )**

scsiBusReset( )

NAME **scsiBusReset( )** – pulse the reset signal on the SCSI bus

SYNOPSIS

```
STATUS scsiBusReset
    (
    SCSI_CTRL * pScsiCtrl     /* ptr to SCSI controller info */
    )
```

DESCRIPTION This routine calls a controller-specific routine to reset a specified controller's SCSI bus. If no controller is specified (*pScsiCtrl* is 0), the value in the global variable **pSysScsiCtrl** is used.

RETURNS **OK**, or **ERROR** if there is no controller or controller-specific routine.

SEE ALSO **scsiLib**

scsiCacheSnoopDisable( )

NAME

scsiCacheSnoopDisable( ) – inform SCSI that hardware snooping of caches is disabled

SYNOPSIS

```
void scsiCacheSnoopDisable
    (
    SCSI_CTRL * pScsiCtrl     /* pointer to a SCSI_CTRL structure */
    )
```

DESCRIPTION

This routine informs the SCSI library that hardware snooping is disabled and that **scsi2Lib** should execute any necessary cache coherency code. In order to make **scsi2Lib** aware that hardware snooping is disabled, this routine should be called after all SCSI-2 initializations, especially after **scsi2CtrlInit( )**.

RETURNS

N/A

SEE ALSO

scsi2Lib

scsiCacheSnoopEnable( )

NAME

scsiCacheSnoopEnable( ) – inform SCSI that hardware snooping of caches is enabled

SYNOPSIS

```
void scsiCacheSnoopEnable
    (
    SCSI_CTRL * pScsiCtrl     /* pointer to a SCSI_CTRL structure */
    )
```

DESCRIPTION

This routine informs the SCSI library that hardware snooping is enabled and that **scsi2Lib** need not execute any cache coherency code. In order to make **scsi2Lib** aware that hardware snooping is enabled, this routine should be called after all SCSI-2 initializations, especially after **scsi2CtrlInit( )**.

RETURNS

N/A

SEE ALSO

scsi2Lib

scsiCacheSynchronize( )

NAME

scsiCacheSynchronize( ) – synchronize the caches for data coherency

SYNOPSIS

```
void scsiCacheSynchronize
    (
    SCSI_THREAD *      pThread, /* ptr to thread info */
    SCSI_CACHE_ACTION action    /* cache action required */
    )
```

DESCRIPTION

This routine performs whatever cache action is necessary to ensure cache coherency with respect to the various buffers involved in a SCSI command. The process is as follows:

1. The buffers for command, identification, and write data, which are simply written to SCSI, are flushed before the command.
2. The status buffer, which is written and then read, is cleared (flushed and invalidated) before the command.
3. The data buffer for a read command, which is only read, is cleared before the command.

The data buffer for a read command is cleared before the command rather than invalidated after it because it may share dirty cache lines with data outside the read buffer. DMA drivers for older versions of the SCSI library have flushed the first and last bytes of the data buffer before the command. However, this approach is not sufficient with the enhanced SCSI library because the amount of data transferred into the buffer may not fill it, which would cause dirty cache lines which contain correct data for the un-filled part of the buffer to be lost when the buffer is invalidated after the command.

To optimize the performance of the driver in supporting different caching policies, the routine uses the **CACHE_USER_FLUSH** macro when flushing the cache. In the absence of a **CACHE_USER_CLEAR** macro, the following steps are taken:

1. If there is a non-**NULL** flush routine in the **cacheUserFuncs** structure, the cache is cleared.
2. If there is a non-**NULL** invalidate routine, the cache is invalidated.
3. Otherwise nothing is done; the cache is assumed to be coherent without any software intervention.

Finally, since flushing (clearing) cache line entries for a large data buffer can be time-consuming, if the data buffer is larger than a preset (run-time configurable) size, the entire cache is flushed.

RETURNS

N/A

SEE ALSO

scsi2Lib

S

scsiErase( )

NAME

scsiErase( ) – issue an ERASE command to a SCSI device

SYNOPSIS

```
STATUS scsiErase
    (
    SCSI_PHYS_DEV * pScsiPhysDev, /* ptr to SCSI physical device */
    BOOL            longErase     /* TRUE for entire tape erase */
    )
```

DESCRIPTION

This routine issues an ERASE command to a specified SCSI device.

RETURNS

OK, or **ERROR** if the command fails.

SEE ALSO

scsiSeqLib

scsiFormatUnit( )

NAME

scsiFormatUnit( ) – issue a **FORMAT_UNIT** command to a SCSI device

SYNOPSIS

```
STATUS scsiFormatUnit
    (
    SCSI_PHYS_DEV * pScsiPhysDev,  /* ptr to SCSI physical device */
    BOOL            cmpDefectList, /* whether defect list is complete */
    int             defListFormat, /* defect list format */
    int             vendorUnique,  /* vendor unique byte */
    int             interleave,    /* interleave factor */
    char *          buffer,        /* ptr to input data buffer */
    int             bufLength      /* length of buffer in bytes */
    )
```

DESCRIPTION

This routine issues a **FORMAT_UNIT** command to a specified SCSI device.

RETURNS

OK, or **ERROR** if the command fails.

SEE ALSO

scsiLib

scsiIdentMsgBuild( )

NAME **scsiIdentMsgBuild( )** – build an identification message

SYNOPSIS

```
int scsiIdentMsgBuild
    (
    UINT8 *         msg,
    SCSI_PHYS_DEV * pScsiPhysDev,
    SCSI_TAG_TYPE   tagType,
    UINT            tagNumber
    )
```

DESCRIPTION This routine builds an identification message in the caller's buffer, based on the specified physical device, tag type, and tag number.

If the target device does not support messages, there is no identification message to build.

Otherwise, the identification message consists of an IDENTIFY byte plus an optional QUEUE TAG message (two bytes), depending on the type of tag used.

NOTE: This function is not intended for use by application programs.

RETURNS The length of the resulting identification message in bytes or -1 for **ERROR**.

SEE ALSO **scsi2Lib**

scsiIdentMsgParse( )

NAME **scsiIdentMsgParse( )** – parse an identification message

SYNOPSIS

```
SCSI_IDENT_STATUS scsiIdentMsgParse
    (
    SCSI_CTRL *         pScsiCtrl,
    UINT8 *             msg,
    int                 msgLength,
    SCSI_PHYS_DEV * *   ppScsiPhysDev,
    SCSI_TAG *          pTagNum
    )
```

S

DESCRIPTION

This routine scans a (possibly incomplete) identification message, validating it in the process. If there is an IDENTIFY message, it identifies the corresponding physical device.

If the physical device is currently processing an untagged (ITL) nexus, identification is complete. Otherwise, the identification is complete only if there is a complete QUEUE TAG message.

If there is no physical device corresponding to the IDENTIFY message, or if the device is processing tagged (ITLQ) nexuses and the tag does not correspond to an active thread (it may have been aborted by a timeout, for example), then the identification sequence fails.

The caller's buffers for physical device and tag number (the results of the identification process) are always updated. This is required by the thread event handler (see **scsiMgrThreadEvent( )**.)

NOTE: This function is not intended for use by application programs.

RETURNS

The identification status (incomplete, complete, or rejected).

SEE ALSO

scsi2Lib

---

scsiInquiry( )

NAME

scsiInquiry( ) – issue an INQUIRY command to a SCSI device

SYNOPSIS

```
STATUS scsiInquiry
    (
    SCSI_PHYS_DEV * pScsiPhysDev, /* ptr to SCSI physical device */
    char *          buffer,       /* ptr to input data buffer */
    int             bufLength     /* length of buffer in bytes */
    )
```

DESCRIPTION

This routine issues an INQUIRY command to a specified SCSI device.

RETURNS

OK, or **ERROR** if the command fails.

SEE ALSO

scsiLib

scsiIoctl( )

NAME

scsiIoctl( ) – perform a device-specific I/O control function

SYNOPSIS

```
STATUS scsiIoctl
    (
    SCSI_PHYS_DEV * pScsiPhysDev, /* ptr to SCSI block device info */
    int             function,     /* function code */
    int             arg           /* argument to pass called function */
    )
```

DESCRIPTION

This routine performs a specified **ioctl** function using a specified SCSI block device.

RETURNS

The status of the request, or **ERROR** if the request is unsupported.

SEE ALSO

scsiLib

scsiLoadUnit( )

NAME

scsiLoadUnit( ) – issue a LOAD/UNLOAD command to a SCSI device

SYNOPSIS

```
STATUS scsiLoadUnit
    (
    SCSI_SEQ_DEV * pScsiSeqDev, /* ptr to SCSI physical device */
    BOOL           load,        /* TRUE=load, FALSE=unload */
    BOOL           reten,       /* TRUE=retention and unload */
    BOOL           eot          /* TRUE=end of tape and unload */
    )
```

DESCRIPTION

This routine issues a LOAD/UNLOAD command to a specified SCSI device.

RETURNS

OK, or **ERROR** if the command fails.

SEE ALSO

scsiSeqLib

scsiMgrBusReset( )

NAME **scsiMgrBusReset( )** – handle a controller-bus reset event

SYNOPSIS

```
void scsiMgrBusReset
    (
    SCSI_CTRL * pScsiCtrl     /* SCSI ctrlr on which bus reset */
    )
```

DESCRIPTION This routine resets in turn: each attached physical device, each target, and the controller-finite-state machine. In practice, this routine implements the SCSI hard reset option.

NOTE: This routine does not physically reset the SCSI bus; see **scsiBusReset( )**. This routine should not be called by application programs.

RETURNS N/A

SEE ALSO **scsiMgrLib**

scsiMgrCtrlEvent( )

NAME **scsiMgrCtrlEvent( )** – send an event to the SCSI controller state machine

SYNOPSIS

```
void scsiMgrCtrlEvent
    (
    SCSI_CTRL *     pScsiCtrl,
    SCSI_EVENT_TYPE eventType
    )
```

DESCRIPTION This routine is called by the thread driver whenever selection, re-selection, or disconnection occurs or when a thread is activated. It manages a simple finite-state machine for the SCSI controller.

NOTE: This function should not be called by application programs.

RETURNS N/A

SEE ALSO **scsiMgrLib**

scsiMgrEventNotify( )

NAME

scsiMgrEventNotify( ) – notify the SCSI manager of a SCSI (controller) event

SYNOPSIS

```
STATUS scsiMgrEventNotify
    (
    SCSI_CTRL *  pScsiCtrl,   /* pointer to SCSI controller structure */
    SCSI_EVENT * pEvent,      /* pointer to the SCSI event */
    int          eventSize    /* size of the event information */
    )
```

DESCRIPTION

This routine posts an event message on the appropriate SCSI manager queue, then notifies the SCSI manager that there is a message to be accepted.

NOTE: This routine should not be called by application programs.

No access serialization is required, because event messages are only posted by the SCSI controller ISR. See the reference entry for **scsiBusResetNotify( )**.

RETURNS

OK, or **ERROR** if the SCSI manager's event queue is full.

SEE ALSO

scsiMgrLib, **scsiBusResetNotify( )**

scsiMgrShow( )

S

NAME

scsiMgrShow( ) – show status information for the SCSI manager

SYNOPSIS

```
void scsiMgrShow
    (
    SCSI_CTRL * pScsiCtrl,      /* SCSI controller to use */
    BOOL        showPhysDevs,   /* TRUE => show phys dev details */
    BOOL        showThreads,    /* TRUE => show thread details */
    BOOL        showFreeThreads /* TRUE => show free thread IDs */
    )
```

DESCRIPTION

This routine shows the current state of the SCSI manager for the specified controller, including the total number of threads created and the number of threads currently free.

Optionally, this routine also shows details for all created physical devices on this controller and all threads for which SCSI requests are outstanding. It also shows the IDs of all free threads.

NOTE: The information displayed is volatile; this routine is best used when there is no activity on the SCSI bus. Threads allocated by a client but for which there are no outstanding SCSI requests are not shown.

RETURNS

N/A

SEE ALSO

scsiMgrLib

scsiMgrThreadEvent( )

NAME

scsiMgrThreadEvent( ) – send an event to the thread state machine

SYNOPSIS

```
void scsiMgrThreadEvent
    (
    SCSI_THREAD *          pThread,
    SCSI_THREAD_EVENT_TYPE eventType
    )
```

DESCRIPTION

This routine forwards an event to the thread's physical device. If the event is completion or deferral, it frees up the tag which was allocated when the thread was activated and either completes or defers the thread.

NOTE: This function should not be called by application programs.

The thread passed into this function does not have to be an active client thread (it may be an identification thread).

If the thread has no corresponding physical device, this routine does nothing. (This occasionally occurs if an unexpected disconnection or bus reset happens when an identification thread has not yet identified which physical device it corresponds to.

RETURNS

N/A

SEE ALSO

scsiMgrLib

scsiModeSelect( )

NAME **scsiModeSelect( )** – issue a **MODE_SELECT** command to a SCSI device

SYNOPSIS

```
STATUS scsiModeSelect
    (
    SCSI_PHYS_DEV * pScsiPhysDev, /* ptr to SCSI physical device */
    int           pageFormat,   /* value of the page format bit (0-1) */
    int           saveParams,   /* value of the save parameters bit (0-1) */
    char *        buffer,       /* ptr to output data buffer */
    int           bufLength     /* length of buffer in bytes */
    )
```

DESCRIPTION This routine issues a **MODE_SELECT** command to a specified SCSI device.

RETURNS **OK**, or **ERROR** if the command fails.

SEE ALSO **scsiLib**

scsiModeSense( )

NAME **scsiModeSense( )** – issue a **MODE_SENSE** command to a SCSI device

SYNOPSIS

```
STATUS scsiModeSense
    (
    SCSI_PHYS_DEV * pScsiPhysDev, /* ptr to SCSI physical device */
    int             pageControl,  /* value of the page control field (0-3) */
    int             pageCode,     /* value of the page code field (0-0x3f) */
    char *          buffer,       /* ptr to input data buffer */
    int             bufLength     /* length of buffer in bytes */
    )
```

DESCRIPTION This routine issues a **MODE_SENSE** command to a specified SCSI device.

RETURNS **OK**, or **ERROR** if the command fails.

SEE ALSO **scsiLib**

S

scsiMsgInComplete( )

NAME

scsiMsgInComplete( ) – handle a complete SCSI message received from the target

SYNOPSIS

```
STATUS scsiMsgInComplete
    (
    SCSI_CTRL *   pScsiCtrl,  /* ptr to SCSI controller info */
    SCSI_THREAD * pThread     /* ptr to thread info */
    )
```

DESCRIPTION

This routine parses the complete message and takes any necessary action, which may include setting up an outgoing message in reply. If the message is not understood, the routine rejects it and returns an **ERROR** status.

NOTE: This function is intended for use only by SCSI controller drivers.

RETURNS

OK, or **ERROR** if the message is not supported.

SEE ALSO

scsi2Lib

scsiMsgOutComplete( )

NAME

scsiMsgOutComplete( ) – perform post-processing after a SCSI message is sent

SYNOPSIS

```
STATUS scsiMsgOutComplete
    (
    SCSI_CTRL *   pScsiCtrl,  /* ptr to SCSI controller info */
    SCSI_THREAD * pThread     /* ptr to thread info */
    )
```

DESCRIPTION

This routine parses the complete message and takes any necessary action.

NOTE: This function is intended for use only by SCSI controller drivers.

RETURNS

OK, or **ERROR** if the message is not supported.

SEE ALSO

scsi2Lib

scsiMsgOutReject( )

NAME

scsiMsgOutReject( ) – perform post-processing when an outgoing message is rejected

SYNOPSIS

```
void scsiMsgOutReject
    (
    SCSI_CTRL *   pScsiCtrl,  /* ptr to SCSI controller info */
    SCSI_THREAD * pThread     /* ptr to thread info */
    )
```

DESCRIPTION

NOTE: This function is intended for use only by SCSI controller drivers.

RETURNS

OK, or **ERROR** if the message is not supported.

SEE ALSO

scsi2Lib

scsiPhysDevCreate( )

NAME

scsiPhysDevCreate( ) – create a SCSI physical device structure

SYNOPSIS

```
SCSI_PHYS_DEV * scsiPhysDevCreate
    (
    SCSI_CTRL * pScsiCtrl,      /* ptr to SCSI controller info */
    int         devBusId,       /* device's SCSI bus ID */
    int         devLUN,         /* device's logical unit number */
    int         reqSenseLength, /* length of REQUEST SENSE data dev returns */
    int         devType,        /* type of SCSI device */
    BOOL        removable,      /* whether medium is removable */
    int         numBlocks,      /* number of blocks on device */
    int         blockSize       /* size of a block in bytes */
    )
```

DESCRIPTION

This routine enables access to a SCSI device and must be the first routine invoked. It must be called once for each physical device on the SCSI bus.

If *reqSenseLength* is **NULL** (0), one or more **REQUEST_SENSE** commands are issued to the device to determine the number of bytes of sense data it typically returns. Note that if the device returns variable amounts of sense data depending on its state, you must consult the device manual to determine the maximum amount of sense data that can be returned.

S

If *devType* is NONE (-1), an INQUIRY command is issued to determine the device type; as an added benefit, it acquires the device's make and model number. The **scsiShow( )** routine displays this information. Common values of *devType* can be found in **scsiLib.h** or in the SCSI specification.

If *numBlocks* or *blockSize* are specified as **NULL** (0), a **READ_CAPACITY** command is issued to determine those values. This occurs only for device types that support **READ_CAPACITY**.

RETURNS A pointer to the created **SCSI_PHYS_DEV** structure, or **NULL** if the routine is unable to create the physical-device structure.

SEE ALSO **scsiLib**

scsiPhysDevDelete( )

NAME **scsiPhysDevDelete( )** – delete a SCSI physical-device structure

SYNOPSIS

```
STATUS scsiPhysDevDelete
    (
    SCSI_PHYS_DEV * pScsiPhysDev /* ptr to SCSI physical device info */
    )
```

DESCRIPTION This routine deletes a specified SCSI physical-device structure.

RETURNS **OK**, or **ERROR** if **pScsiPhysDev** is **NULL** or **SCSI_BLK_DEVs** have been created on the device.

SEE ALSO **scsiLib**

scsiPhysDevIdGet( )

NAME **scsiPhysDevIdGet( )** – return a pointer to a **SCSI_PHYS_DEV** structure

SYNOPSIS

```
SCSI_PHYS_DEV * scsiPhysDevIdGet
    (
    SCSI_CTRL * pScsiCtrl,    /* ptr to SCSI controller info */
    int         devBusId,     /* device's SCSI bus ID */
```

```
    int           devLUN        /* device's logical unit number */
    )
```

DESCRIPTION This routine returns a pointer to the **SCSI_PHYS_DEV** structure of the SCSI physical device located at a specified bus ID (*devBusId*) and logical unit number (*devLUN*) and attached to a specified SCSI controller (*pScsiCtrl*).

RETURNS A pointer to the specified **SCSI_PHYS_DEV** structure, or **NULL** if the structure does not exist.

SEE ALSO **scsiLib**

scsiPhysDevShow( )

NAME **scsiPhysDevShow( )** – show status information for a physical device

SYNOPSIS

```
void scsiPhysDevShow
    (
    SCSI_PHYS_DEV * pScsiPhysDev, /* physical device to be displayed */
    BOOL            showThreads,  /* show IDs of associated threads */
    BOOL            noHeader      /* do not print title line */
    )
```

DESCRIPTION This routine shows the state, the current nexus type, the current tag number, the number of tagged commands in progress, and the number of waiting and active threads for a SCSI physical device. Optionally, it shows the IDs of waiting and active threads, if any. This routine may be called at any time, but note that all of the information displayed is volatile.

RETURNS N/A

SEE ALSO **scsi2Lib**

S

scsiRdSecs( )

NAME **scsiRdSecs( )** – read sector(s) from a SCSI block device

SYNOPSIS

```
STATUS scsiRdSecs
    (
    SCSI_BLK_DEV * pScsiBlkDev, /* ptr to SCSI block device info */
    int            sector,      /* sector number to be read */
    int            numSecs,     /* total sectors to be read */
    char *         buffer       /* ptr to input data buffer */
    )
```

DESCRIPTION This routine reads the specified physical sector(s) from a specified physical device.

RETURNS **OK**, or **ERROR** if the sector(s) cannot be read.

SEE ALSO **scsiLib**

scsiRdTape( )

NAME **scsiRdTape( )** – read bytes or blocks from a SCSI tape device

SYNOPSIS

```
int scsiRdTape
    (
    SCSI_SEQ_DEV * pScsiSeqDev, /* ptr to SCSI sequential device info */
    UINT           count,       /* total bytes or blocks to be read */
    char *         buffer,      /* ptr to input data buffer */
    BOOL           fixedSize    /* if variable size blocks */
    )
```

DESCRIPTION This routine reads the specified number of bytes or blocks from a specified physical device. If the boolean *fixedSize* is true, then *numBytes* represents the number of blocks of size *blockSize*, defined in the **pScsiPhysDev** structure. If variable block sizes are used (*fixedSize* = **FALSE**), then *numBytes* represents the actual number of bytes to be read.

RETURNS Number of bytes or blocks actually read, 0 if EOF, or **ERROR**.

SEE ALSO **scsiSeqLib**

scsiReadCapacity( )

NAME **scsiReadCapacity( )** – issue a **READ_CAPACITY** command to a SCSI device

SYNOPSIS

```
STATUS scsiReadCapacity
    (
    SCSI_PHYS_DEV * pScsiPhysDev, /* ptr to SCSI physical device */
    int *           pLastLBA,     /* where to return last logical block */
                                  /* address */
    int *           pBlkLength    /* where to return block length */
    )
```

DESCRIPTION This routine issues a **READ_CAPACITY** command to a specified SCSI device.

RETURNS **OK**, or **ERROR** if the command fails.

SEE ALSO **scsiLib**

scsiRelease( )

NAME **scsiRelease( )** – issue a RELEASE command to a SCSI device

SYNOPSIS

```
STATUS scsiRelease
    (
    SCSI_PHYS_DEV * pScsiPhysDev /* ptr to SCSI physical device */
    )
```

DESCRIPTION This routine issues a RELEASE command to a specified SCSI device.

RETURNS **OK**, or **ERROR** if the command fails.

SEE ALSO **scsiDirectLib**

S

scsiReleaseUnit( )

NAME **scsiReleaseUnit( )** – issue a RELEASE UNIT command to a SCSI device

SYNOPSIS

```
STATUS scsiReleaseUnit
    (
    SCSI_SEQ_DEV * pScsiSeqDev /* ptr to SCSI sequential device */
    )
```

DESCRIPTION This routine issues a RELEASE UNIT command to a specified SCSI device.

RETURNS **OK**, or **ERROR** if the command fails.

SEE ALSO **scsiSeqLib**

scsiReqSense( )

NAME **scsiReqSense( )** – issue a **REQUEST_SENSE** command to a SCSI device and read results

SYNOPSIS

```
STATUS scsiReqSense
    (
    SCSI_PHYS_DEV * pScsiPhysDev, /* ptr to SCSI physical device */
    char *          buffer,       /* ptr to input data buffer */
    int             bufLength     /* length of buffer in bytes */
    )
```

DESCRIPTION This routine issues a **REQUEST_SENSE** command to a specified SCSI device and reads the results.

RETURNS **OK**, or **ERROR** if the command fails.

SEE ALSO **scsiLib**

scsiReserve( )

NAME **scsiReserve( )** – issue a RESERVE command to a SCSI device

SYNOPSIS

```
STATUS scsiReserve
    (
    SCSI_PHYS_DEV * pScsiPhysDev /* ptr to SCSI physical device */
    )
```

DESCRIPTION This routine issues a RESERVE command to a specified SCSI device.

RETURNS **OK**, or **ERROR** if the command fails.

SEE ALSO **scsiDirectLib**

scsiReserveUnit( )

NAME **scsiReserveUnit( )** – issue a RESERVE UNIT command to a SCSI device

SYNOPSIS

```
STATUS scsiReserveUnit
    (
    SCSI_SEQ_DEV * pScsiSeqDev /* ptr to SCSI sequential device */
    )
```

DESCRIPTION This routine issues a RESERVE UNIT command to a specified SCSI device.

RETURNS **OK**, or **ERROR** if the command fails.

SEE ALSO **scsiSeqLib**

scsiRewind( )

NAME
scsiRewind( ) – issue a REWIND command to a SCSI device

SYNOPSIS
```
STATUS scsiRewind
    (
    SCSI_SEQ_DEV * pScsiSeqDev /* ptr to SCSI Sequential device */
    )
```

DESCRIPTION
This routine issues a REWIND command to a specified SCSI device.

RETURNS
OK, or **ERROR** if the command fails.

SEE ALSO
scsiSeqLib

scsiSeqDevCreate( )

NAME
scsiSeqDevCreate( ) – create a SCSI sequential device

SYNOPSIS
```
SEQ_DEV *scsiSeqDevCreate
    (
    SCSI_PHYS_DEV * pScsiPhysDev /* ptr to SCSI physical device info */
    )
```

DESCRIPTION
This routine creates a SCSI sequential device and saves a pointer to this **SEQ_DEV** in the SCSI physical device. The following functions are initialized in this structure:

sd_seqRd	**scsiRdTape( )**
sd_seqWrt	**scsiWrtTape( )**
sd_ioctl	**scsiIoctl( )** (in **scsiLib**)
sd_seqWrtFileMarks	**scsiWrtFileMarks( )**
sd_statusChk	**scsiSeqStatusCheck( )**
sd_reset	(not used)
sd_rewind	**scsiRewind( )**
sd_reserve	**scsiReserve( )**
sd_release	**scsiRelease( )**
sd_readBlkLim	**scsiSeqReadBlockLimits( )**
sd_load	**scsiLoadUnit( )**
sd_space	**scsiSpace( )**
sd_erase	**scsiErase( )**

Only one **SEQ_DEV** per **SCSI_PHYS_DEV** is allowed, unlike **BLK_DEVs** where an entire list is maintained. Therefore, this routine can be called only once per creation of a sequential device.

RETURNS A pointer to the **SEQ_DEV** structure, or **NULL** if the command fails.

SEE ALSO **scsiSeqLib**

scsiSeqIoctl( )

NAME **scsiSeqIoctl( )** – perform an I/O control function for sequential access devices

SYNOPSIS

```
int scsiSeqIoctl
    (
    SCSI_SEQ_DEV * pScsiSeqDev, /* ptr to SCSI sequential device */
    int            function,    /* ioctl function code */
    int            arg          /* argument to pass to called function */
    )
```

DESCRIPTION This routine issues **scsiSeqLib** commands to perform sequential device-specific I/O control operations.

RETURNS **OK** or **ERROR**.

ERRNO **S_scsiLib_INVALID_BLOCK_SIZE**

SEE ALSO **scsiSeqLib**

scsiSeqReadBlockLimits( )

NAME **scsiSeqReadBlockLimits( )** – issue a **READ_BLOCK_LIMITS** command to a SCSI device

SYNOPSIS

```
STATUS scsiSeqReadBlockLimits
    (
    SCSI_SEQ_DEV * pScsiSeqDev,   /* ptr to SCSI sequential device */
    int *         pMaxBlockLength, /* where to return maximum block length */
    UINT16 *      pMinBlockLength  /* where to return minimum block length */
    )
```

S

DESCRIPTION This routine issues a **READ_BLOCK_LIMITS** command to a specified SCSI device.

RETURNS **OK**, or **ERROR** if the command fails.

SEE ALSO **scsiSeqLib**

scsiSeqStatusCheck( )

NAME **scsiSeqStatusCheck( )** – detect a change in media

SYNOPSIS

```
STATUS scsiSeqStatusCheck
    (
    SCSI_SEQ_DEV * pScsiSeqDev /* ptr to a sequential dev */
    )
```

DESCRIPTION This routine issues a **TEST_UNIT_READY** command to a SCSI device to detect a change in media. It is called by file systems before executing **open( )** or **creat( )**.

RETURNS **OK** or **ERROR**.

SEE ALSO **scsiSeqLib**

scsiShow( )

NAME **scsiShow( )** – list the physical devices attached to a SCSI controller

SYNOPSIS

```
STATUS scsiShow
    (
    SCSI_CTRL * pScsiCtrl     /* ptr to SCSI controller info */
    )
```

DESCRIPTION This routine displays the SCSI bus ID, logical unit number (LUN), vendor ID, product ID, firmware revision (rev.), device type, number of blocks, block size in bytes, and a pointer to the associated **SCSI_PHYS_DEV** structure for each physical SCSI device known to be attached to a specified SCSI controller.

NOTE: If *pScsiCtrl* is **NULL**, the value of the global variable **pSysScsiCtrl** is used, unless it is also **NULL**.

RETURNS
OK, or **ERROR** if both *pScsiCtrl* and **pSysScsiCtrl** are **NULL**.

SEE ALSO
scsiLib

scsiSpace( )

NAME
scsiSpace( ) – move the tape on a specified physical SCSI device

SYNOPSIS

```
STATUS scsiSpace
    (
    SCSI_SEQ_DEV * pScsiSeqDev, /* ptr to SCSI sequential device info */
    int            count,       /* count for space command */
    int            spaceCode    /* code for the type of space command */
    )
```

DESCRIPTION
This routine moves the tape on a specified SCSI physical device. There are two types of space code that are mandatory in SCSI; currently these are the only two supported:

Code	Description	Support
000	Blocks	Yes
001	File marks	Yes
010	Sequential file marks	No
011	End-of-data	No
100	Set marks	No
101	Sequential set marks	No

RETURNS
OK, or **ERROR** if an error is returned by the device.

ERRNO
S_scsiLib_ILLEGAL_REQUEST

SEE ALSO
scsiSeqLib

S

scsiStartStopUnit( )

NAME **scsiStartStopUnit( )** – issue a **START_STOP_UNIT** command to a SCSI device

SYNOPSIS

```
STATUS scsiStartStopUnit
    (
    SCSI_PHYS_DEV * pScsiPhysDev, /* ptr to SCSI physical device */
    BOOL            start         /* TRUE == start, FALSE == stop */
    )
```

DESCRIPTION This routine issues a **START_STOP_UNIT** command to a specified SCSI device.

RETURNS **OK**, or **ERROR** if the command fails.

SEE ALSO **scsiDirectLib**

scsiSyncXferNegotiate( )

NAME **scsiSyncXferNegotiate( )** – initiate or continue negotiating transfer parameters

SYNOPSIS

```
void scsiSyncXferNegotiate
    (
    SCSI_CTRL *          pScsiCtrl,   /* ptr to SCSI controller info */
    SCSI_TARGET *        pScsiTarget, /* ptr to SCSI target info */
    SCSI_SYNC_XFER_EVENT eventType    /* tells what has just happened */
    )
```

DESCRIPTION This routine manages negotiation by means of a finite-state machine which is driven by "significant events" such as incoming and outgoing messages. Each SCSI target has its own independent state machine.

NOTE: If the controller does not support synchronous transfer or if the target's maximum REQ/ACK offset is zero, attempts to initiate a round of negotiation are ignored.

This function is intended for use only by SCSI controller drivers.

RETURNS N/A

SEE ALSO **scsi2Lib**

scsiTapeModeSelect( )

NAME **scsiTapeModeSelect( )** – issue a **MODE_SELECT** command to a SCSI tape device

SYNOPSIS

```
STATUS scsiTapeModeSelect
    (
    SCSI_PHYS_DEV * pScsiPhysDev, /* ptr to SCSI physical device */
    int             pageFormat,   /* value of the page format bit (0-1) */
    int             saveParams,   /* value of the save parameters bit (0-1) */
    char *          buffer,       /* ptr to output data buffer */
    int             bufLength     /* length of buffer in bytes */
    )
```

DESCRIPTION This routine issues a **MODE_SELECT** command to a specified SCSI device.

RETURNS **OK**, or **ERROR** if the command fails.

SEE ALSO **scsiSeqLib**

scsiTapeModeSense( )

NAME **scsiTapeModeSense( )** – issue a **MODE_SENSE** command to a SCSI tape device

SYNOPSIS

```
STATUS scsiTapeModeSense
    (
    SCSI_PHYS_DEV * pScsiPhysDev, /* ptr to SCSI physical device */
    int             pageControl,  /* value of the page control field (0-3) */
    int             pageCode,     /* value of the page code field (0-0x3f) */
    char *          buffer,       /* ptr to input data buffer */
    int             bufLength     /* length of buffer in bytes */
    )
```

DESCRIPTION This routine issues a **MODE_SENSE** command to a specified SCSI tape device.

RETURNS **OK**, or **ERROR** if the command fails.

SEE ALSO **scsiSeqLib**

S

scsiTargetOptionsGet( )

NAME

scsiTargetOptionsGet( ) – get options for one or all SCSI targets

SYNOPSIS

```
STATUS scsiTargetOptionsGet
    (
    SCSI_CTRL *    pScsiCtrl, /* ptr to SCSI controller info */
    int            devBusId,  /* target to interrogate */
    SCSI_OPTIONS * pOptions   /* buffer to return options */
    )
```

DESCRIPTION

This routine copies the current options for the specified target into the caller's buffer.

RETURNS

OK, or **ERROR** if the bus ID is invalid.

SEE ALSO

scsi2Lib

scsiTargetOptionsSet( )

NAME

scsiTargetOptionsSet( ) – set options for one or all SCSI targets

SYNOPSIS

```
STATUS scsiTargetOptionsSet
    (
    SCSI_CTRL *    pScsiCtrl, /* ptr to SCSI controller info */
    int            devBusId,  /* target to affect, or all */
    SCSI_OPTIONS * pOptions,  /* buffer containing new options */
    UINT           which      /* which options to change */
    )
```

DESCRIPTION

This routine sets the options defined by the bit mask **which** for the specified target (or all targets if **devBusId** is **SCSI_SET_OPT_ALL_TARGETS**).

The bit mask **which** can be any combination of the following, bitwise OR'd together (corresponding fields in the **SCSI_OPTIONS** structure are shown in parentheses):

SCSI_SET_OPT_TIMEOUT	**selTimeOut**	select timeout period, microseconds
SCSI_SET_OPT_MESSAGES	**messages**	**FALSE** to disable SCSI messages
SCSI_SET_OPT_DISCONNECT	**disconnect**	**FALSE** to disable discon/recon
SCSI_SET_OPT_XFER_PARAMS	**maxOffset,**	max sync xfer offset, 0>async
	minPeriod	min sync xfer period, x 4 nsec.

SCSI_SET_OPT_TAG_PARAMS	**tagType,**	default tag type (**SCSI_TAG_***)
	maxTags	max cmd tags available
SCSI_SET_OPT_WIDE_PARAMS	**xferWidth**	data transfer width setting. xferWidth = 0 ; 8 bits wide xferWidth = 1 ; 16 bits wide

NOTE: This routine can be used after the target device has already been used; in this case, however, it is not possible to change the tag parameters. This routine must not be used while there is any SCSI activity on the specified target(s).

RETURNS **OK**, or **ERROR** if the bus ID or options are invalid.

SEE ALSO **scsi2Lib**

scsiTargetOptionsShow( )

NAME **scsiTargetOptionsShow( )** – display options for specified SCSI target

SYNOPSIS

```
STATUS scsiTargetOptionsShow
    (
    SCSI_CTRL * pScsiCtrl,    /* ptr to SCSI controller info */
    int         devBusId      /* target to interrogate */
    )
```

DESCRIPTION This routine displays the current target options for the specified target in the following format:

Target Options (id *scsi bus ID*):
  selection TimeOut: *timeout* nano secs
  messages allowed: **TRUE** or **FALSE**
  disconnect allowed: **TRUE** or **FALSE**
  REQ/ACK offset: *negotiated offset*
  transfer period: *negotiated period*
  transfer width: 8 or 16 bits maximum transfer rate: *peak transfer rate* MB/sec
  tag type: *tag type*
  maximum tags: *max tags*

RETURNS **OK**, or **ERROR** if the bus ID is invalid.

SEE ALSO **scsi2Lib**

S

scsiTestUnitRdy( )

NAME **scsiTestUnitRdy( )** – issue a **TEST_UNIT_READY** command to a SCSI device

SYNOPSIS

```
STATUS scsiTestUnitRdy
    (
    SCSI_PHYS_DEV * pScsiPhysDev /* ptr to SCSI physical device */
    )
```

DESCRIPTION This routine issues a **TEST_UNIT_READY** command to a specified SCSI device.

RETURNS **OK**, or **ERROR** if the command fails.

SEE ALSO **scsiLib**

scsiThreadInit( )

NAME **scsiThreadInit( )** – perform generic SCSI thread initialization

SYNOPSIS

```
STATUS scsiThreadInit
    (
    SCSI_THREAD * pThread
    )
```

DESCRIPTION This routine initializes the controller-independent parts of a thread structure, which are specific to the SCSI manager.

NOTE: This function should not be called by application programs. It is intended to be used by SCSI controller drivers.

RETURNS **OK**, or **ERROR** if the thread cannot be initialized.

SEE ALSO **scsi2Lib**

scsiWideXferNegotiate( )

NAME **scsiWideXferNegotiate( )** – initiate or continue negotiating wide parameters

SYNOPSIS

```
void scsiWideXferNegotiate
    (
    SCSI_CTRL *           pScsiCtrl,   /* ptr to SCSI controller info */
    SCSI_TARGET *         pScsiTarget, /* ptr to SCSI target info */
    SCSI_WIDE_XFER_EVENT eventType     /* tells what has just happened */
    )
```

DESCRIPTION This routine manages negotiation means of a finite-state machine which is driven by "significant events" such as incoming and outgoing messages. Each SCSI target has its own independent state machine.

NOTE: If the controller does not support wide transfers or the target's transfer width is zero, attempts to initiate a round of negotiation are ignored; this is because zero is the default narrow transfer.

This function is intended for use only by SCSI controller drivers.

RETURNS N/A

SEE ALSO **scsi2Lib**

scsiWrtFileMarks( )

NAME **scsiWrtFileMarks( )** – write file marks to a SCSI sequential device

SYNOPSIS

```
STATUS scsiWrtFileMarks
    (
    SCSI_SEQ_DEV * pScsiSeqDev, /* ptr to SCSI sequential device info */
    int            numMarks,    /* number of file marks to write */
    BOOL           shortMark    /* TRUE to write short file mark */
    )
```

DESCRIPTION This routine writes file marks to a specified physical device.

RETURNS **OK**, or **ERROR** if the file mark cannot be written.

SEE ALSO **scsiSeqLib**

S

scsiWrtSecs( )

NAME

scsiWrtSecs( ) – write sector(s) to a SCSI block device

SYNOPSIS

```
STATUS scsiWrtSecs
    (
    SCSI_BLK_DEV * pScsiBlkDev, /* ptr to SCSI block device info */
    int            sector,      /* sector number to be written */
    int            numSecs,     /* total sectors to be written */
    char *         buffer       /* ptr to input data buffer */
    )
```

DESCRIPTION

This routine writes the specified physical sector(s) to a specified physical device.

RETURNS

OK, or **ERROR** if the sector(s) cannot be written.

SEE ALSO

scsiLib

scsiWrtTape( )

NAME

scsiWrtTape( ) – write data to a SCSI tape device

SYNOPSIS

```
STATUS scsiWrtTape
    (
    SCSI_SEQ_DEV * pScsiSeqDev, /* ptr to SCSI sequential device info */
    int            numBytes,    /* total bytes or blocks to be written */
    char *         buffer,      /* ptr to input data buffer */
    BOOL           fixedSize    /* if variable size blocks */
    )
```

DESCRIPTION

This routine writes data to the current block on a specified physical device. If the boolean *fixedSize* is true, then *numBytes* represents the number of blocks of size *blockSize*, defined in the **pScsiPhysDev** structure. If variable block sizes are used (*fixedSize* = **FALSE**), then *numBytes* represents the actual number of bytes to be written. If *numBytes* is greater than the **maxBytesLimit** field defined in the **pScsiPhysDev** structure, then more than one SCSI transaction is used to transfer the data.

RETURNS

OK, or **ERROR** if the data cannot be written or zero bytes are written.

SEE ALSO

scsiSeqLib

select( )

NAME

select( ) – pend on a set of file descriptors

SYNOPSIS

```
int select
    (
    int               width,      /* number of bits to examine from 0 */
    fd_set *          pReadFds,   /* read fds */
    fd_set *          pWriteFds,  /* write fds */
    fd_set *          pExceptFds, /* exception fds (unsupported) */
    struct timeval *  pTimeOut    /* max time to wait, NULL = forever */
    )
```

DESCRIPTION

This routine permits a task to pend until one of a set of file descriptors becomes ready. Three parameters -- *pReadFds*, *pWriteFds*, and *pExceptFds* -- point to file descriptor sets in which each bit corresponds to a particular file descriptor. Bits set in the read file descriptor set (*pReadFds*) will cause **select( )** to pend until data is available on any of the corresponding file descriptors, while bits set in the write file descriptor set (*pWriteFds*) will cause **select( )** to pend until any of the corresponding file descriptors become writable. (The *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's Guide: I/O System* 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 **errno**s generated by this routine include:

S_selectLib_NO_SELECT_SUPPORT_IN_DRIVER
A driver associated with one or more *fd*s 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 **selectLib**, *VxWorks Programmer's Guide: I/O System*

selectInit( )

NAME **selectInit( )** – initialize the select facility

SYNOPSIS

```
void selectInit
    (
    int numFiles              /* maximum number of open files */
    )
```

DESCRIPTION This routine initializes the UNIX BSD 4.3 select facility. It should be called only once, and typically is called from the root task, **usrRoot( )**, in **usrConfig.c**. It installs a task create hook such that a select context is initialized for each task.

RETURNS N/A

SEE ALSO **selectLib**

selNodeAdd( )

NAME **selNodeAdd( )** – add a wake-up node to a **select( )** wake-up list

SYNOPSIS

```
STATUS selNodeAdd
    (
    SEL_WAKEUP_LIST * pWakeupList, /* list of tasks to wake up */
    SEL_WAKEUP_NODE * pWakeupNode  /* node to add to list */
    )
```

DESCRIPTION This routine adds a wake-up node to a device's wake-up list. It is typically called from a driver's **FIOSELECT** function.

RETURNS **OK**, or **ERROR** if memory is insufficient.

SEE ALSO **selectLib**

selNodeDelete( )

NAME **selNodeDelete( )** – find and delete a node from a **select( )** wake-up list

SYNOPSIS

```
STATUS selNodeDelete
    (
    SEL_WAKEUP_LIST * pWakeupList, /* list of tasks to wake up */
    SEL_WAKEUP_NODE * pWakeupNode  /* node to delete from list */
    )
```

DESCRIPTION This routine deletes a specified wake-up node from a specified wake-up list. Typically, it is called by a driver's **FIOUNSELECT** function.

RETURNS **OK**, or **ERROR** if the node is not found in the wake-up list.

SEE ALSO **selectLib**

selWakeup( )

NAME
selWakeup( ) – wake up a task pended in **select( )**

SYNOPSIS

```
void selWakeup
    (
    SEL_WAKEUP_NODE * pWakeupNode /* node to wake up */
    )
```

DESCRIPTION
This routine wakes up a task pended in **select( )**. Once a driver's **FIOSELECT** function installs a wake-up node in a device's wake-up list (using **selNodeAdd( )**) and checks to make sure the device is ready, this routine ensures that the **select( )** call does not pend.

RETURNS
N/A

SEE ALSO
selectLib

selWakeupAll( )

NAME
selWakeupAll( ) – wake up all tasks in a **select( )** wake-up list

SYNOPSIS

```
void selWakeupAll
    (
    SEL_WAKEUP_LIST * pWakeupList, /* list of tasks to wake up */
    SELECT_TYPE       type         /* readers (SELREAD) or writers (SELWRITE)
*/
    )
```

DESCRIPTION
This routine wakes up all tasks pended in **select( )** that are waiting for a device; it is called by a driver when the device becomes ready. The *type* parameter specifies the task to be awakened, either reader tasks (SELREAD) or writer tasks (SELWRITE).

RETURNS
N/A

SEE ALSO
selectLib

selWakeupListInit( )

NAME **selWakeupListInit( )** – initialize a **select( )** wake-up list

SYNOPSIS

```
void selWakeupListInit
    (
    SEL_WAKEUP_LIST * pWakeupList /* wake-up list to initialize */
    )
```

DESCRIPTION This routine should be called in a device's create routine to initialize the **SEL_WAKEUP_LIST** structure.

RETURNS N/A

SEE ALSO **selectLib**

selWakeupListLen( )

NAME **selWakeupListLen( )** – get the number of nodes in a **select( )** wake-up list

SYNOPSIS

```
int selWakeupListLen
    (
    SEL_WAKEUP_LIST * pWakeupList /* list of tasks to wake up */
    )
```

DESCRIPTION This routine returns the number of nodes in a specified **SEL_WAKEUP_LIST**. It can be used by a driver to determine if any tasks are currently pended in **select( )** on this device, and whether these tasks need to be activated with **selWakeupAll( )**.

RETURNS The number of nodes currently in a **select( )** wake-up list, or **ERROR**.

SEE ALSO **selectLib**

S

selWakeupListTerm( )

NAME **selWakeupListTerm( )** – terminate a **select( )** wake-up list

SYNOPSIS

```
void selWakeupListTerm
    (
    SEL_WAKEUP_LIST * pWakeupList /* wake-up list to terminate */
    )
```

DESCRIPTION This routine should be called in a device's terminate routine to terminate the **SEL_WAKEUP_LIST** structure.

RETURNS N/A

SEE ALSO **selectLib**

selWakeupType( )

NAME **selWakeupType( )** – get the type of a **select( )** wake-up node

SYNOPSIS

```
SELECT_TYPE selWakeupType
    (
    SEL_WAKEUP_NODE * pWakeupNode /* node to get type of */
    )
```

DESCRIPTION This routine returns the type of a specified **SEL_WAKEUP_NODE**. It is typically used in a device's **FIOSELECT** function to determine if the device is being selected for read or write operations.

RETURNS SELREAD (read operation) or SELWRITE (write operation).

SEE ALSO **selectLib**

semBCreate( )

NAME **semBCreate( )** – create and initialize a binary semaphore

SYNOPSIS

```
SEM_ID semBCreate
    (
    int         options,      /* semaphore options */
    SEM_B_STATE initialState  /* initial semaphore state */
    )
```

DESCRIPTION This routine allocates and initializes a binary semaphore. The semaphore is initialized to the *initialState* of either **SEM_FULL** (1) or **SEM_EMPTY** (0).

The *options* parameter specifies the queuing style for blocked tasks. Tasks can be queued on a priority basis or a first-in-first-out basis. These options are **SEM_Q_PRIORITY** (0x1) and **SEM_Q_FIFO** (0x0), respectively. That parameter also specifies if **semGive( )** should return **ERROR** when the semaphore fails to send events. This option is turned off by default; it is activated by doing a bitwise-OR of **SEM_EVENTSEND_ERR_NOTIFY** (0x10) with the queuing style of the semaphore.

RETURNS The semaphore ID, or **NULL** if memory cannot be allocated.

SEE ALSO **semBLib**

semBSmCreate( )

NAME **semBSmCreate( )** – create and initialize a shared memory binary semaphore (VxMP Opt.)

SYNOPSIS

```
SEM_ID semBSmCreate
    (
    int         options,      /* semaphore options */
    SEM_B_STATE initialState  /* initial semaphore state */
    )
```

DESCRIPTION This routine allocates and initializes a shared memory binary semaphore. The semaphore is initialized to an *initialState* of either **SEM_FULL** (available) or **SEM_EMPTY** (not available). The shared semaphore structure is allocated from the shared semaphore dedicated memory partition.

S

The semaphore ID returned by this routine can be used directly by the generic semaphore-handling routines in **semLib** -- **semGive( )**, **semTake( )**, and **semFlush( )** -- and the show routines, such as **show( )** and **semShow( )**.

The queuing style for blocked tasks is set by *options*; the only supported queuing style for shared memory semaphores is first-in-first-out, selected by **SEM_Q_FIFO**.

Before this routine can be called, the shared memory objects facility must be initialized (see **semSmLib**).

The maximum number of shared memory semaphores (binary plus counting) that can be created is **SM_OBJ_MAX_SEM**, a configurable parameter.

AVAILABILITY This routine is distributed as a component of the unbundled shared memory support option, VxMP.

RETURNS The semaphore ID, or **NULL** if memory cannot be allocated from the shared semaphore dedicated memory partition.

ERRNO **S_memLib_NOT_ENOUGH_MEMORY**, **S_semLib_INVALID_QUEUE_TYPE**, **S_semLib_INVALID_STATE**, **S_smObjLib_LOCK_TIMEOUT**

SEE ALSO **semSmLib**, **semLib**, **semBLib**, **smObjLib**, **semShow**, *VxWorks Programmer's Guide: Basic OS*

---

semCCreate( )

NAME **semCCreate( )** – create and initialize a counting semaphore

SYNOPSIS

```
SEM_ID semCCreate
    (
    int options,              /* semaphore option modes */
    int initialCount          /* initial count */
    )
```

DESCRIPTION This routine allocates and initializes a counting semaphore. The semaphore is initialized to the specified initial count.

The *options* parameter specifies the queuing style for blocked tasks. Tasks may be queued on a priority basis or a first-in-first-out basis. These options are **SEM_Q_PRIORITY** (0x1) and **SEM_Q_FIFO** (0x0), respectively. That parameter also specifies if **semGive( )** should return **ERROR** when the semaphore fails to send events. This option is turned off by default; it is activated by doing a bitwise-OR of **SEM_EVENTSEND_ERR_NOTIFY** (0x10) with the queuing style of the semaphore.

RETURNS The semaphore ID, or **NULL** if memory cannot be allocated.

SEE ALSO **semCLib**

semClear( )

NAME **semClear( )** – take a release 4.x semaphore, if the semaphore is available

SYNOPSIS

```
STATUS semClear
    (
    SEM_ID semId                /* semaphore ID to empty */
    )
```

DESCRIPTION This routine takes a VxWorks 4.x semaphore if it is available (full), otherwise no action is taken except to return **ERROR**. This routine never preempts the caller.

RETURNS **OK**, or **ERROR** if the semaphore is unavailable.

SEE ALSO **semOLib**

semCreate( )

NAME **semCreate( )** – create and initialize a release 4.x binary semaphore

SYNOPSIS

```
SEM_ID semCreate (void)
```

DESCRIPTION This routine allocates a VxWorks 4.x binary semaphore. The semaphore is initialized to empty. After initialization, it must be given before it can be taken.

RETURNS The semaphore ID, or **NULL** if memory cannot be allocated.

SEE ALSO **semOLib**, **semInit( )**

semCSmCreate( )

NAME

semCSmCreate( ) – create and initialize a shared memory counting semaphore (VxMP Opt.)

SYNOPSIS

```
SEM_ID semCSmCreate
    (
    int options,              /* semaphore options */
    int initialCount          /* initial semaphore count */
    )
```

DESCRIPTION

This routine allocates and initializes a shared memory counting semaphore. The initial count value of the semaphore is specified by *initialCount*.

The semaphore ID returned by this routine can be used directly by the generic semaphore-handling routines in **semLib** -- **semGive( )**, **semTake( )** and **semFlush( )** -- and the show routines, such as **show( )** and **semShow( )**.

The queuing style for blocked tasks is set by *options*; the only supported queuing style for shared memory semaphores is first-in-first-out, selected by **SEM_Q_FIFO**.

Before this routine can be called, the shared memory objects facility must be initialized (see **semSmLib**).

The maximum number of shared memory semaphores (binary plus counting) that can be created is **SM_OBJ_MAX_SEM**, a configurable parameter.

AVAILABILITY

This routine is distributed as a component of the unbundled shared memory support option, VxMP.

RETURNS

The semaphore ID, or **NULL** if memory cannot be allocated from the shared semaphore dedicated memory partition.

ERRNO

S_memLib_NOT_ENOUGH_MEMORY, **S_semLib_INVALID_QUEUE_TYPE**, **S_smObjLib_LOCK_TIMEOUT**

SEE ALSO

semSmLib, **semLib**, **semCLib**, **smObjLib**, **semShow**, *VxWorks Programmer's Guide: Basic OS*

semDelete( )

NAME **semDelete( )** – delete a semaphore

SYNOPSIS

```
STATUS semDelete
    (
    SEM_ID semId              /* semaphore ID to delete */
    )
```

DESCRIPTION This routine terminates and deallocates any memory associated with a specified semaphore. All tasks pending on the semaphore or pending for the reception of events meant to be sent from the semaphore will unblock and return **ERROR**.

WARNING: Take care when deleting semaphores, particularly those used for mutual exclusion, to avoid deleting a semaphore out from under a task that already has taken (owns) that semaphore. Applications should adopt the protocol of only deleting semaphores that the deleting task has successfully taken.

RETURNS **OK**, or **ERROR** if the semaphore ID is invalid.

ERRNO

S_intLib_NOT_ISR_CALLABLE
Routine cannot be called from ISR.

S_objLib_OBJ_ID_ERROR
Semaphore ID is invalid.

S_smObjLib_NO_OBJECT_DESTROY
Deleting a shared semaphore is not permitted

SEE ALSO **semLib**, **semBLib**, **semCLib**, **semMLib**, **semSmLib**

S

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. When the semaphore becomes available but no task is pending on it, the events specified will be sent to the task registered by this function. A task can overwrite its own registration without first invoking **semEvStop( )** or specifying the **ALLOW_OVERWRITE** option.

The *option* parameter is used for 3 user options:

EVENTS_SEND_ONCE (0x1)
tells the semaphore to send the events one time only. Specify if the events are to be sent only once or every time the semaphore is free until **semEvStop( )** is called.

EVENTS_ALLOW_OVERWRITE (0x2)
allows subsequent registrations to overwrite the current one. Specify if another task can register itself while the current task is still registered. If so, the current task registration is overwritten without any warning.

EVENTS_SEND_IF_FREE (0x4)
tells the registration process to send events if the semaphore is free. Specify if events are to be sent at the time of the registration in the case the semaphore is free.

If none of these options are to be used, the option

EVENTS_OPTIONS_NONE
has to be passed to the *options* parameter.

WARNING: This routine cannot be called from interrupt level.

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 chose to send events right away and that operation failed.

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.

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( )**

semFlush( )

NAME

semFlush( ) – unblock every task pended on a semaphore

SYNOPSIS

```
STATUS semFlush
    (
    SEM_ID semId              /* semaphore ID to unblock everyone for */
    )
```

DESCRIPTION

This routine atomically unblocks all tasks pended on a specified semaphore, *i.e.*, all tasks will be unblocked before any is allowed to run. The state of the underlying semaphore is unchanged. All pended tasks will enter the ready queue before having a chance to execute.

The flush operation is useful as a means of broadcast in synchronization applications. Its use is illegal for mutual-exclusion semaphores created with **semMCreate( )**.

S

RETURNS **OK**, or **ERROR** if the semaphore ID is invalid or the operation is not supported.

ERRNO **S_objLib_OBJ_ID_ERROR**

SEE ALSO **semLib**, **semBLib**, **semCLib**, **semMLib**, **semSmLib**

semGive( )

NAME **semGive( )** – give a semaphore

SYNOPSIS

```
STATUS semGive
    (
    SEM_ID semId              /* semaphore ID to give */
    )
```

DESCRIPTION This routine performs the give operation on a specified semaphore. Depending on the type of semaphore, the state of the semaphore and of the pending tasks may be affected. If no tasks are pending on the semaphore and a task has previously registered to receive events from the semaphore, these events are sent in the context of this call. This may result in the unpending of the task waiting for the events. If the semaphore fails to send events and if it was created using the **SEM_EVENTSEND_ERR_NOTIFY** option, **ERROR** is returned even though the give operation was successful. The behavior of **semGive( )** is discussed fully in the library description of the specific semaphore type being used.

RETURNS **OK** on success or **ERROR** otherwise

ERRNO

S_intLib_NOT_ISR_CALLABLE
Routine was called from an ISR for a mutex semaphore.

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**, **semSmLib**, **semEvStart( )**

semInfo( )

NAME

semInfo( ) – get a list of task IDs that are blocked on a semaphore

SYNOPSIS

```
int semInfo
    (
    SEM_ID semId,             /* semaphore ID to summarize */
    int    idList[],          /* array of task IDs to be filled in */
    int    maxTasks           /* max tasks idList can accommodate */
    )
```

DESCRIPTION

This routine reports the tasks blocked on a specified semaphore. Up to *maxTasks* task IDs are copied to the array specified by *idList*. The array is unordered.

WARNING: There is no guarantee that all listed tasks are still valid or that new tasks have not been blocked by the time **semInfo( )** returns.

RETURNS

The number of blocked tasks placed in *idList*.

SEE ALSO

semShow

semInit( )

NAME

semInit( ) – initialize a static binary semaphore

SYNOPSIS

```
STATUS semInit
    (
    SEMAPHORE * pSemaphore    /* 4.x semaphore to initialize */
    )
```

DESCRIPTION

This routine initializes static VxWorks 4.x semaphores. In some instances, a semaphore cannot be created with **semCreate( )** but is a static object.

RETURNS

OK, or **ERROR** if the semaphore cannot be initialized.

SEE ALSO

semOLib, **semCreate( )**

S

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.

RETURNS

The semaphore ID, or **NULL** if the semaphore cannot be created.

ERRNO

S_semLib_INVALID_OPTION
Invalid option was passed to **semMCreate( )**.

S_memLib_NOT_ENOUGH_MEMORY
Not enough memory available to create the semaphore.

SEE ALSO

semMLib, **semLib**, **semBLib**, **taskSafe( )**, **taskUnsafe( )**

semMGiveForce( )

NAME **semMGiveForce( )** – give a mutual-exclusion semaphore without restrictions

SYNOPSIS

```
STATUS semMGiveForce
    (
    SEM_ID semId              /* semaphore ID to give */
    )
```

DESCRIPTION This routine gives a mutual-exclusion semaphore, regardless of semaphore ownership. It is intended as a debugging aid only.

The routine is particularly useful when a task dies while holding some mutual-exclusion semaphore, because the semaphore can be resurrected. The routine will give the semaphore to the next task in the pend queue or make the semaphore full if no tasks are pending. In effect, execution will continue as if the task owning the semaphore had actually given the semaphore.

WARNING: This routine should be used only as a debugging aid, when the condition of the semaphore is known.

RETURNS **OK**, or **ERROR** if the semaphore ID is invalid.

SEE ALSO **semMLib**, **semGive( )**

semPxLibInit( )

NAME **semPxLibInit( )** – initialize POSIX semaphore support

SYNOPSIS

```
STATUS semPxLibInit (void)
```

DESCRIPTION This routine must be called before using POSIX semaphores.

RETURNS **OK**, or **ERROR** if there is an error installing the semaphore library.

SEE ALSO **semPxLib**

S

semPxShowInit( )

NAME **semPxShowInit( )** – initialize the POSIX semaphore show facility

SYNOPSIS

```
STATUS semPxShowInit (void)
```

DESCRIPTION This routine links the POSIX semaphore show routine into the VxWorks system. It is called automatically when the this show facility is configured into VxWorks using either of the following methods:

- If you use the configuration header files, define **INCLUDE_SHOW_ROUTINES** in **config.h**.
- If you use the Tornado project facility, select **INCLUDE_POSIX_SEM_SHOW**.

RETURNS **OK**, or **ERROR** if an error occurs installing the file pointer show routine.

SEE ALSO **semPxShow**

semShow( )

NAME **semShow( )** – show information about a semaphore

SYNOPSIS

```
STATUS semShow
    (
    SEM_ID semId,              /* semaphore to display */
    int    level               /* 0 = summary, 1 = details */
    )
```

DESCRIPTION This routine displays the state and optionally the pended tasks of a semaphore.

A summary of the state of the semaphore is displayed as follows:

```
Semaphore Id        : 0x585f2
Semaphore Type      : BINARY
Task Queuing        : PRIORITY
Pended Tasks        : 1
State               : EMPTY {Count if COUNTING, Owner if MUTEX}
Options             : 0x1       SEM_Q_PRIORITY
VxWorks Events
--------------
Registered Task     : 0x594f0 (t1)
Event(s) to Send    : 0x1
```

```
Options              : 0x7      EVENTS_SEND_ONCE
                                EVENTS_ALLOW_OVERWRITE
                                EVENTS_SEND_IF_FREE
```

If *level* is 1, then more detailed information will be displayed. If tasks are blocked on the queue, they are displayed in the order in which they will unblock, as follows:

```
Pended Tasks
------------
   NAME      TID    PRI DELAY
---------- -------- --- -----
tExcTask    3fd678   0    21
tLogTask    3f8ac0   0   611
```

RETURNS **OK** or **ERROR**.

SEE ALSO **semShow**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

---

semShowInit( )

NAME **semShowInit( )** – initialize the semaphore show facility

SYNOPSIS `void semShowInit (void)`

DESCRIPTION This routine links the semaphore show facility into the VxWorks system. It is called automatically when the semaphore show facility is configured into VxWorks using either of the following methods:

– If you use the configuration header files, define **INCLUDE_SHOW_ROUTINES** in **config.h**.

– If you use the Tornado project facility, select **INCLUDE_SEM_SHOW**.

RETURNS N/A

SEE ALSO **semShow**

semTake( )

NAME **semTake( )** – take a semaphore

SYNOPSIS

```
STATUS semTake
    (
    SEM_ID semId,              /* semaphore ID to take */
    int    timeout             /* timeout in ticks */
    )
```

DESCRIPTION This routine performs the take operation on a specified semaphore. Depending on the type of semaphore, the state of the semaphore and the calling task may be affected. The behavior of **semTake( )** is discussed fully in the library description of the specific semaphore type being used.

A timeout in ticks may be specified. If a task times out, **semTake( )** will return **ERROR**. Timeouts of **WAIT_FOREVER** (-1) and **NO_WAIT** (0) indicate to wait indefinitely or not to wait at all.

When **semTake( )** returns due to timeout, it sets the **errno** to **S_objLib_OBJ_TIMEOUT** (defined in **objLib.h**).

The **semTake( )** routine is not callable from interrupt service routines.

RETURNS **OK**, or **ERROR** if the semaphore ID is invalid or the task timed out.

ERRNO **S_intLib_NOT_ISR_CALLABLE**, **S_objLib_OBJ_ID_ERROR**, **S_objLib_OBJ_UNAVAILABLE**

SEE ALSO **semLib**, **semBLib**, **semCLib**, **semMLib**, **semSmLib**

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*. Do not call this routine with an unnamed semaphore (*i.e.*, one created by

sem_init( )); the effects are undefined. The **sem_close( )** call deallocates any system resources allocated by the system for use by this task for this semaphore.

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, the semaphore vanishes when the last task closes it.

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.

RETURNS

0 (**OK**), or -1 (**ERROR**) if unsuccessful.

ERRNO

EINVAL
- invalid semaphore descriptor.

SEE ALSO

semPxLib, **sem_unlink( )**, **sem_open( )**, **sem_init( )**

sem_destroy( )

NAME

sem_destroy( ) – destroy an unnamed semaphore (POSIX)

SYNOPSIS

```
int sem_destroy
    (
    sem_t * sem               /* semaphore descriptor */
    )
```

DESCRIPTION

This routine is used to destroy the unnamed semaphore indicated by *sem*.

The **sem_destroy( )** call can only destroy a semaphore created by **sem_init( )**. Calling **sem_destroy( )** with a named semaphore will cause a **EINVAL** error. Subsequent use of the *sem* semaphore will cause an **EINVAL** error in the calling function.

If one or more tasks is blocked on the semaphore, the semaphore is not destroyed.

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.

S

ERRNO
EINVAL
- invalid semaphore descriptor.
EBUSY
- one or more tasks is blocked on the semaphore.

SEE ALSO
semPxLib, **sem_init( )**

sem_getvalue( )

NAME
sem_getvalue( ) – get the value of a semaphore (POSIX)

SYNOPSIS

```
int sem_getvalue
    (
    sem_t * sem,              /* semaphore descriptor */
    int *   sval              /* buffer by which the value is returned */
    )
```

DESCRIPTION
This routine updates the location referenced by the *sval* argument to have the value of the semaphore referenced by *sem* without affecting the state of the semaphore. The updated value represents an actual semaphore value that occurred at some unspecified time during the call, but may not be the actual value of the semaphore when it is returned to the calling task.

If *sem* is locked, the value returned by **sem_getvalue( )** will either be zero or a negative number whose absolute value represents the number of tasks waiting for the semaphore at some unspecified time during the call.

RETURNS
0 (**OK**), or -1 (**ERROR**) if unsuccessful.

ERRNO
EINVAL
- invalid semaphore descriptor.

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,      /* process sharing */
    unsigned int value         /* semaphore initialization value */
    )
```

DESCRIPTION

This routine is used to initialize the unnamed semaphore *sem*. The value of the initialized semaphore is *value*. Following a successful call to **sem_init( )** the semaphore may be used in subsequent calls to **sem_wait( )**, **sem_trywait( )**, and **sem_post( )**. This semaphore remains usable until the semaphore is destroyed.

The *pshared* parameter currently has no effect.

Only *sem* itself may be used for synchronization.

RETURNS

0 (**OK**), or -1 (**ERROR**) if unsuccessful.

ERRNO

EINVAL
- *value* exceeds **SEM_VALUE_MAX**.
ENOSPC
- unable to initialize semaphore due to resource constraints.

SEE ALSO

semPxLib, **sem_wait( )**, **sem_trywait( )**, **sem_post( )**

sem_open( )

NAME

sem_open( ) – initialize/open a named semaphore (POSIX)

SYNOPSIS

```
sem_t * sem_open
    (
    const char * name,         /* semaphore name */
    int          oflag,        /* semaphore creation flags */
                 ...           /* extra optional parameters */
    )
```

DESCRIPTION

This routine establishes a connection between a named semaphore and a task. Following a call to **sem_open( )** with a semaphore name *name*, the task may reference the semaphore associated with *name* using the address returned by this call. This semaphore may be used in subsequent calls to **sem_wait( )**, **sem_trywait( )**, and **sem_post( )**. The semaphore remains usable until the semaphore is closed by a successful call to **sem_close( )**.

The *oflag* argument controls whether the semaphore is created or merely accessed by the call to **sem_open( )**. The following flag bits may be set in *oflag*:

O_CREAT
: Use this flag to create a semaphore if it does not already exist. If **O_CREAT** is set and the semaphore already exists, **O_CREAT** has no effect except as noted below under **O_EXCL**. Otherwise, **sem_open( )** creates a semaphore. **O_CREAT** requires a third and fourth argument: *mode*, which is of type **mode_t**, and *value*, which is of type unsigned int. *mode* has no effect in this implementation. The semaphore is created with an initial value of *value*. Valid initial values for semaphores must be less than or equal to **SEM_VALUE_MAX**.

O_EXCL
: If **O_EXCL** and **O_CREAT** are set, **sem_open( )** will fail if the semaphore name exists. If **O_EXCL** is set and **O_CREAT** is not set, the named semaphore is not created.

To determine whether a named semaphore already exists in the system, call **sem_open( )** with the flags **O_CREAT | O_EXCL**. If the **sem_open( )** call fails, the semaphore exists.

If a task makes multiple calls to **sem_open( )** with the same value for *name*, then the same semaphore address is returned for each such call, provided that there have been no calls to **sem_unlink( )** for this semaphore.

References to copies of the semaphore will produce undefined results.

NOTE

The current implementation has the following limitations:

– A semaphore cannot be closed with calls to **_exit( )** or **exec( )**.

– A semaphore cannot be implemented as a file.

– Semaphore names will not appear in the file system.

RETURNS

A pointer to **sem_t**, or -1 (**ERROR**) if unsuccessful.

ERRNO

EEXIST
- **O_CREAT | 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_post( )

NAME **sem_post( )** – unlock (give) a semaphore (POSIX)

SYNOPSIS

```
int sem_post
    (
    sem_t * sem               /* semaphore descriptor */
    )
```

DESCRIPTION This routine unlocks the semaphore referenced by *sem* by performing the semaphore unlock operation on that semaphore.

If the semaphore value resulting from the operation is positive, then no tasks were blocked waiting for the semaphore to become unlocked; the semaphore value is simply incremented.

If the value of the semaphore resulting from this semaphore is zero, then one of the tasks blocked waiting for the semaphore will return successfully from its call to **sem_wait( )**.

NOTE: The **_POSIX_PRIORITY_SCHEDULING** functionality is not yet supported.

Note that the POSIX terms *unlock* and *post* correspond to the term *give* used in other VxWorks semaphore documentation.

RETURNS 0 (**OK**), or -1 (**ERROR**) if unsuccessful.

ERRNO **EINVAL**
- invalid semaphore descriptor.

SEE ALSO **semPxLib**, **sem_wait( )**, **sem_trywait( )**

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 return, the state of the semaphore is always locked (either as a result of this call or by a previous **sem_wait( )** or **sem_trywait( )**). The semaphore will remain locked until **sem_post( )** is executed and returns successfully.

Deadlock detection is not implemented.

Note that the POSIX term *lock* corresponds to the term *take* used in other VxWorks semaphore documentation.

RETURNS

0 (**OK**), or -1 (**ERROR**) if unsuccessful.

ERRNO

EAGAIN
- semaphore is already locked.

EINVAL
- invalid semaphore descriptor.

SEE ALSO

semPxLib, **sem_wait( )**, **sem_post( )**

sem_unlink( )

NAME

sem_unlink( ) – remove a named semaphore (POSIX)

SYNOPSIS

```
int sem_unlink
    (
    const char * name          /* semaphore name */
    )
```

DESCRIPTION This routine removes the string *name* from the semaphore name table, and marks the corresponding semaphore for destruction. An unlinked semaphore is destroyed when the last task closes it with **sem_close( )**. After a particular name is removed from the table, 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
- semaphore name too long.

ENOENT
- named semaphore does not exist.

SEE ALSO **semPxLib**, **sem_open( )**, **sem_close( )**

---

sem_wait( )

NAME **sem_wait( )** – lock (take) a semaphore, blocking if not available (POSIX)

SYNOPSIS

```
int sem_wait
    (
    sem_t * sem              /* semaphore descriptor */
    )
```

DESCRIPTION This routine locks the semaphore referenced by *sem* by performing the semaphore lock operation on that semaphore. If the semaphore value is currently zero, the calling task will not return from the call to **sem_wait( )** until it either locks the semaphore or the call is interrupted by a signal.

On return, the state of the semaphore is locked and will remain locked until **sem_post( )** is executed and returns successfully.

Deadlock detection is not implemented.

Note that the POSIX term *lock* corresponds to the term *take* used in other VxWorks documentation regarding semaphores.

RETURNS 0 (**OK**), or -1 (**ERROR**) if unsuccessful.

ERRNO

EINVAL
- invalid semaphore descriptor, or semaphore destroyed while task waiting.

SEE ALSO **semPxLib**, **sem_trywait( )**, **sem_post( )**

S

send( )

NAME

send( ) – send data to a socket

SYNOPSIS

```
int send
    (
    int          s,          /* socket to send to */
    const char * buf,        /* pointer to buffer to transmit */
    int          bufLen,     /* length of buffer */
    int          flags       /* flags to underlying protocols */
    )
```

DESCRIPTION

This routine transmits data to a previously established connection-based (stream) socket.

The maximum length of *buf* is subject to the limits on TCP buffer size; see the discussion of **SO_SNDBUF** in the **setsockopt( )** manual entry.

You may OR the following values into the *flags* parameter with this operation:

MSG_OOB (0x1)
Out-of-band data.

MSG_DONTROUTE (0x4)
Send without using routing tables.

RETURNS

The number of bytes sent, or **ERROR** if the call fails.

SEE ALSO

sockLib, **setsockopt( )**, **sendmsg( )**

sendAdvert( )

NAME

sendAdvert( ) – send an advertisement to one location

SYNOPSIS

```
void sendAdvert
    (
    int            index,
    struct in_addr dstAddr
    )
```

DESCRIPTION This routine sends a router advertisement using the data stored for its' corresponding interface. Only the primary network address of the interface is used as the advertised router address.

RETURNS N/A

SEE ALSO **rdiscLib**

sendAdvertAll( )

NAME **sendAdvertAll( )** – send an advertisement to all active locations

SYNOPSIS

```
void sendAdvertAll (void)
```

DESCRIPTION This routine sends a router advertisement using the data stored for each corresponding interface. Only the primary network address of the interface is used as the advertised router address.

RETURNS N/A

SEE ALSO **rdiscLib**

sendmsg( )

NAME **sendmsg( )** – send a message to a socket

SYNOPSIS

```
int sendmsg
    (
    int              sd,       /* socket to send to */
    struct msghdr *  mp,       /* scatter-gather message header */
    int              flags     /* flags to underlying protocols */
    )
```

DESCRIPTION This routine sends a message to a datagram socket. It may be used in place of **sendto( )** to decrease the overhead of reconstructing the message-header structure (**msghdr**) for each message.

For BSD 4.4 sockets a copy of the *mp*>msg_iov array will be made. This requires a cluster from the network stack system pool of size *mp*>msg_iovlen * sizeof (struct iovec) or 8 bytes.

RETURNS The number of bytes sent, or **ERROR** if the call fails.

SEE ALSO **sockLib**, **sendto( )**

sendto( )

NAME **sendto( )** – send a message to a socket

SYNOPSIS

```
int sendto
    (
    int               s,      /* socket to send data to */
    caddr_t           buf,    /* pointer to data buffer */
    int               bufLen, /* length of buffer */
    int               flags,  /* flags to underlying protocols */
    struct sockaddr * to,     /* recipient's address */
    int               tolen   /* length of to sockaddr */
    )
```

DESCRIPTION This routine sends a message to the datagram socket named by *to*. The socket *s* is received by the receiver as the sending socket.

The maximum length of *buf* is subject to the limits on UDP buffer size. See the discussion of **SO_SNDBUF** in the **setsockopt( )** manual entry.

You can OR the following values into the *flags* parameter with this operation:

MSG_OOB (0x1)
  Out-of-band data.

MSG_DONTROUTE (0x4)
  Send without using routing tables.

RETURNS The number of bytes sent, or **ERROR** if the call fails.

SEE ALSO **sockLib**, **setsockopt( )**

set_new_handler( )

NAME **set_new_handler( )** – set **new_handler** to user-defined function (C++)

SYNOPSIS

```
extern void (*set_new_handler (void(* pNewNewHandler)()))  ()
```

DESCRIPTION This function is used to define the function that will be called when operator new cannot allocate memory.

The **new_handler** acts for all threads in the system; you cannot set a different handler for different tasks.

RETURNS A pointer to the previous value of **new_handler**.

INCLUDE FILES new

SEE ALSO **cplusLib**

set_terminate( )

NAME **set_terminate( )** – set terminate to user-defined function (C++)

SYNOPSIS

```
extern void (*set_terminate (void(* terminate_handler)()))  ()
```

DESCRIPTION This function is used to define the **terminate_handler** which will be called when an uncaught exception is raised.

The **terminate_handler** acts for all threads in the system; you cannot set a different handler for different tasks.

RETURNS The previous **terminate_handler**.

INCLUDE FILES exception

SEE ALSO **cplusLib**

setbuf( )

NAME **setbuf( )** – specify the buffering for a stream (ANSI)

SYNOPSIS

```
void setbuf
    (
    FILE * fp,                  /* stream to set buffering for */
    char * buf                  /* buffer to use */
    )
```

DESCRIPTION Except that it returns no value, this routine is equivalent to **setvbuf( )** invoked with the *mode* **_IOFBF** (full buffering) and *size* BUFSIZ, or (if *buf* is a null pointer), with the *mode* **_IONBF** (no buffering).

INCLUDE FILES **stdio.h**

RETURNS N/A

SEE ALSO **ansiStdio**, **setvbuf( )**

setbuffer( )

NAME **setbuffer( )** – specify buffering for a stream

SYNOPSIS

```
void setbuffer
    (
    FILE * fp,                  /* stream to set buffering for */
    char * buf,                 /* buffer to use */
    int    size                 /* buffer size */
    )
```

DESCRIPTION This routine specifies a buffer *buf* to be used for a stream in place of the automatically allocated buffer. If *buf* is **NULL**, the stream is unbuffered. This routine should be called only after the stream has been associated with an open file and before any other operation is performed on the stream.

This routine is provided for compatibility with earlier VxWorks releases.

INCLUDE FILES **stdio.h**

RETURNS N/A

SEE ALSO **ansiStdio**, **setvbuf( )**

sethostname( )

NAME **sethostname( )** – set the symbolic name of this machine

SYNOPSIS

```
int sethostname
    (
    char * name,              /* machine name */
    int    nameLen            /* length of name */
    )
```

DESCRIPTION This routine sets the target machine's symbolic name, which can be used for identification.

RETURNS **OK** or **ERROR**.

SEE ALSO **hostLib**

setjmp( )

NAME **setjmp( )** – save the calling environment in a **jmp_buf** argument (ANSI)

SYNOPSIS

```
int setjmp
    (
    jmp_buf env
    )
```

DESCRIPTION This routine saves the calling environment in *env*, in order to permit a **longjmp( )** call to restore that environment (thus performing a non-local goto).

Constraints on Calling Environment

The **setjmp( )** routine may only be used in the following contexts:

– as the entire controlling expression of a selection or iteration statement;

– as one operand of a relational or equality operator, in the controlling expression of a selection or iteration statement;

S

– as the operand of a single-argument **!** operator, in the controlling expression of a selection or iteration statement; or

– as a complete C statement containing nothing other than the **setjmp( )** call (though the result may be cast to **void**)

RETURNS From a direct invocation, **setjmp( )** returns zero. From a call to **longjmp( )**, it returns a non-zero value specified as an argument to **longjmp( )**.

SEE ALSO **ansiSetjmp**, **longjmp( )**

setlinebuf( )

NAME **setlinebuf( )** – set line buffering for standard output or standard error

SYNOPSIS

```
int setlinebuf
    (
    FILE * fp                    /* stream - stdout or stderr */
    )
```

DESCRIPTION This routine changes **stdout** or **stderr** streams from block-buffered or unbuffered to line-buffered. Unlike **setbuf( )**, **setbuffer( )**, or **setvbuf( )**, it can be used at any time the stream is active.

A stream can be changed from unbuffered or line-buffered to fully buffered using **freopen( )**. A stream can be changed from fully buffered or line-buffered to unbuffered using **freopen( )** followed by **setbuf( )** with a buffer argument of **NULL**.

This routine is provided for compatibility with earlier VxWorks releases.

INCLUDE **stdio.h**

RETURNS **OK**, or **ERROR** if *fp* is not a valid stream.

SEE ALSO **ansiStdio**

setlocale( )

NAME **setlocale( )** – set the appropriate locale (ANSI)

SYNOPSIS

```
char *setlocale
    (
    int          category,    /* category to change */
    const char * localeName   /* locale name */
    )
```

DESCRIPTION This function is included for ANSI compatibility. Only the default is implemented. At program start-up, the equivalent of the following is executed:

```
    setlocale (LC_ALL, "C");
```

This specifies the program's entire locale and the minimal environment for C translation.

INCLUDE FILES **locale.h**, **string.h**, **stdlib.h**

RETURNS A pointer to the string "C".

SEE ALSO **ansiLocale**

setsockopt( )

NAME **setsockopt( )** – set socket options

SYNOPSIS

```
STATUS setsockopt
    (
    int    s,                 /* target socket */
    int    level,             /* protocol level of option */
    int    optname,           /* option name */
    char * optval,            /* pointer to option value */
    int    optlen             /* option length */
    )
```

DESCRIPTION This routine sets the options associated with a socket. To manipulate options at the "socket" level, *level* should be **SOL_SOCKET**. Any other levels should use the appropriate protocol number.

S

OPTIONS FOR STREAM SOCKETS

The following sections discuss the socket options available for stream (TCP) sockets.

SO_KEEPALIVE -- Detecting a Dead Connection

Specify the **SO_KEEPALIVE** option to make the transport protocol (TCP) initiate a timer to detect a dead connection:

```
setsockopt (sock, SOL_SOCKET, SO_KEEPALIVE, &optval, sizeof (optval));
```

This prevents an application from hanging on an invalid connection. The value at *optval* for this option is an integer (type **int**), either 1 (on) or 0 (off).

The integrity of a connection is verified by transmitting zero-length TCP segments triggered by a timer, to force a response from a peer node. If the peer does not respond after repeated transmissions of the KEEPALIVE segments, the connection is dropped, all protocol data structures are reclaimed, and processes sleeping on the connection are awakened with an **ETIMEDOUT** error.

The **ETIMEDOUT** timeout can happen in two ways. If the connection is not yet established, the KEEPALIVE timer expires after idling for **TCPTV_KEEP_INIT**. If the connection is established, the KEEPALIVE timer starts up when there is no traffic for **TCPTV_KEEP_IDLE**. If no response is received from the peer after sending the KEEPALIVE segment **TCPTV_KEEPCNT** times with interval **TCPTV_KEEPINTVL**, TCP assumes that the connection is invalid. The **TCPTV_KEEP_INIT**, **TCPTV_KEEP_IDLE**, **TCPTV_KEEPCNT**, and **TCPTV_KEEPINTVL** parameters are defined in the file **target/h/netinet/tcp_timer.h**.

SO_LINGER -- Closing a Connection

Specify the **SO_LINGER** option to determine whether TCP should perform a “graceful” close:

```
setsockopt (sock, SOL_SOCKET, SO_LINGER, &optval, sizeof (optval));
```

To achieve a “graceful” close in response to the shutdown of a connection, TCP puts itself through an elaborate set of state transitions. The goal is to assure that all the unacknowledged data in the transmission channel are acknowledged, and that the peer is shut down properly.

The value at *optval* indicates the amount of time to linger if there is unacknowledged data, using **struct linger** in **target/h/sys/socket.h**. The **linger** structure has two members: **l_onoff** and **l_linger**. **l_onoff** can be set to 1 to turn on the **SO_LINGER** option, or set to 0 to turn off the **SO_LINGER** option. **l_linger** indicates the amount of time to linger. If **l_onoff** is turned on and **l_linger** is set to 0, a default value **TCP_LINGERTIME** (specified in **netinet/tcp_timer.h**) is used for incoming connections accepted on the socket.

When **SO_LINGER** is turned on and the **l_linger** field is set to 0, TCP simply drops the connection by sending out an RST (if a connection is already established). This frees up the space for the TCP protocol control block, and wakes up all tasks sleeping on the socket.

For the client side socket, the value of **l_linger** is not changed if it is set to 0. To make sure that the value of **l_linger** is 0 on a newly accepted socket connection, issue another **setsockopt( )** after the **accept( )** call.

Currently the exact value of **l_linger** time is actually ignored (other than checking for 0); that is, TCP performs the state transitions if **l_linger** is not 0, but does not explicitly use its value.

TCP_NODELAY -- Delivering Messages Immediately

Specify the **TCP_NODELAY** option for real-time protocols, such as the X Window System Protocol, that require immediate delivery of many small messages:

```
setsockopt (sock, IPPROTO_TCP, TCP_NODELAY, &optval, sizeof (optval));
```

The value at *optval* is an integer (type **int**) set to either 1 (on) or 0 (off).

By default, the VxWorks TCP implementation employs an algorithm that attempts to avoid the congestion that can be produced by a large number of small TCP segments. This typically arises with virtual terminal applications (such as **telnet** or **rlogin**) across networks that have low bandwidth and long delays. The algorithm attempts to have no more than one outstanding unacknowledged segment in the transmission channel while queueing up the rest of the smaller segments for later transmission. Another segment is sent only if enough new data is available to make up a maximum sized segment, or if the outstanding data is acknowledged.

This congestion-avoidance algorithm works well for virtual terminal protocols and bulk data transfer protocols such as FTP without any noticeable side effects. However, real-time protocols that require immediate delivery of many small messages, such as the X Window System Protocol, need to defeat this facility to guarantee proper responsiveness in their operation.

TCP_NODELAY is a mechanism to turn off the use of this algorithm. If this option is turned on and there is data to be sent out, TCP bypasses the congestion-avoidance algorithm: any available data segments are sent out if there is enough space in the send window.

S

TCP_MAXSEG -- Changing TCP MSS for the connection

Specify the **TCP_MAXSEG** option to decrease the maximum allowable size of an outgoing TCP segment. This option cannot be used to increase the MSS.

```
setsockopt (sock, IPPROTO_TCP, TCP_MAXSEG, &optval, sizeof (optval));
```

The value at *optval* is an integer set to the desired MSS (*e.g.*, 1024).

When a TCP socket is created, the MSS is initialized to the default MSS value which is determined by the configuration parameter **TCP_MSS_DFLT** (512 by default). When a connection request is received from the other end with an MSS option, the MSS is modified depending on the value of the received MSS and on the results of Path MTU Discovery (which is enabled by default). The MSS may be set as high as the outgoing interface MTU (1460 for an Ethernet). Therefore, after a call to **socket** but before a connection is established, an application can only decrease the MSS from its default of 512.

After a connection is established, the application can decrease the MSS from whatever value was selected.

SO_DEBUG -- Debugging the underlying protocol

Specify the **SO_DEBUG** option to let the underlying protocol module record debug information.

```
setsockopt (sock, SOL_SOCKET, SO_DEBUG, &optval, sizeof (optval));
```

The value at *optval* for this option is an integer (type **int**), either 1 (on) or 0 (off).

OPTION FOR DATAGRAM SOCKETS

The following section discusses an option for datagram (UDP) sockets.

SO_BROADCAST -- Sending to Multiple Destinations

Specify the **SO_BROADCAST** option when an application needs to send data to more than one destination:

```
setsockopt (sock, SOL_SOCKET, SO_BROADCAST, &optval, sizeof (optval));
```

The value at *optval* is an integer (type *int*), either 1 (on) or 0 (off).

OPTIONS FOR DATAGRAM AND RAW SOCKETS

The following section discusses options for multicasting on UDP and RAW sockets.

IP_ADD_MEMBERSHIP -- Join a Multicast Group

Specify the **IP_ADD_MEMBERSHIP** option when a process needs to join multicast group:

```
setsockopt (sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, (char *)&ipMreq,
            sizeof (ipMreq));
```

The value of *ipMreq* is an **ip_mreq** structure. **ipMreq.imr_multiaddr.s_addr** is the internet multicast address **ipMreq.imr_interface.s_addr** is the internet unicast address of the interface through which the multicast packet needs to pass.

IP_DROP_MEMBERSHIP -- Leave a Multicast Group

Specify the **IP_DROP_MEMBERSHIP** option when a process needs to leave a previously joined multicast group:

```
setsockopt (sock, IPPROTO_IP, IP_DROP_MEMBERSHIP, (char *)&ipMreq,
            sizeof (ipMreq));
```

The value of *ipMreq* is an **ip_mreq** structure. **ipMreq.imr_multiaddr.s_addr** is the internet multicast address. **ipMreq.imr_interface.s_addr** is the internet unicast address of the interface to which the multicast address was bound.

IP_MULTICAST_IF -- Select a Default Interface for Outgoing Multicasts

Specify the **IP_MULTICAST_IF** option when an application needs to specify an outgoing network interface through which all multicast packets are sent:

```
setsockopt (sock, IPPROTO_IP, IP_MULTICAST_IF, (char *)&ifAddr,
            sizeof (mCastAddr));
```

The value of *ifAddr* is an **in_addr** structure. **ifAddr.s_addr** is the internet network interface address.

IP_MULTICAST_TTL -- Select a Default TTL

Specify the **IP_MULTICAST_TTL** option when an application needs to select a default TTL (time to live) for outgoing multicast packets:

```
setsockopt (sock, IPPROTO_IP, IP_MULTICAST_TTL, &optval, sizeof(optval));
```

The value at *optval* is an integer (type *int*), time to live value.

optval(TTL)	Application	Scope
0		same interface
1		same subnet
31	local event video	
32		same site
63	local event audio	
64		same region
95	IETF channel 2 video	
127	IETF channel 1 video	
128		same continent
159	IETF channel 2 audio	
191	IETF channel 1 audio	
223	IETF channel 2 low-rate audio	
255	IETF channel 1 low-rate audio	
	unrestricted in scope	

IP_MULTICAST_LOOP -- Enable or Disable Loopback

Enable or disable loopback of outgoing multicasts.

```
setsockopt (sock, IPPROTO_IP, IP_MULTICAST_LOOP, &optval, sizeof(optval));
```

The value at *optval* is an integer (type *int*), either 1(on) or 0 (off).

OPTIONS FOR DATAGRAM, STREAM AND RAW SOCKETS

The following section discusses options for RAW, DGRAM or STREAM sockets.

IP_OPTIONS -- set options to be included in outgoing datagrams

Sets the IP options sent from this socket with every packet.

```
setsockopt (sock, IPPROTO_IP, IP_OPTIONS, optbuf, optbuflen);
```

Here *optbuf* is a buffer containing the options.

S

IP_TOS-- set options to be included in outgoing datagrams

Sets the Type-Of-Service field for each packet sent from this socket.

```
setsockopt (sock, IPPROTO_IP, IP_TOS, &optval, sizeof(optval));
```

Here *optval* is an integer (type *int*). This integer can be set to **IPTOS_LOWDELAY**, **IPTOS_THROUGHPUT**, **IPTOS_RELIABILITY**, or **IPTOS_MINCOST**, to indicate how the packets sent on this socket should be prioritized.

IP_TTL-- set the time-to-live field in outgoing datagrams

Sets the Time-To-Live field for each packet sent from this socket.

```
setsockopt (sock, IPPROTO_IP, IP_TTL, &optval, sizeof(optval));
```

Here *optval* is an integer (type *int*), indicating the number of hops a packet can take before it is discarded.

IP_RECVRETOPTS -- [un-]set queueing of reversed source route

Sets whether or not reversed source route queueing will be enabled for incoming datagrams. (Not implemented)

```
setsockopt (sock, IPPROTO_IP, IP_RECVRETOPTS, &optval, sizeof(optval));
```

Here *optval* is a boolean (type *int*). However, this option is currently not implemented, so setting it will not change the behavior of the system.

IP_RECVDSTADDR -- [un-]set queueing of IP destination address

Sets whether or not the socket will receive the IP address of the destination of an incoming datagram in control data.

```
setsockopt (sock, IPPROTO_IP, IP_RECVDSTADDR, &optval, sizeof(optval));
```

Here *optval* is a boolean (type *int*).

OPTIONS FOR BOTH STREAM AND DATAGRAM SOCKETS

The following sections describe options that can be used with either stream or datagram sockets.

SO_REUSEADDR -- Reusing a Socket Address

Specify the **SO_REUSEADDR** option to bind a stream socket to a local port that may be still bound to another stream socket:

```
setsockopt (sock, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof (optval));
```

The value at *optval* is an integer (type *int*), either 1 (on) or 0 (off).

When the **SO_REUSEADDR** option is turned on, applications may bind a stream socket to a local port. This is possible even if the port is still bound to another stream socket. It is even possible if that other socket is associated with a “zombie” protocol control block context that has not yet freed from previous sessions. The uniqueness of port number combinations for each connection is still preserved through sanity checks performed at actual connection setup time. If this option is not turned on and an application attempts to bind to a port that is being used by a zombie protocol control block, the **bind( )** call fails.

SO_REUSEPORT -- Reusing a Socket address and port

This option is similar to the **SO_REUSEADDR** option but it allows binding to the same local address and port combination.

```
setsockopt (sock, SOL_SOCKET, SO_REUSEPORT, &optval, sizeof (optval));
```

The value at *optval* is an integer (type *int*), either 1 (on) or 0 (off).

The **SO_REUSEPORT** option is mainly required by multicast applications where a number of applications need to bind to the same multicast address and port to receive multicast data. Unlike **SO_REUSEADDR** where only the later applications need to set this option, with **SO_REUSEPORT** all applications including the first to bind to the port are required to set this option. For multicast addresses **SO_REUSEADDR** and **SO_REUSEPORT** show the same behavior so **SO_REUSEADDR** can be used instead.

SO_SNDBUF -- Specifying the Size of the Send Buffer

Specify the **SO_SNDBUF** option to adjust the maximum size of the socket-level send buffer:

```
setsockopt (sock, SOL_SOCKET, SO_SNDBUF, &optval, sizeof (optval));
```

The value at *optval* is an integer (type **int**) that specifies the size of the socket-level send buffer to be allocated.

When stream or datagram sockets are created, each transport protocol reserves a set amount of space at the socket level for use when the sockets are attached to a protocol. For TCP, the default size of the send buffer is 8192 bytes. For UDP, the default size of the send buffer is 9216 bytes. Socket-level buffers are allocated dynamically from the mbuf pool.

The effect of setting the maximum size of buffers (for both **SO_SNDBUF** and **SO_RCVBUF**, described below) is not actually to allocate the mbufs from the mbuf pool. Instead, the effect is to set the high-water mark in the protocol data structure, which is used later to limit the amount of mbuf allocation. Thus, the maximum size specified for the socket level send and receive buffers can affect the performance of bulk data transfers. For example, the size of the TCP receive windows is limited by the remaining socket-level buffer space. These parameters must be adjusted to produce the optimal result for a given application.

SO_RCVBUF -- Specifying the Size of the Receive Buffer

Specify the **SO_RCVBUF** option to adjust the maximum size of the socket-level receive buffer:

```
setsockopt (sock, SOL_SOCKET, SO_RCVBUF, &optval, sizeof (optval));
```

The value at *optval* is an integer (type **int**) that specifies the size of the socket-level receive buffer to be allocated.

When stream or datagram sockets are created, each transport protocol reserves a set amount of space at the socket level for use when the sockets are attached to a protocol. For TCP, the default size is 8192 bytes. UDP reserves 41600 bytes, enough space for up to forty incoming datagrams (1 Kbyte each).

See the **SO_SNDBUF** discussion above for a discussion of the impact of buffer size on application performance.

SO_OOBINLINE -- Placing Urgent Data in the Normal Data Stream

Specify the **SO_OOBINLINE** option to place urgent data within the normal receive data stream:

```
setsockopt (sock, SOL_SOCKET, SO_OOBINLINE, &optval, sizeof (optval));
```

TCP provides an expedited data service that does not conform to the normal constraints of sequencing and flow control of data streams. The expedited service delivers "out-of-band" (urgent) data ahead of other "normal" data to provide interrupt-like services (for example, when you hit a CTRL-C during **telnet** or **rlogin** session while data is being displayed on the screen.)

TCP does not actually maintain a separate stream to support the urgent data. Instead, urgent data delivery is implemented as a pointer (in the TCP header) which points to the sequence number of the octet following the urgent data. If more than one transmission of urgent data is received from the peer, they are all put into the normal stream. This is intended for applications that cannot afford to miss out on any urgent data but are usually too slow to respond to them promptly.

RETURNS

OK, or **ERROR** if there is an invalid socket, an unknown option, an option length greater than MLEN, insufficient mbufs, or the call is unable to set the specified option.

SEE ALSO

sockLib

setvbuf( )

NAME

setvbuf( ) – specify buffering for a stream (ANSI)

SYNOPSIS

```
int setvbuf
    (
    FILE * fp,             /* stream to set buffering for */
    char * buf,            /* buffer to use (optional) */
    int    mode,           /* _IOFBF = fully buffered _IOLBF = line */
                           /* buffered _IONBF = unbuffered */
    size_t size            /* buffer size */
    )
```

DESCRIPTION

This routine sets the buffer size and buffering mode for a specified stream. It should be called only after the stream has been associated with an open file and before any other operation is performed on the stream. The argument *mode* determines how the stream will be buffered, as follows:

_IOFBF
input/output is to be fully buffered.

_IOLBF
input/output is to be line buffered.

_IONBF
input/output is to be unbuffered.

If *buf* is not a null pointer, the array it points to may be used instead of a buffer allocated by **setvbuf( )**. The argument *size* specifies the size of the array. The contents of the array at any time are indeterminate.

INCLUDE FILES

stdio.h

RETURNS

Zero, or non-zero if *mode* is invalid or the request cannot be honored.

SEE ALSO

ansiStdio

shell( )

NAME

shell( ) – the shell entry point

SYNOPSIS

```
void shell
    (
    BOOL interactive          /* should be TRUE, except for a script */
    )
```

DESCRIPTION

This routine is the shell task. It is started with a single parameter that indicates whether this is an interactive shell to be used from a terminal or a socket, or a shell that executes a script.

Normally, the shell is spawned in interactive mode by the root task, **usrRoot( )**, when VxWorks starts up. After that, **shell( )** is called only to execute scripts, or when the shell is restarted after an abort.

The shell gets its input from standard input and sends output to standard output. Both standard input and standard output are initially assigned to the console, but are redirected by **telnetdTask( )** and **rlogindTask( )**.

The shell is not reentrant, since **yacc** does not generate a reentrant parser. Therefore, there can be only a single shell executing at one time.

RETURNS

N/A

SEE ALSO

shellLib, *VxWorks Programmer's Guide: Target Shell*

shellHistory( )

NAME

shellHistory( ) – display or set the size of shell history

SYNOPSIS

```
void shellHistory
    (
    int size                  /* 0 = display, >0 = set history to new size */
    )
```

DESCRIPTION

This routine displays shell history, or resets the default number of commands displayed by shell history to *size*. By default, history size is 20 commands. Shell history is actually maintained by **ledLib**.

RETURNS N/A

SEE ALSO **shellLib**, **ledLib**, **h( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

shellInit( )

NAME **shellInit( )** – start the shell

SYNOPSIS

```
STATUS shellInit
    (
    int stackSize,          /* shell stack (0 = previous/default value) */
    int arg                 /* argument to shell task */
    )
```

DESCRIPTION This routine starts the shell task. If the configuration macro **INCLUDE_SHELLis** defined, **shellInit( )** is called by the root task, **usrRoot( )**, in **usrConfig.c**.

RETURNS **OK** or **ERROR**.

SEE ALSO **shellLib**, *VxWorks Programmer's Guide: Target Shell*

shellLock( )

NAME **shellLock( )** – lock access to the shell

SYNOPSIS

```
BOOL shellLock
    (
    BOOL request            /* TRUE = lock, FALSE = unlock */
    )
```

DESCRIPTION This routine locks or unlocks access to the shell. When locked, cooperating tasks, such as **telnetdTask( )** and **rlogindTask( )**, will not take the shell.

RETURNS **TRUE** if *request* is "lock" and the routine successfully locks the shell, otherwise **FALSE**. **TRUE** if *request* is "unlock" and the routine successfully unlocks the shell, otherwise **FALSE**.

SEE ALSO **shellLib**, *VxWorks Programmer's Guide: Target Shell*

S

shellOrigStdSet( )

NAME **shellOrigStdSet( )** – set the shell's default input/output/error file descriptors

SYNOPSIS

```
void shellOrigStdSet
    (
    int which,              /* STD_IN, STD_OUT, STD_ERR */
    int fd                  /* fd to be default */
    )
```

DESCRIPTION This routine is called to change the shell's default standard input/output/error file descriptor. Normally, it is used only by the shell, **rlogindTask( )**, and **telnetdTask( )**. Values for *which* can be **STD_IN**, **STD_OUT**, or **STD_ERR**, as defined in **vxWorks.h**. Values for *fd* can be the file descriptor for any file or device.

RETURNS N/A

SEE ALSO **shellLib**

shellPromptSet( )

NAME **shellPromptSet( )** – change the shell prompt

SYNOPSIS

```
void shellPromptSet
    (
    char * newPrompt        /* string to become new shell prompt */
    )
```

DESCRIPTION This routine changes the shell prompt string to *newPrompt*.

RETURNS N/A

SEE ALSO **shellLib**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

shellScriptAbort( )

NAME

shellScriptAbort( ) – signal the shell to stop processing a script

SYNOPSIS

```
void shellScriptAbort (void)
```

DESCRIPTION

This routine signals the shell to abort processing a script file. It can be called from within a script if an error is detected.

RETURNS

N/A

SEE ALSO

shellLib, *VxWorks Programmer's Guide: Target Shell*

show( )

NAME

show( ) – print information on a specified object

SYNOPSIS

```
void show
    (
    int objId,              /* object ID */
    int level               /* information level */
    )
```

DESCRIPTION

This command prints information on the specified object. System objects include tasks, local and shared semaphores, local and shared message queues, local and shared memory partitions, watchdogs, and symbol tables. An information level is interpreted by the objects show routine on a class by class basis. Refer to the object's library manual page for more information.

RETURNS

N/A

SEE ALSO

usrLib, **i( )**, **ti( )**, **lkup( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

S

shutdown( )

NAME

shutdown( ) – shut down a network connection

SYNOPSIS

```
STATUS shutdown
    (
    int s,                          /* socket to shut down */
    int how                         /* 0 = receives disallowed */
                                    /* 1 = sends disallowed */
                                    /* 2 = sends and receives disallowed */
    )
```

DESCRIPTION

This routine shuts down all, or part, of a connection-based socket *s*. If the value of *how* is 0, receives are disallowed. If *how* is 1, sends are disallowed. If *how* is 2, both sends and receives are disallowed.

RETURNS

ERROR if the socket is invalid or has no registered socket-specific routines; otherwise **shutdown( )** returns the return value from the socket-specific shutdown routine (typically **OK** in the case of a successful shutdown or **ERROR** otherwise).

SEE ALSO

sockLib

sigaction( )

NAME

sigaction( ) – examine and/or specify the action associated with a signal (POSIX)

SYNOPSIS

```
int sigaction
    (
    int                         signo, /* signal of handler of interest */
    const struct sigaction * pAct,  /* location of new handler */
    struct sigaction *       pOact  /* location to store old handler */
    )
```

DESCRIPTION

This routine allows the calling process to examine and/or specify the action to be associated with a specific signal.

RETURNS

OK (0), or **ERROR** (-1) if the signal number is invalid.

ERRNO

EINVAL

SEE ALSO

sigLib

sigaddset( )

NAME

sigaddset( ) – add a signal to a signal set (POSIX)

SYNOPSIS

```
int sigaddset
    (
    sigset_t * pSet,          /* signal set to add signal to */
    int        signo          /* signal to add */
    )
```

DESCRIPTION

This routine adds the signal specified by *signo* to the signal set specified by *pSet*.

RETURNS

OK (0), or **ERROR** (-1) if the signal number is invalid.

ERRNO

EINVAL

SEE ALSO

sigLib

sigblock( )

NAME

sigblock( ) – add to a set of blocked signals

SYNOPSIS

```
int sigblock
    (
    int mask                  /* mask of additional signals to be blocked */
    )
```

DESCRIPTION

This routine adds the signals in *mask* to the task's set of blocked signals. A one (1) in the bit mask indicates that the specified signal is blocked from delivery. Use the macro **SIGMASK** to construct the mask for a specified signal number.

RETURNS

The previous value of the signal mask.

SEE ALSO

sigLib, **sigprocmask( )**

S

sigdelset( )

NAME **sigdelset( )** – delete a signal from a signal set (POSIX)

SYNOPSIS

```
int sigdelset
    (
    sigset_t * pSet,           /* signal set to delete signal from */
    int        signo           /* signal to delete */
    )
```

DESCRIPTION This routine deletes the signal specified by *signo* from the signal set specified by *pSet*.

RETURNS **OK** (0), or **ERROR** (-1) if the signal number is invalid.

ERRNO **EINVAL**

SEE ALSO **sigLib**

sigemptyset( )

NAME **sigemptyset( )** – initialize a signal set with no signals included (POSIX)

SYNOPSIS

```
int sigemptyset
    (
    sigset_t * pSet            /* signal set to initialize */
    )
```

DESCRIPTION This routine initializes the signal set specified by *pSet*, such that all signals are excluded.

RETURNS **OK** (0), or **ERROR** (-1) if the signal set cannot be initialized.

ERRNO No errors are detectable.

SEE ALSO **sigLib**

sigfillset( )

NAME **sigfillset( )** – initialize a signal set with all signals included (POSIX)

SYNOPSIS

```
int sigfillset
    (
    sigset_t * pSet           /* signal set to initialize */
    )
```

DESCRIPTION This routine initializes the signal set specified by *pSet*, such that all signals are included.

RETURNS **OK** (0), or **ERROR** (-1) if the signal set cannot be initialized.

ERRNO No errors are detectable.

SEE ALSO **sigLib**

sigInit( )

NAME **sigInit( )** – initialize the signal facilities

SYNOPSIS

```
int sigInit (void)
```

DESCRIPTION This routine initializes the signal facilities. It is usually called from the system start-up routine **usrInit( )** in usrConfig, before interrupts are enabled.

RETURNS **OK**, or **ERROR** if the delete hooks cannot be installed.

ERRNO **S_taskLib_TASK_HOOK_TABLE_FULL**

SEE ALSO **sigLib**

sigismember( )

NAME **sigismember( )** – test to see if a signal is in a signal set (POSIX)

SYNOPSIS

```
int sigismember
    (
    const sigset_t * pSet,    /* signal set to test */
    int              signo    /* signal to test for */
    )
```

DESCRIPTION This routine tests whether the signal specified by *signo* is a member of the set specified by *pSet*.

RETURNS 1 if the specified signal is a member of the specified set, **OK** (0) if it is not, or **ERROR** (-1) if the test fails.

ERRNO EINVAL

SEE ALSO **sigLib**

signal( )

NAME **signal( )** – specify the handler associated with a signal

SYNOPSIS

```
void (*signal
    (
    intsigno,
    void(*pHandler) ()
    ))  ()
```

DESCRIPTION This routine chooses one of three ways in which receipt of the signal number *signo* is to be subsequently handled. If the value of *pHandler* is **SIG_DFL**, default handling for that signal will occur. If the value of *pHandler* is **SIG_IGN**, the signal will be ignored. Otherwise, *pHandler* must point to a function to be called when that signal occurs.

RETURNS The value of the previous signal handler, or **SIG_ERR**.

SEE ALSO **sigLib**

sigpending( )

NAME **sigpending( )** – retrieve the set of pending signals blocked from delivery (POSIX)

SYNOPSIS

```
int sigpending
    (
    sigset_t * pSet           /* location to store pending signal set */
    )
```

DESCRIPTION This routine stores the set of signals that are blocked from delivery and that are pending for the calling process in the space pointed to by *pSet*.

RETURNS **OK** (0), or **ERROR** (-1) if the signal TCB cannot be allocated.

ERRNO **ENOMEM**

SEE ALSO **sigLib**

sigprocmask( )

NAME **sigprocmask( )** – examine and/or change the signal mask (POSIX)

SYNOPSIS

```
int sigprocmask
    (
    int                how,     /* how signal mask will be changed */
    const sigset_t * pSet,      /* location of new signal mask */
    sigset_t *       pOset      /* location to store old signal mask */
    )
```

DESCRIPTION This routine allows the calling process to examine and/or change its signal mask. If the value of *pSet* is not **NULL**, it points to a set of signals to be used to change the currently blocked set.

The value of *how* indicates the manner in which the set is changed and consists of one of the following, defined in **signal.h**:

SIG_BLOCK
the resulting set is the union of the current set and the signal set pointed to by *pSet*.

SIG_UNBLOCK
the resulting set is the intersection of the current set and the complement of the signal set pointed to by *pSet*.

SIG_SETMASK
the resulting set is the signal set pointed to by *pSset*.

RETURNS **OK** (0), or **ERROR** (-1) if *how* is invalid.

ERRNO **EINVAL**

SEE ALSO **sigLib**, **sigsetmask( )**, **sigblock( )**

sigqueue( )

NAME **sigqueue( )** – send a queued signal to a task

SYNOPSIS

```
int sigqueue
    (
    int                tid,
    int                signo,
    const union sigval value
    )
```

DESCRIPTION The function **sigqueue( )** sends the signal specified by *signo* with the signal-parameter value specified by *value* to the process specified by *tid*.

RETURNS **OK** (0), or **ERROR** (-1) if the task ID or signal number is invalid, or if there are no queued-signal buffers available.

ERRNO **EINVAL, EAGAIN**

SEE ALSO **sigLib**

sigqueueInit( )

NAME

sigqueueInit( ) – initialize the queued signal facilities

SYNOPSIS

```
int sigqueueInit
    (
    int nQueues
    )
```

DESCRIPTION

This routine initializes the queued signal facilities. It must be called before any call to **sigqueue( )**. It is usually called from the system start-up routine **usrInit( )** in usrConfig, after **sysInit( )** is called.

It allocates *nQueues* buffers to be used by **sigqueue( )**. A buffer is used by each call to **sigqueue( )** and freed when the signal is delivered (thus if a signal is block, the buffer is unavailable until the signal is unblocked.)

RETURNS

OK, or **ERROR** if memory could not be allocated.

SEE ALSO

sigLib

sigsetmask( )

NAME

sigsetmask( ) – set the signal mask

SYNOPSIS

```
int sigsetmask
    (
    int mask                  /* new signal mask */
    )
```

DESCRIPTION

This routine sets the calling task's signal mask to a specified value. A one (1) in the bit mask indicates that the specified signal is blocked from delivery. Use the macro **SIGMASK** to construct the mask for a specified signal number.

RETURNS

The previous value of the signal mask.

SEE ALSO

sigLib, **sigprocmask( )**

sigsuspend( )

NAME **sigsuspend( )** – suspend the task until delivery of a signal (POSIX)

SYNOPSIS

```
int sigsuspend
    (
    const sigset_t * pSet     /* signal mask while suspended */
    )
```

DESCRIPTION This routine suspends the task until delivery of a signal. While suspended, *pSet* is used as the set of masked signals.

NOTE: Since the **sigsuspend( )** function suspends thread execution indefinitely, there is no successful completion return value.

RETURNS -1, always.

ERRNO **EINTR**

SEE ALSO **sigLib**

sigtimedwait( )

NAME **sigtimedwait( )** – wait for a signal

SYNOPSIS

```
int sigtimedwait
    (
    const sigset_t *        pSet,    /* the signal mask while suspended */
    struct siginfo *        pInfo,   /* return value */
    const struct timespec * pTimeout
    )
```

DESCRIPTION The function **sigtimedwait( )** selects the pending signal from the set specified by *pSet*. If multiple signals in *pSet* are pending, it will remove and return the lowest numbered one. If no signal in *pSet* is pending at the time of the call, the task will be suspend until one of the signals in *pSet* become pending, it is interrupted by an unblocked caught signal, or until the time interval specified by *pTimeout* has expired. If *pTimeout* is **NULL**, then the timeout interval is forever.

If the *pInfo* argument is non-**NULL**, the selected signal number is stored in the **si_signo** member, and the cause of the signal is stored in the **si_code** member. If the signal is a queued signal, the value is stored in the **si_value** member of *pInfo*; otherwise the content of **si_value** is undefined.

The following values are defined in **signal.h** for **si_code**:

SI_USER
the signal was sent by the **kill( )** function.

SI_QUEUE
the signal was sent by the **sigqueue( )** function.

SI_TIMER
the signal was generated by the expiration of a timer set by **timer_settime( )**.

SI_ASYNCIO
the signal was generated by the completion of an asynchronous I/O request.

SI_MESGQ
the signal was generated by the arrival of a message on an empty message queue.

The function **sigtimedwait( )** provides a synchronous mechanism for tasks to wait for 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.

RETURNS

Upon successful completion (that is, one of the signals specified by *pSet* is pending or is generated) **sigtimedwait( )** will return the selected signal number. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

ERRNO

EINTR
The wait was interrupted by an unblocked, caught signal.

EAGAIN
No signal specified by *pSet* was delivered within the specified timeout period.

EINVAL
The *pTimeout* argument specified a **tv_nsec** value less than zero or greater than or equal to 1000 million.

SEE ALSO

sigLib, **sigwait( )**

S

sigvec( )

NAME

sigvec( ) – install a signal handler

SYNOPSIS

```
int sigvec
    (
    int                    sig,  /* signal to attach handler to */
    const struct sigvec * pVec, /* new handler information */
    struct sigvec *       pOvec /* previous handler information */
    )
```

DESCRIPTION

This routine binds a signal handler routine referenced by *pVec* to a specified signal *sig*. It can also be used to determine which handler, if any, has been bound to a particular signal: **sigvec( )** copies current signal handler information for *sig* to *pOvec* and does not install a signal handler if *pVec* is set to **NULL** (0).

Both *pVec* and *pOvec* are pointers to a structure of type **struct sigvec**. The information passed includes not only the signal handler routine, but also the signal mask and additional option bits. The structure **sigvec** and the available options are defined in **signal.h**.

RETURNS

OK (0), or **ERROR** (-1) if the signal number is invalid or the signal TCB cannot be allocated.

ERRNO

EINVAL, **ENOMEM**

SEE ALSO

sigLib

sigwait( )

NAME

sigwait( ) – wait for a signal to be delivered (POSIX)

SYNOPSIS

```
int sigwait
    (
    const sigset_t * pSet,
    int *            pSig
    )
```

DESCRIPTION

This routine waits until one of the signals specified in *pSet* is delivered to the calling thread. It then stores the number of the signal received in the location pointed to by *pSig*.

The signals in *pSet* must not be ignored on entrance to **sigwait( )**. If the delivered signal has a signal handler function attached, that function is not called.

RETURNS OK, or **ERROR** on failure.

SEE ALSO **sigLib**, **sigtimedwait( )**

sigwaitinfo( )

NAME **sigwaitinfo( )** – wait for real-time signals

SYNOPSIS

```
int sigwaitinfo
    (
    const sigset_t * pSet,    /* the signal mask while suspended */
    struct siginfo * pInfo    /* return value */
    )
```

DESCRIPTION The function **sigwaitinfo( )** is equivalent to calling **sigtimedwait( )** with *pTimeout* equal to **NULL**. See that manual entry for more information.

RETURNS Upon successful completion (that is, one of the signals specified by *pSet* is pending or is generated) **sigwaitinfo( )** returns the selected signal number. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

ERRNO **EINTR**
The wait was interrupted by an unblocked, caught signal.

SEE ALSO **sigLib**

S

sin( )

NAME

sin( ) – compute a sine (ANSI)

SYNOPSIS

```
double sin
    (
    double x                  /* angle in radians */
    )
```

DESCRIPTION

This routine computes the sine of *x* in double precision. The angle *x* is expressed in radians.

INCLUDE FILES

math.h

RETURNS

The double-precision sine of *x*.

SEE ALSO

ansiMath, **mathALib**

sincos( )

NAME

sincos( ) – compute both a sine and cosine

SYNOPSIS

```
void sincos
    (
    double x,           /* angle in radians */
    double *sinResult, /* sine result buffer */
    double *cosResult  /* cosine result buffer */
    )
```

DESCRIPTION

This routine computes both the sine and cosine of *x* in double precision. The sine is copied to *sinResult* and the cosine is copied to *cosResult*.

INCLUDE FILES

math.h

RETURNS

N/A

SEE ALSO

mathALib

sincosf( )

NAME **sincosf( )** – compute both a sine and cosine

SYNOPSIS

```
void sincosf
    (
    float x,          /* angle in radians */
    float *sinResult, /* sine result buffer */
    float *cosResult  /* cosine result buffer */
    )
```

DESCRIPTION This routine computes both the sine and cosine of *x* in single precision. The sine is copied to *sinResult* and the cosine is copied to *cosResult*. The angle *x* is expressed in radians.

INCLUDE FILES **math.h**

RETURNS N/A

SEE ALSO **mathALib**

sinf( )

NAME **sinf( )** – compute a sine (ANSI)

SYNOPSIS

```
float sinf
    (
    float x    /* angle in radians */
    )
```

DESCRIPTION This routine returns the sine of *x* in single precision. The angle *x* is expressed in radians.

INCLUDE FILES **math.h**

RETURNS The single-precision sine of *x*.

SEE ALSO **mathALib**

S

sinh( )

NAME **sinh( )** – compute a hyperbolic sine (ANSI)

SYNOPSIS

```
double sinh
    (
    double x                 /* number whose hyperbolic sine is required */
    )
```

DESCRIPTION This routine returns the hyperbolic sine of *x* in double precision (IEEE double, 53 bits).

A range error occurs if *x* is too large.

INCLUDE FILES **math.h**

RETURNS The double-precision hyperbolic sine of *x*.

Special cases:
If *x* is +INF, -INF, or NaN, **sinh( )** returns *x*.

SEE ALSO **ansiMath**, **mathALib**

sinhf( )

NAME **sinhf( )** – compute a hyperbolic sine (ANSI)

SYNOPSIS

```
float sinhf
    (
    float x   /* number whose hyperbolic sine is required */
    )
```

DESCRIPTION This routine returns the hyperbolic sine of *x* in single precision.

INCLUDE FILES **math.h**

RETURNS The single-precision hyperbolic sine of *x*.

SEE ALSO **mathALib**

sleep( )

NAME

sleep( ) – delay for a specified amount of time

SYNOPSIS

```
unsigned int sleep
    (
    unsigned int secs
    )
```

DESCRIPTION

This routine causes the calling task to be blocked for *secs* seconds.

The time the task is blocked for may be longer than requested due to the rounding up of the request to the timer's resolution or to other scheduling activities (*e.g.*, a higher priority task intervenes).

RETURNS

Zero if the requested time has elapsed, or the number of seconds remaining if it was interrupted.

ERRNO

EINVAL, **EINTR**

SEE ALSO

timerLib, **nanosleep( )**, **taskDelay( )**

smMemAddToPool( )

NAME

smMemAddToPool( ) – add memory to shared memory system partition (VxMP Opt.)

SYNOPSIS

```
STATUS smMemAddToPool
    (
    char *   pPool,           /* pointer to memory pool */
    unsigned poolSize         /* block size in bytes */
    )
```

DESCRIPTION

This routine adds memory to the shared memory system partition after the initial allocation of memory. The memory added need not be contiguous with memory previously assigned, but it must be in the same address space.

pPool is the global address of shared memory added to the partition. The memory area pointed to by *pPool* must be in the same address space as the shared memory anchor and shared memory pool.

poolSize is the size in bytes of shared memory added to the partition.

S

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS **OK**, or **ERROR** if access to the shared memory system partition fails.

ERRNO **S_smObjLib_LOCK_TIMEOUT**

SEE ALSO **smMemLib**

smMemCalloc( )

NAME **smMemCalloc( )** – allocate memory for array from shared memory system partition (VxMP Opt.)

SYNOPSIS

```
void * smMemCalloc
    (
    int elemNum,              /* number of elements */
    int elemSize              /* size of elements */
    )
```

DESCRIPTION This routine allocates a block of memory for an array that contains *elemNum* elements of size *elemSize* from the shared memory system partition. The return value is the local address of the allocated shared memory block.

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS A pointer to the block, or **NULL** if the memory cannot be allocated.

ERRNO **S_memLib_NOT_ENOUGH_MEMORY**
S_smObjLib_LOCK_TIMEOUT

SEE ALSO **smMemLib**

smMemFindMax( )

NAME **smMemFindMax( )** – find largest free block in shared memory system partition (VxMP)

SYNOPSIS

```
int smMemFindMax (void)
```

DESCRIPTION This routine searches for the largest block in the shared memory system partition free list and returns its size.

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS The size (in bytes) of the largest available block, or **ERROR** if the attempt to access the partition fails.

ERRNO **S_smObjLib_LOCK_TIMEOUT**

SEE ALSO **smMemLib**

smMemFree( )

NAME **smMemFree( )** – free a shared memory system partition block of memory (VxMP Opt.)

SYNOPSIS

```
STATUS smMemFree
    (
    void * ptr                /* pointer to block of memory to be freed */
    )
```

DESCRIPTION This routine takes a block of memory previously allocated with **smMemMalloc( )** or **smMemCalloc( )** and returns it to the free shared memory system pool.

It is an error to free a block of memory that was not previously allocated.

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS **OK**, or **ERROR** if the block is invalid.

ERRNO **S_memLib_BLOCK_ERROR**, **S_smObjLib_LOCK_TIMEOUT**

SEE ALSO **smMemLib**, **smMemMalloc( )**, **smMemCalloc( )**

S

smMemMalloc( )

NAME

smMemMalloc( ) – allocate block of memory from shared memory system partition (VxMP Opt.)

SYNOPSIS

```
void * smMemMalloc
    (
    unsigned nBytes           /* number of bytes to allocate */
    )
```

DESCRIPTION

This routine allocates a block of memory from the shared memory system partition whose size is equal to or greater than *nBytes*. The return value is the local address of the allocated shared memory block.

AVAILABILITY

This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS

A pointer to the block, or **NULL** if the memory cannot be allocated.

ERRNO

S_memLib_NOT_ENOUGH_MEMORY
S_smObjLib_LOCK_TIMEOUT

SEE ALSO

smMemLib

smMemOptionsSet( )

NAME

smMemOptionsSet( ) – set debug options for shared memory system partition (VxMP Opt.)

SYNOPSIS

```
STATUS smMemOptionsSet
    (
    unsigned options          /* options for system partition */
    )
```

DESCRIPTION

This routine sets the debug options for the shared system memory partition. Two kinds of errors are detected: attempts to allocate more memory than is available, and bad blocks found when memory is freed or reallocated. In both cases, the following options can be selected for actions to be taken when an error is detected: (1) return the error status, (2) log an error message and return the error status, or (3) log an error message and suspend the calling task. These options are discussed in detail in the library manual entry for **smMemLib**.

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS **OK** or **ERROR**.

ERRNO **S_smObjLib_LOCK_TIMEOUT**

SEE ALSO **smMemLib**

smMemRealloc( )

NAME **smMemRealloc( )** – reallocate block of memory from shared memory system partition (VxMP Opt.)

SYNOPSIS

```
void * smMemRealloc
    (
    void *   pBlock,          /* block to be reallocated */
    unsigned newSize          /* new block size */
    )
```

DESCRIPTION This routine changes the size of a specified block and returns a pointer to the new block of shared memory. The contents that fit inside the new size (or old size, if smaller) remain unchanged. The return value is the local address of the reallocated shared memory block.

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS A pointer to the new block of memory, or **NULL** if the reallocation cannot be completed.

ERRNO **S_memLib_NOT_ENOUGH_MEMORY**
S_memLib_BLOCK_ERROR
S_smObjLib_LOCK_TIMEOUT

SEE ALSO **smMemLib**

smMemShow( )

NAME

smMemShow( ) – show the shared memory system partition blocks and statistics (VxMP Opt.)

SYNOPSIS

```
void smMemShow
    (
    int type                  /* 0 = statistics, 1 = statistics & list */
    )
```

DESCRIPTION

This routine displays the total amount of free space in the shared memory system partition, including number of blocks, average block size, and maximum block size. It also shows the number of blocks currently allocated, and the average allocated block size.

If *type* is 1, it displays a list of all the blocks in the free list of the shared memory system partition.

WARNING: This routine locks access to the shared memory system partition while displaying the information. This can compromise the access time to the partition from other CPUs in the system. Generally, this routine is used for debugging purposes only.

EXAMPLE

```
-> smMemShow 1
FREE LIST:
  num    addr       size
  --- ---------- ----------
    1   0x4ffef0        264
    2   0x4fef18       1700
SUMMARY:
    status         bytes    blocks   ave block  max block
--------------- --------- -------- ---------- ----------
        current
           free      1964        2        982       1700
          alloc      2356        1       2356          -
     cumulative
          alloc      2620        2       1310          -
value = 0 = 0x0
```

AVAILABILITY

This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS

N/A

SEE ALSO

smMemShow, **windsh**, *VxWorks Programmer's Guide: Target Shell*, *Tornado User's Guide: Shell*

smNameAdd( )

NAME

smNameAdd( ) – add a name to the shared memory name database (VxMP Opt.)

SYNOPSIS

```
STATUS smNameAdd
    (
    char * name,              /* name string to enter in database */
    void * value,             /* value associated with name */
    int    type               /* type associated with name */
    )
```

DESCRIPTION

This routine adds a name of specified object type and value to the shared memory objects name database.

The *name* parameter is an arbitrary null-terminated string with a maximum of 20 characters, including EOS.

By convention, *type* values of less than 0x1000 are reserved by VxWorks; all other values are user definable. The following types are predefined in **smNameLib.h**:

Name	Value	Type
T_SM_SEM_B	= 0	shared binary semaphore
T_SM_SEM_C	= 1	shared counting semaphore
T_SM_MSG_Q	= 2	shared message queue
T_SM_PART_ID	= 3	shared memory partition
T_SM_BLOCK	= 4	shared memory allocated block

A name can be entered only once in the database, but there can be more than one name associated with an object ID.

AVAILABILITY

This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS

OK, or **ERROR** if there is insufficient memory for *name* to be allocated, if *name* is already in the database, or if the database is already full.

ERRNO

S_smNameLib_NOT_INITIALIZED
S_smNameLib_NAME_TOO_LONG
S_smNameLib_NAME_ALREADY_EXIST
S_smNameLib_DATABASE_FULL
S_smObjLib_LOCK_TIMEOUT

SEE ALSO

smNameLib, **smNameShow**

S

smNameFind( )

NAME

smNameFind( ) – look up a shared memory object by name (VxMP Opt.)

SYNOPSIS

```
STATUS smNameFind
    (
    char *    name,           /* name to search for */
    void * * pValue,          /* pointer where to return value */
    int *     pType,          /* pointer where to return object type */
    int       waitType        /* NO_WAIT or WAIT_FOREVER */
    )
```

DESCRIPTION

This routine searches the shared memory objects name database for an object matching a specified *name*. If the object is found, its value and type are copied to the addresses pointed to by *pValue* and *pType*. The value of *waitType* can be one of the following:

NO_WAIT (0)
: The call returns immediately, even if *name* is not in the database.

WAIT_FOREVER (-1)
: The call returns only when *name* is available in the database. If *name* is not already in, the database is scanned periodically as the routine waits for *name* to be entered.

AVAILABILITY

This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS

OK, or **ERROR** if the object is not found, if *name* is too long, or the wait type is invalid.

ERRNO

S_smNameLib_NOT_INITIALIZED
S_smNameLib_NAME_TOO_LONG
S_smNameLib_NAME_NOT_FOUND
S_smNameLib_INVALID_WAIT_TYPE
S_smObjLib_LOCK_TIMEOUT

SEE ALSO

smNameLib, **smNameShow**

smNameFindByValue( )

NAME

smNameFindByValue( ) – look up a shared memory object by value (VxMP Opt.)

SYNOPSIS

```
STATUS smNameFindByValue
    (
    void * value,              /* value to search for */
    char * name,               /* pointer where to return name */
    int *  pType,              /* pointer where to return object type */
    int    waitType            /* NO_WAIT or WAIT_FOREVER */
    )
```

DESCRIPTION

This routine searches the shared memory name database for an object matching a specified value. If the object is found, its name and type are copied to the addresses pointed to by *name* and *pType*. The value of *waitType* can be one of the following:

NO_WAIT (0)
The call returns immediately, even if the object value is not in the database.

WAIT_FOREVER (-1)
The call returns only when the object value is available in the database.

AVAILABILITY

This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS

OK, or **ERROR** if *value* is not found or if the wait type is invalid.

ERRNO

S_smNameLib_NOT_INITIALIZED
S_smNameLib_VALUE_NOT_FOUND
S_smNameLib_INVALID_WAIT_TYPE
S_smObjLib_LOCK_TIMEOUT

SEE ALSO

smNameLib, **smNameShow**

S

smNameRemove( )

NAME

smNameRemove( ) – remove an object from the shared memory objects name database (VxMP Opt.)

SYNOPSIS

```
STATUS smNameRemove
    (
    char * name               /* name of object to remove */
    )
```

DESCRIPTION

This routine removes an object called *name* from the shared memory objects name database.

AVAILABILITY

This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS

OK, or **ERROR** if the object name is not in the database or if *name* is too long.

ERRNO

S_smNameLib_NOT_INITIALIZED
S_smNameLib_NAME_TOO_LONG
S_smNameLib_NAME_NOT_FOUND
S_smObjLib_LOCK_TIMEOUT

SEE ALSO

smNameLib, **smNameShow**

smNameShow( )

NAME

smNameShow( ) – show the contents of the shared memory objects name database (VxMP Opt.)

SYNOPSIS

```
STATUS smNameShow
    (
    int level                 /* information level */
    )
```

DESCRIPTION

This routine displays the names, values, and types of objects stored in the shared memory objects name database. Predefined types are shown, using their ASCII representations; all other types are printed in hexadecimal.

The *level* parameter defines the level of database information displayed. If *level* is 0, only statistics on the database contents are displayed. If *level* is greater than 0, then both statistics and database contents are displayed.

WARNING: This routine locks access to the shared memory objects name database while displaying its contents. This can compromise the access time to the name database from other CPUs in the system. Generally, this routine is used for debugging purposes only.

EXAMPLE

```
-> smNameShow
Names in Database  Max : 30  Current : 6  Free : 24
-> smNameShow 1
Names in Database  Max : 30  Current : 6  Free : 24
Name                Value         Type
---------------- ----------- -------------
inputImage        0x802340    SM_MEM_BLOCK
ouputImage        0x806340    SM_MEM_BLOCK
imagePool         0x802001    SM_MEM_PART
imageInSem        0x8e0001    SM_SEM_B
imageOutSem       0x8e0101    SM_SEM_C
actionQ           0x8e0201    SM_MSG_Q
userObject        0x8e0400    0x1b0
```

AVAILABILITY

This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS

OK, or **ERROR** if the name facility is not initialized.

ERRNO

S_smNameLib_NOT_INITIALIZED
S_smObjLib_LOCK_TIMEOUT

SEE ALSO

smNameShow, **smNameLib**

---

smNetShow( )

NAME

smNetShow( ) – show information about a shared memory network

SYNOPSIS

```
STATUS smNetShow
    (
    char * ifName,            /* backplane interface name (NULL == "sm0") */
    BOOL   zero               /* TRUE = zap totals */
    )
```

DESCRIPTION This routine displays information about the different CPUs configured in a shared memory network specified by *ifName*. It prints error statistics and zeros these fields if *zero* is set to **TRUE**.

EXAMPLE

```
-> smNetShow
Anchor at 0x800000
heartbeat = 705, header at 0x800010, free pkts = 237.
cpu int type    arg1       arg2       arg3    queued pkts
--- -------- ---------- ---------- ---------- -----------
 0  poll            0x0        0x0        0x0       0
 1  poll            0x0        0x0        0x0       0
 2  bus-int         0x3       0xc9        0x0       0
 3  mbox-2         0x2d     0x8000        0x0       0
input packets = 192     output packets = 164
output errors = 0       collisions = 0
value = 1 = 0x1
```

RETURNS **OK**, or **ERROR** if there is a hardware setup problem or the routine cannot be initialized.

SEE ALSO **smNetShow**, **smNetLib**

---

smObjAttach( )

NAME **smObjAttach( )** – attach the calling CPU to the shared memory objects facility (VxMP Opt.)

SYNOPSIS

```
STATUS smObjAttach
    (
    SM_OBJ_DESC * pSmObjDesc  /* pointer to shared memory descriptor */
    )
```

DESCRIPTION This routine "attaches" the calling CPU to the shared memory objects facility. The shared memory area is identified by the shared memory descriptor with an address specified by *pSmObjDesc*. The descriptor must already have been initialized by calling **smObjInit( )**.

This routine is called automatically when the component **INCLUDE_SM_OBJ** is included.

This routine will complete the attach process only if and when the shared memory has been initialized by the master CPU. If the shared memory is not recognized as active within the timeout period (10 minutes), this routine returns **ERROR**.

The **smObjAttach( )** routine connects the shared memory objects handler to the shared memory interrupt. Note that this interrupt may be shared between the shared memory network driver and the shared memory objects facility when both are used at the same time.

WARNING: Once a CPU has attached itself to the shared memory objects facility, it cannot be detached. Since the shared memory network driver and the shared memory objects facility use the same low-level attaching mechanism, a CPU cannot be detached from a shared memory network driver if the CPU also uses shared memory objects.

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS **OK**, or **ERROR** if the shared memory objects facility is not active or the number of CPUs exceeds the maximum.

ERRNO **S_smLib_INVALID_CPU_NUMBER**

SEE ALSO **smObjLib**, **smObjSetup( )**, **smObjInit( )**

smObjGlobalToLocal( )

NAME **smObjGlobalToLocal( )** – convert a global address to a local address (VxMP Opt.)

SYNOPSIS

```
void * smObjGlobalToLocal
    (
    void * globalAdrs         /* global address to convert */
    )
```

DESCRIPTION This routine converts a global shared memory address *globalAdrs* to its corresponding local value. This routine does not verify that *globalAdrs*is really a valid global shared memory address.

All addresses stored in shared memory are global. Any access made to shared memory by the local CPU must be done using local addresses. This routine and **smObjLocalToGlobal( )** are used to convert between these address types.

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS The local shared memory address pointed to by *globalAdrs*.

SEE ALSO **smObjLib**, **smObjLocalToGlobal( )**

S

smObjInit( )

NAME

smObjInit( ) – initialize a shared memory objects descriptor (VxMP Opt.)

SYNOPSIS

```
void smObjInit
    (
    SM_OBJ_DESC * pSmObjDesc,      /* ptr to shared memory descriptor */
    SM_ANCHOR *   anchorLocalAdrs, /* shared memory anchor local adrs */
    int           ticksPerBeat,    /* cpu ticks per heartbeat */
    int           smObjMaxTries,   /* max no. of tries to obtain spinLock */
    int           intType,         /* interrupt method */
    int           intArg1,         /* interrupt argument #1 */
    int           intArg2,         /* interrupt argument #2 */
    int           intArg3          /* interrupt argument #3 */
    )
```

DESCRIPTION

This routine initializes a shared memory descriptor. The descriptor must already be allocated in the CPU's local memory. Once the descriptor has been initialized by this routine, the CPU may attach itself to the shared memory area by calling **smObjAttach( )**.

Only the shared memory descriptor itself is modified by this routine. No structures in shared memory are affected.

Parameters:

pSmObjDesc
The address of the shared memory descriptor to be initialized; this structure must be allocated before **smObjInit( )** is called.

anchorLocalAdrs
The memory address by which the local CPU may access the shared memory anchor. This address may vary among CPUs in the system because of address offsets (particularly if the anchor is located in one CPU's dual-ported memory).

ticksPerBeat
Specifies the frequency of the shared memory anchor's heartbeat. The frequency is expressed in terms of how many CPU ticks on the local CPU correspond to one heartbeat period.

smObjMaxTries
Specifies the maximum number of tries to obtain access to an internal mutually exclusive data structure.

intType, intArg1, intArg2, intArg3
Allow a CPU to announce the method by which it is to be notified of shared memory events. See the manual entry for **if_sm** for a discussion about interrupt types and their associated parameters.

This routine is called automatically when the component **INCLUDE_SM_OBJ** is included.

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS N/A

SEE ALSO **smObjLib**, **smObjSetup( )**, **smObjAttach( )**

smObjLibInit( )

NAME **smObjLibInit( )** – install the shared memory objects facility (VxMP Opt.)

SYNOPSIS

```
STATUS smObjLibInit (void)
```

DESCRIPTION This routine installs the shared memory objects facility. It is called automatically when the component **INCLUDE_SM_OBJ** is included.

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS **OK**, or **ERROR** if the shared memory objects facility has already been installed.

SEE ALSO **smObjLib**

S

smObjLocalToGlobal( )

NAME **smObjLocalToGlobal( )** – convert a local address to a global address (VxMP Opt.)

SYNOPSIS

```
void * smObjLocalToGlobal
    (
    void * localAdrs          /* local address to convert */
    )
```

DESCRIPTION This routine converts a local shared memory address *localAdrs* to its corresponding global value. This routine does not verify that *localAdrs* is really a valid local shared memory address.

All addresses stored in shared memory are global. Any access made to shared memory by the local CPU must be done using local addresses. This routine and **smObjGlobalToLocal( )** are used to convert between these address types.

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS The global shared memory address pointed to by *localAdrs*.

SEE ALSO **smObjLib**, **smObjGlobalToLocal( )**

smObjSetup( )

NAME **smObjSetup( )** – initialize the shared memory objects facility (VxMP Opt.)

SYNOPSIS

```
STATUS smObjSetup
    (
    SM_OBJ_PARAMS * smObjParams /* setup parameters */
    )
```

DESCRIPTION This routine initializes the shared memory objects facility by filling the shared memory header. It must be called only once by the shared memory master CPU. It is called automatically only by the master CPU, when the component **INCLUDE_SM_OBJ** is included.

Any CPU on the system backplane can use the shared memory objects facility; however, the facility must first be initialized on the master CPU. Then before other CPUs are attached to the shared memory area by **smObjAttach( )**, each must initialize its own shared memory objects descriptor using **smObjInit( )**. This mechanism is similar to the one used by the shared memory network driver.

The *smObjParams* parameter is a pointer to a structure containing the values used to describe the shared memory objects setup. This structure is defined as follows in **smObjLib.h**:

```
typedef struct sm_obj_params    /* setup parameters */
    {
    BOOL        allocatedPool;  /* TRUE if shared memory pool is malloced */
    SM_ANCHOR * pAnchor;        /* shared memory anchor                   */
    char *      smObjFreeAdrs;  /* start address of shared memory pool    */
    int         smObjMemSize;   /* memory size reserved for shared memory */
    int         maxCpus;        /* max number of CPUs in the system       */
    int         maxTasks;       /* max number of tasks using smObj        */
```

```
int         maxSems;        /* max number of shared semaphores        */
int         maxMsgQueues;   /* max number of shared message queues    */
int         maxMemParts;    /* max number of shared memory partitions */
int         maxNames;       /* max number of names of shared objects  */
} SM_OBJ_PARAMS;
```

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS **OK**, or **ERROR** if the shared memory pool cannot hold all the requested objects or the number of CPUs exceeds the maximum.

ERRNO **S_smObjLib_TOO_MANY_CPU**
S_smObjLib_SHARED_MEM_TOO_SMALL

SEE ALSO **smObjLib**, **smObjInit( )**, **smObjAttach( )**

---

smObjShow( )

NAME **smObjShow( )** – display the current status of shared memory objects (VxMP Opt.)

SYNOPSIS `STATUS smObjShow (void)`

DESCRIPTION This routine displays useful information about the current status of shared memory objects facilities.

WARNING: The information returned by this routine is not static and may be obsolete by the time it is examined. This information is generally used for debugging purposes only.

EXAMPLE

```
-> smObjShow
Shared Mem Anchor Local Addr: 0x600.
Shared Mem Hdr Local Addr:    0xb1514.
Attached CPU :                5
Max Tries to Take Lock:       1
Shared Object Type    Current    Maximum  Available
-------------------- ---------- --------- ----------
Tasks                         1        20         19
Binary Semaphores             8        30         20
Counting Semaphores           2        30         20
Messages Queues               3        10          7
Memory Partitions             1         4          3
Names in Database            16       100         84
```

S

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS **OK**, or **ERROR** if no shared memory objects are initialized.

ERRNO **S_smObjLib_NOT_INITIALIZED**

SEE ALSO **smObjShow**, **smObjLib**

smObjTimeoutLogEnable( )

NAME **smObjTimeoutLogEnable( )** – control logging of failed attempts to take a spin-lock (VxMP Opt.)

SYNOPSIS

```
void smObjTimeoutLogEnable
    (
    BOOL timeoutLogEnable     /* TRUE to enable, FALSE to disable */
    )
```

DESCRIPTION This routine enables or disables the printing of a message when an attempt to take a shared memory spin-lock fails.

By default, message logging is enabled.

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS N/A

SEE ALSO **smObjLib**

sntpcTimeGet( )

NAME

sntpcTimeGet( ) – retrieve the current time from a remote source

SYNOPSIS

```
STATUS sntpcTimeGet
    (
    char *             pServerAddr, /* server IP address or hostname */
    u_int              timeout,     /* timeout interval in ticks */
    struct timespec * pCurrTime    /* storage for retrieved time value */
    )
```

DESCRIPTION

This routine stores the current time as reported by an SNTP/NTP server in the location indicated by *pCurrTime*. The reported time is first converted to the elapsed time since January 1, 1970, 00:00, GMT, which is the base value used by UNIX systems. If *pServerAddr* is **NULL**, the routine listens for messages sent by an SNTP/NTP server in broadcast mode. Otherwise, this routine sends a request to the specified SNTP/NTP server and extracts the reported time from the reply. In either case, an error is returned if no message is received within the interval specified by *timeout*. Typically, SNTP/NTP servers operating in broadcast mode send update messages every 64 to 1024 seconds. An infinite timeout value is specified by **WAIT_FOREVER**.

RETURNS

OK, or **ERROR** if unsuccessful.

ERRNO

S_sntpcLib_INVALID_PARAMETER, **S_sntpcLib_INVALID_ADDRESS**, **S_sntpcLib_TIMEOUT**, **S_sntpcLib_SERVER_UNSYNC**, **S_sntpcLib_VERSION_UNSUPPORTED**

SEE ALSO

sntpcLib

sntpsClockSet( )

NAME

sntpsClockSet( ) – assign a routine to access the reference clock

SYNOPSIS

```
STATUS sntpsClockSet
    (
    FUNCPTR pClockHookRtn     /* new interface to reference clock */
    )
```

DESCRIPTION

This routine installs a hook routine that is called to access the reference clock used by the SNTP server. This hook routine must use the following interface:

```
    STATUS sntpsClockHook (int request, void *pBuffer);
```

The hook routine should copy one of three settings used by the server to construct outgoing NTP messages into *pBuffer* according to the value of the *request* parameter. If the requested setting is available, the installed routine should return **OK** (or **ERROR** otherwise).

This routine calls the given hook routine with the *request* parameter set to **SNTPS_ID** to get the 32-bit reference identifier in the format specified in RFC 1769. It also calls the hook routine with *request* set to **SNTPS_RESOLUTION** to retrieve a 32-bit value containing the clock resolution in nanoseconds. That value will be used to determine the 8-bit signed integer indicating the clock precision (according to the format specified in RFC 1769). Other library routines will set the *request* parameter to **SNTPS_TIME** to retrieve the current 64-bit NTP timestamp from *pBuffer* in host byte order. The routine **sntpsNsecToFraction( )** will convert a value in nanoseconds to the format required for the NTP fractional part.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

OK or **ERROR**.

ERRNO

N/A

SEE ALSO

sntpsLib

sntpsConfigSet( )

NAME **sntpsConfigSet( )** – change SNTP server broadcast settings

SYNOPSIS

```
STATUS sntpsConfigSet
    (
    int    setting,           /* configuration option to change */
    void * pValue             /* new value for parameter */
    )
```

DESCRIPTION This routine alters the configuration of the SNTP server when operating in broadcast mode. A *setting* value of **SNTPS_DELAY** interprets the contents of *pValue* as the new 16-bit broadcast interval. When *setting* equals **SNTPS_ADDRESS**, *pValue* should provide the string representation of an IP broadcast or multicast address (for example, "224.0.1.1"). Any changed settings will take effect after the current broadcast interval is completed and the corresponding NTP message is sent.

RETURNS **OK** or **ERROR**.

ERRNO **S_sntpsLib_INVALID_PARAMETER**

SEE ALSO **sntpsLib**

sntpsNsecToFraction( )

NAME **sntpsNsecToFraction( )** – convert portions of a second to NTP format

SYNOPSIS

```
ULONG sntpsNsecToFraction
    (
    ULONG nsecs               /* nanoseconds to convert to binary fraction */
    )
```

DESCRIPTION This routine is provided for convenience in fulfilling an **SNTPS_TIME** request to the clock hook. It converts a value in nanoseconds to the fractional part of the NTP timestamp format. The routine is not designed to convert non-normalized values greater than or equal to one second. Although the NTP time format provides a precision of about 200 pico-seconds, rounding errors in the conversion process decrease the accuracy as the input value increases. In the worst case, only the 24 most significant bits are valid, which reduces the precision to tenths of a micro-second.

RETURNS Value for NTP fractional part in host-byte order.

ERRNO N/A

SEE ALSO **sntpsLib**

so( )

NAME **so( )** – single-step, but step over a subroutine

SYNOPSIS

```
STATUS so
    (
    int task                  /* task to step; 0 = use default */
    )
```

DESCRIPTION This routine single-steps a task that is stopped at a breakpoint. However, if the next instruction is a JSR or BSR, **so( )** breaks at the instruction following the subroutine call instead.

To execute, enter:

```
    -> so [task]
```

If *task* is omitted or zero, the last task referenced is assumed.

SEE ALSO **dbgLib**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

socket( )

NAME **socket( )** – open a socket

SYNOPSIS

```
int socket
    (
    int domain,               /* address family (for example, AF_INET) */
    int type,                 /* SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW */
    int protocol              /* socket protocol (usually 0) */
    )
```

DESCRIPTION This routine opens a socket and returns a socket descriptor. The socket descriptor is passed to the other socket routines to identify the socket. The socket descriptor is a standard I/O system file descriptor (*fd*) and can be used with the **close( )**, **read( )**, **write( )**, and **ioctl( )** routines.

Available socket types include:

SOCK_STREAM
Specifies a connection-based (stream) socket.

SOCK_DGRAM
Specifies a datagram (UDP) socket.

SOCK_RAW
Specifies a raw socket.

RETURNS A socket descriptor, or **ERROR**.

SEE ALSO **sockLib**

sockUploadPathClose( )

NAME **sockUploadPathClose( )** – close the socket upload path (Windview)

SYNOPSIS

```
void sockUploadPathClose
    (
    UPLOAD_ID upId            /* generic upload-path descriptor */
    )
```

DESCRIPTION This routine closes the socket connection to the event receiver on the host.

RETURNS N/A

SEE ALSO **wvSockUploadPathLib**, **sockUploadPathCreate( )**

S

sockUploadPathCreate( )

NAME **sockUploadPathCreate( )** – establish an upload path to the host using a socket (Windview)

SYNOPSIS

```
UPLOAD_ID sockUploadPathCreate
    (
    char * ipAddress,          /* server's hostname or IP address in */
                               /* .-notation */
    short  port                /* port number to bind to */
    )
```

DESCRIPTION This routine initializes the TCP/IP connection to the host process that receives uploaded events. It can be retried if the connection attempt fails.

RETURNS The **UPLOAD_ID**, or **NULL** if the connection cannot be completed or memory for the ID is not available.

SEE ALSO **wvSockUploadPathLib**, **sockUploadPathClose( )**

sockUploadPathLibInit( )

NAME **sockUploadPathLibInit( )** – initialize **wvSockUploadPathLib** library (Windview)

SYNOPSIS

```
STATUS sockUploadPathLibInit (void)
```

DESCRIPTION This routine initializes **wvSockUploadPathLib** by pulling in the routines in this file for use with WindView. It is called during system configuration from **usrWindview.c**.

RETURN **OK**.

SEE ALSO **wvSockUploadPathLib**

sockUploadPathWrite( )

NAME

sockUploadPathWrite( ) – write to the socket upload path (Windview)

SYNOPSIS

```
int sockUploadPathWrite
    (
    UPLOAD_ID upId,           /* generic upload-path descriptor */
    char *    pStart,         /* address of data to write */
    size_t    size            /* number of bytes of data at pStart */
    )
```

DESCRIPTION

This routine writes *size* bytes of data beginning at *pStart* to the upload path between the target and the event receiver on the host.

RETURNS

The number of bytes written, or **ERROR**.

SEE ALSO

wvSockUploadPathLib, **sockUploadPathCreate( )**

sp( )

NAME

sp( ) – spawn a task with default parameters

SYNOPSIS

```
int sp
    (
    FUNCPTR func,           /* function to call */
    int     arg1,           /* first of nine args to pass to spawned task */
    int     arg2,
    int     arg3,
    int     arg4,
    int     arg5,
    int     arg6,
    int     arg7,
    int     arg8,
    int     arg9
    )
```

DESCRIPTION

This command spawns a specified function as a task with the following defaults:

priority:
100

S

stack size:
20,000 bytes

task ID:
highest not currently used

task options:
VX_FP_TASK - execute with floating-point coprocessor support.

task name:
A name of the form **tN** where N is an integer which increments as new tasks are spawned, e.g., **t1**, **t2**, **t3**, etc.

The task ID is displayed after the task is spawned.

This command is a short form of the underlying **taskSpawn( )** routine, convenient for spawning tasks in which the default parameters are satisfactory. If the default parameters are unacceptable, **taskSpawn( )** should be called directly.

RETURNS A task ID, or **ERROR** if the task cannot be spawned.

SEE ALSO **usrLib**, **taskLib**, **taskSpawn( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

sprintf( )

NAME **sprintf( )** – write a formatted string to a buffer (ANSI)

SYNOPSIS

```
int sprintf
    (
    char *       buffer,       /* buffer to write to */
    const char * fmt,          /* format string */
                 ...           /* optional arguments to format */
    )
```

DESCRIPTION This routine copies a formatted string to a specified buffer, which is null-terminated. Its function and syntax are otherwise identical to **printf( )**.

RETURNS The number of characters copied to *buffer*, not including the **NULL** terminator.

SEE ALSO **fioLib**, **printf( )**, *American National Standard for Information Systems -Programming Language - C, ANSI X3.159-1989: Input/Output (**stdio.h**)*

spy( )

NAME
spy( ) – begin periodic task activity reports

SYNOPSIS
```
void spy
    (
    int freq,               /* reporting freq in sec, 0 = default of 5 */
    int ticksPerSec         /* interrupt clock freq, 0 = default of 100 */
    )
```

DESCRIPTION
This routine collects task activity data and periodically runs **spyReport( )**. Data is gathered *ticksPerSec* times per second, and a report is made every *freq* seconds. If *freq* is zero, it defaults to 5 seconds. If *ticksPerSec* is omitted or zero, it defaults to 100.

This routine spawns **spyTask( )** to do the actual reporting.

It is not necessary to call **spyClkStart( )** before running **spy( )**.

RETURNS
N/A

SEE ALSO
usrLib, **spyLib**, **spyClkStart( )**, **spyTask( )**, *VxWorks Programmer's Guide: Target Shell*

spyClkStart( )

NAME
spyClkStart( ) – start collecting task activity data

SYNOPSIS
```
STATUS spyClkStart
    (
    int intsPerSec          /* timer interrupt freq, 0 = default of 100 */
    )
```

DESCRIPTION
This routine begins data collection by enabling the auxiliary clock interrupts at a frequency of *intsPerSec* interrupts per second. If *intsPerSec* is omitted or zero, the frequency will be 100. Data from previous collections is cleared.

RETURNS
OK, or **ERROR** if the CPU has no auxiliary clock, or if task create and delete hooks cannot be installed.

SEE ALSO
usrLib, **spyLib**, **sysAuxClkConnect( )**, *VxWorks Programmer's Guide: Target Shell*

spyClkStop( )

NAME **spyClkStop( )** – stop collecting task activity data

SYNOPSIS

```
void spyClkStop (void)
```

DESCRIPTION This routine disables the auxiliary clock interrupts. Data collected remains valid until the next **spyClkStart( )** call.

RETURNS N/A

SEE ALSO **usrLib**, **spyLib**, **spyClkStart( )**, *VxWorks Programmer's Guide: Target Shell*

spyHelp( )

NAME **spyHelp( )** – display task monitoring help menu

SYNOPSIS

```
void spyHelp (void)
```

DESCRIPTION This routine displays a summary of **spyLib** utilities:

```
spyHelp                       Print this list
spyClkStart [ticksPerSec]     Start task activity monitor running
                                at ticksPerSec ticks per second
spyClkStop                    Stop collecting data
spyReport                     Prints display of task activity
                                statistics
spyStop                       Stop collecting data and reports
spy     [freq[,ticksPerSec]]  Start spyClkStart and do a report
                                every freq seconds
ticksPerSec defaults to 100.  freq defaults to 5 seconds.
```

RETURNS N/A

SEE ALSO **usrLib**, **spyLib**, *VxWorks Programmer's Guide: Target Shell*

spyLibInit( )

NAME **spyLibInit( )** – initialize task CPU utilization tool package

SYNOPSIS

```
void spyLibInit (void)
```

DESCRIPTION This routine initializes the task CPU utilization tool package. If the configuration macro **INCLUDE_SPY** is defined, it is called by the root task, **usrRoot( )**, in **usrConfig.c**.

RETURNS N/A

SEE ALSO **spyLib**, **usrLib**

spyReport( )

NAME **spyReport( )** – display task activity data

SYNOPSIS

```
void spyReport (void)
```

DESCRIPTION This routine reports on data gathered at interrupt level for the amount of CPU time utilized by each task, the amount of time spent at interrupt level, the amount of time spent in the kernel, and the amount of idle time. Time is displayed in ticks and as a percentage, and the data is shown since both the last call to **spyClkStart( )** and the last **spyReport( )**. If no interrupts have occurred since the last **spyReport( )**, nothing is displayed.

RETURNS N/A

SEE ALSO **usrLib**, **spyLib**, **spyClkStart( )**, *VxWorks Programmer's Guide: Target Shell*

spyStop( )

NAME **spyStop( )** – stop spying and reporting

SYNOPSIS

```
void spyStop (void)
```

DESCRIPTION This routine calls **spyClkStop( )**. Any periodic reporting by **spyTask( )** is terminated.

RETURNS N/A

SEE ALSO **usrLib**, **spyLib**, **spyClkStop( )**, **spyTask( )**, *VxWorks Programmer's Guide: Target Shell*

spyTask( )

NAME **spyTask( )** – run periodic task activity reports

SYNOPSIS

```
void spyTask
    (
    int freq                  /* reporting frequency, in seconds */
    )
```

DESCRIPTION This routine is spawned as a task by **spy( )** to provide periodic task activity reports. It prints a report, delays for the specified number of seconds, and repeats.

RETURNS N/A

SEE ALSO **usrLib**, **spyLib**, **spy( )**, *VxWorks Programmer's Guide: Target Shell*

sqrt( )

NAME **sqrt( )** – compute a non-negative square root (ANSI)

SYNOPSIS

```
double sqrt
    (
    double x                  /* value to compute the square root of */
    )
```

DESCRIPTION This routine computes the non-negative square root of *x* in double precision. A domain error occurs if the argument is negative.

INCLUDE FILES **math.h**

RETURNS The double-precision square root of *x* or 0 if *x* is negative.

ERRNO **EDOM**

SEE ALSO **ansiMath**, **mathALib**

sqrtf( )

NAME **sqrtf( )** – compute a non-negative square root (ANSI)

SYNOPSIS

```
float sqrtf
    (
    float x    /* value to compute the square root of */
    )
```

DESCRIPTION This routine returns the non-negative square root of *x* in single precision.

INCLUDE FILES **math.h**

RETURNS The single-precision square root of *x*.

SEE ALSO **mathALib**

sr( )

NAME
sr( ) – return the contents of the status register (68K, SH)

SYNOPSIS

```
int sr
    (
    int taskId                /* task ID, 0 means default task */
    )
```

DESCRIPTION
This command extracts the contents of the status register from the TCB of a specified task. If *taskId* is omitted or zero, the last task referenced is assumed.

For SH, similar routines are provided for all control registers (**gbr**, **vbr**): **gbr( )**, **vbr( )**.

RETURNS
The contents of the status register (or the requested control register).

SEE ALSO
dbgArchLib, *VxWorks Programmer's Guide: Target Shell*

srand( )

NAME
srand( ) – reset the value of the seed used to generate random numbers (ANSI)

SYNOPSIS

```
void * srand
    (
    uint_t seed               /* random number seed */
    )
```

DESCRIPTION
This routine resets the seed value used by **rand( )**. If **srand( )** is then called with the same seed value, the sequence of pseudo-random numbers is repeated. If **rand( )** is called before any calls to **srand( )** have been made, the same sequence shall be generated as when **srand( )** is first called with the seed value of 1.

INCLUDE FILES
stdlib.h

RETURNS
N/A

SEE ALSO
ansiStdlib, **rand( )**

sscanf( )

NAME

sscanf( ) – read and convert characters from an ASCII string (ANSI)

SYNOPSIS

```
int sscanf
    (
    const char * str,          /* string to scan */
    const char * fmt,          /* format string */
                 ...           /* optional arguments to format string */
    )
```

DESCRIPTION

This routine reads characters from the string *str*, interprets them according to format specifications in the string *fmt*, which specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input.

If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

The format is a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: one or more white-space characters; an ordinary multibyte character (neither % nor a white-space character); or a conversion specification. Each conversion specification is introduced by the % character. After the %, the following appear in sequence:

– An optional assignment-suppressing character *.

– An optional non-zero decimal integer that specifies the maximum field width.

– An optional **h**, **l** (ell) or **ll** (ell-ell) indicating the size of the

receiving object. The conversion specifiers **d**, **i**, and **n** should be preceded by **h** if the corresponding argument is a pointer to **short int** rather than a pointer to **int**, or by **l** if it is a pointer to **long int**, or by **ll** if it is a pointer to **long long int**. Similarly, the conversion specifiers **o**, **u**, and **x** shall be preceded by **h** if the corresponding argument is a pointer to **unsigned short int** rather than a pointer to **unsigned int**, or by **l** if it is a pointer to **unsigned long int**, or by **ll** if it is a pointer to **unsigned long long int**. Finally, the conversion specifiers **e**, **f**, and **g** shall be preceded by **l** if the corresponding argument is a pointer to **double** rather than a pointer to **float**. If a **h**, **l** or **ll** appears with any other conversion specifier, the behavior is undefined.

WARNING: ANSI C also specifies an optional **L** in some of the same contexts as **l** above, corresponding to a **long double** * argument. However, the current release of the VxWorks libraries does not support **long double** data; using the optional **L** gives unpredictable results.

S

– A character that specifies the type of conversion to be applied. The

valid conversion specifiers are described below.

The **sscanf( )** routine executes each directive of the format in turn. If a directive fails, as detailed below, **sscanf( )** returns. Failures are described as input failures (due to the unavailability of input characters), or matching failures (due to inappropriate input).

A directive composed of white-space character(s) is executed by reading input up to the first non-white-space character (which remains unread), or until no more characters can be read.

A directive that is an ordinary multibyte character is executed by reading the next characters of the stream. If one of the characters differs from one comprising the directive, the directive fails, and the differing and subsequent characters remain unread.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

Input white-space characters (as specified by the **isspace( )** function) are skipped, unless the specification includes a **[**, **c**, or **n** specifier.

An input item is read from the stream, unless the specification includes an **n** specifier. An input item is defined as the longest matching sequence of input characters, unless that exceeds a specified field width, in which case it is the initial subsequence of that length in the sequence. The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails: this condition is a matching failure, unless an error prevented input from the stream, in which case it is an input failure.

Except in the case of a % specifier, the input item is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a *, the result of the conversion is placed in the object pointed to by the first argument following the *fmt* argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

The following conversion specifiers are valid:

d

Matches an optionally signed decimal integer whose format is the same as expected for the subject sequence of the **strtol( )** function with the value 10 for the *base* argument. The corresponding argument should be a pointer to **int**.

i

Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the **strtol( )** function with the value 0 for the *base* argument. The corresponding argument should be a pointer to **int**.

o

Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the **strtoul( )** function with the value 8 for the *base* argument. The corresponding argument should be a pointer to **unsigned int**.

u

Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **strtoul( )** function with the value 10 for the *base* argument. The corresponding argument should be a pointer to **unsigned int**.

x

Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the **strtoul( )** function with the value 16 for the *base* argument. The corresponding argument should be a pointer to **unsigned int**.

e, **f**, **g**

Match an optionally signed floating-point number, whose format is the same as expected for the subject string of the **strtod( )** function. The corresponding argument should be a pointer to **float**.

s

Matches a sequence of non-white-space characters. The corresponding argument should be a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which will be added automatically.

[

Matches a non-empty sequence of characters from a set of expected characters (the **scanset**). The corresponding argument should be a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which is added automatically. The conversion specifier includes all subsequent character in the format string, up to and including the matching right bracket (**]**). The characters between the brackets (the **scanlist**) comprise the scanset, unless the character after the left bracket is a circumflex (**^**) in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with "[]" or "[^]", the right bracket character is in the scanlist and the next right bracket character is the matching right bracket that ends the specification; otherwise the first right bracket character is the one that ends the specification.

c

Matches a sequence of characters of the number specified by the field width (1 if no field width is present in the directive). The corresponding argument should be a pointer to the initial character of an array large enough to accept the sequence. No null character is added.

p

Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the %p conversion of the **fprintf( )** function. The corresponding argument should be a pointer to a pointer to **void**.

VxWorks defines its pointer input field to be consistent with pointers written by the **fprintf( )** function (“0x” hexadecimal notation). If the input item is a value converted earlier during the same program execution, the pointer that results should compare equal to that value; otherwise the behavior of the %p conversion is undefined.

n

No input is consumed. The corresponding argument should be a pointer to **int** into which the number of characters read from the input stream so far by this call to **sscanf( )** is written. Execution of a %n directive does not increment the assignment count returned when **sscanf( )** completes execution.

%

Matches a single **%**; no conversion or assignment occurs. The complete conversion specification is %%.

If a conversion specification is invalid, the behavior is undefined.

The conversion specifiers **E**, **G**, and **X** are also valid and behave the same as **e**, **g**, and **x**, respectively.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (if any) is terminated with an input failure.

If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream. Trailing white space (including new-line characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the %n directive.

INCLUDE FILES **fioLib.h**

RETURNS The number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure; or EOF if an input failure occurs before any conversion.

SEE ALSO **fioLib**, **fscanf( )**, **scanf( )**, *American National Standard for Information Systems -Programming Language - C, ANSI X3.159-1989: Input/Output (**stdio.h**)*

stackEntryIsBottom( )

NAME

stackEntryIsBottom( ) – test if an interface has no layers beneath it

SYNOPSIS

```
BOOL stackEntryIsBottom
    (
    int index                /* interface to examine */
    )
```

DESCRIPTION

This routine returns **TRUE** if an interface has no layers beneath it. This helper function is not exported.

RETURNS

TRUE if the interface is the bottom-most layer in a stack
FALSE otherwise or on error

SEE ALSO

m2IfLib

stackEntryIsTop( )

NAME

stackEntryIsTop( ) – test if an ifStackTable interface has no layers above

SYNOPSIS

```
BOOL stackEntryIsTop
    (
    int index                /* the interface to examine */
    )
```

DESCRIPTION

This routine returns **TRUE** if an interface is not below any other interface. That is, it returns **TRUE** if the given interface is topmost on a stack. This helper function is not exported.

RETURNS

TRUE is interface is topmost
FALSE otherwise or for errors

SEE ALSO

m2IfLib

S

stat( )

NAME

stat( ) – get file status information using a pathname (POSIX)

SYNOPSIS

```
STATUS stat
    (
    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.

On return, the fields in the **stat** structure are updated to reflect the characteristics of the file.

RETURNS

OK or **ERROR**.

SEE ALSO

dirLib, **fstat( )**, **ls( )**

statfs( )

NAME

statfs( ) – get file status information using a pathname (POSIX)

SYNOPSIS

```
STATUS statfs
    (
    char *           name,     /* name of file to check */
    struct statfs * pStat      /* pointer to statfs structure */
    )
```

DESCRIPTION

This routine obtains various characteristics of a file system. This routine is equivalent to **fstatfs( )**, except that the *name* of the file is specified, rather than an open file descriptor.

The *pStat* parameter is a pointer to a **statfs** structure (defined in **stat.h**). This structure must have already been allocated before this routine is called.

Upon return, the fields in the **statfs** structure are updated to reflect the characteristics of the file.

RETURNS **OK** or **ERROR**.

SEE ALSO **dirLib**, **fstatfs( )**, **ls( )**

---

stdioFp( )

NAME **stdioFp( )** – return the standard input/output/error FILE of the current task

SYNOPSIS

```
FILE * stdioFp
    (
    int stdFd                /* fd of standard FILE to return (0,1,2) */
    )
```

DESCRIPTION This routine returns the specified standard FILE structure address of the current task. It is provided primarily to give access to standard input, standard output, and standard error from the shell, where the usual **stdin**, **stdout**, **stderr** macros cannot be used.

INCLUDE FILES **stdio.h**

RETURNS The standard FILE structure address of the specified file descriptor, for the current task.

SEE ALSO **ansiStdio**

S

---

stdioInit( )

NAME **stdioInit( )** – initialize standard I/O support

SYNOPSIS

```
STATUS stdioInit (void)
```

DESCRIPTION This routine installs standard I/O support. It must be called before using **stdio** buffering. If **INCLUDE_STDIO** is defined in **configAll.h**, it is called automatically by the root task **usrRoot( )** in **usrConfig.c**.

RETURNS **OK**, or **ERROR** if the standard I/O facilities cannot be installed.

SEE ALSO **ansiStdio**

stdioShow( )

NAME **stdioShow( )** – display file pointer internals

SYNOPSIS

```
STATUS stdioShow
    (
    FILE * fp,              /* stream */
    int    level            /* level */
    )
```

DESCRIPTION This routine displays information about a specified stream.

RETURNS **OK**, or **ERROR** if the file pointer is invalid.

SEE ALSO **ansiStdio**

stdioShowInit( )

NAME **stdioShowInit( )** – initialize the standard I/O show facility

SYNOPSIS

```
STATUS stdioShowInit (void)
```

DESCRIPTION This routine links the file pointer show routine into the VxWorks system. It is called automatically when this show facility is configured into VxWorks using either of the following methods:

– If you use the configuration header files, define **INCLUDE_SHOW_ROUTINES** in **config.h**.

– If you use the Tornado project facility, select **INCLUDE_STDIO_SHOW**.

RETURNS **OK**, or **ERROR** if an error occurs installing the file pointer show routine.

SEE ALSO **ansiStdio**

strcat( )

NAME **strcat( )** – concatenate one string to another (ANSI)

SYNOPSIS

```
char * strcat
    (
    char *       destination, /* string to be appended to */
    const char * append       /* string to append to destination */
    )
```

DESCRIPTION This routine appends a copy of string *append* to the end of string *destination*. The resulting string is null-terminated.

INCLUDE FILES **string.h**

RETURNS A pointer to *destination*.

SEE ALSO **ansiString**

strchr( )

NAME **strchr( )** – find the first occurrence of a character in a string (ANSI)

SYNOPSIS

```
char * strchr
    (
    const char * s,           /* string in which to search */
    int          c            /* character to find in string */
    )
```

DESCRIPTION This routine finds the first occurrence of character *c* in string *s*. The terminating null is considered to be part of the string.

INCLUDE FILES **string.h**

RETURNS The address of the located character, or **NULL** if the character is not found.

SEE ALSO **ansiString**

S

strcmp( )

NAME

strcmp( ) – compare two strings lexicographically (ANSI)

SYNOPSIS

```
int strcmp
    (
    const char * s1,          /* string to compare */
    const char * s2           /* string to compare s1 to */
    )
```

DESCRIPTION

This routine compares string *s1* to string *s2* lexicographically.

INCLUDE FILES

string.h

RETURNS

An integer greater than, equal to, or less than 0, according to whether *s1* is lexicographically greater than, equal to, or less than *s2*, respectively.

SEE ALSO

ansiString

strcoll( )

NAME

strcoll( ) – compare two strings as appropriate to **LC_COLLATE** (ANSI)

SYNOPSIS

```
int strcoll
    (
    const char * s1,          /* string 1 */
    const char * s2           /* string 2 */
    )
```

DESCRIPTION

This routine compares two strings, both interpreted as appropriate to the **LC_COLLATE** category of the current locale.

INCLUDE FILES

string.h

RETURNS

An integer greater than, equal to, or less than zero, according to whether string *s1* is greater than, equal to, or less than string *s2* when both are interpreted as appropriate to the current locale.

SEE ALSO

ansiString

strcpy( )

NAME

strcpy( ) – copy one string to another (ANSI)

SYNOPSIS

```
char * strcpy
    (
    char *       s1,          /* string to copy to */
    const char * s2           /* string to copy from */
    )
```

DESCRIPTION

This routine copies string *s2* (including EOS) to string *s1*.

INCLUDE FILES

string.h

RETURNS

A pointer to *s1*.

SEE ALSO

ansiString

strcspn( )

NAME

strcspn( ) – return the string length up to the first character from a given set (ANSI)

SYNOPSIS

```
size_t strcspn
    (
    const char * s1,          /* string to search */
    const char * s2           /* set of characters to look for in s1 */
    )
```

DESCRIPTION

This routine computes the length of the maximum initial segment of string *s1* that consists entirely of characters not included in string *s2*.

INCLUDE FILES

string.h

RETURNS

The length of the string segment.

SEE ALSO

ansiString, **strpbrk( )**, **strspn( )**

strerror( )

NAME

strerror( ) – map an error number to an error string (ANSI)

SYNOPSIS

```
char * strerror
    (
    int errcode                  /* error code */
    )
```

DESCRIPTION

This routine maps the error number in *errcode* to an error message string. It returns a pointer to a static buffer that holds the error string.

This routine is not reentrant. For a reentrant version, see **strerror_r( )**.

INCLUDE

string.h

RETURNS

A pointer to the buffer that holds the error string.

SEE ALSO

ansiString, **strerror_r( )**

strerror_r( )

NAME

strerror_r( ) – map an error number to an error string (POSIX)

SYNOPSIS

```
STATUS strerror_r
    (
    int    errcode,              /* error number */
    char * buffer                /* string buffer */
    )
```

DESCRIPTION

This call maps the error code in *errcode* to an error message string which it stores in *buffer*.

This routine is the POSIX reentrant version of **strerror( )**.

INCLUDE FILES

string.h

RETURNS

OK or **ERROR**.

SEE ALSO

ansiString, **strerror( )**

strftime( )

NAME

strftime( ) – convert broken-down time into a formatted string (ANSI)

SYNOPSIS

```
size_t strftime
    (
    char *            s,      /* string array */
    size_t            n,      /* maximum size of array */
    const char *      format, /* format of output string */
    const struct tm * tptr    /* broken-down time */
    )
```

DESCRIPTION

This routine formats the broken-down time in *tptr* based on the conversion specified in the string *format*, and places the result in the string *s*.

The format is a multibyte character sequence, beginning and ending in its initial state. The *format* string consists of zero or more conversion specifiers and ordinary multibyte characters. A conversion specifier consists of a % character followed by a character that determines the behavior of the conversion. All ordinary multibyte characters (including the terminating **NULL** character) are copied unchanged to the array. If copying takes place between objects that overlap, the behavior is undefined. No more than *n* characters are placed into the array.

Each conversion specifier is replaced by appropriate characters as described in the following list. The appropriate characters are determined by the **LC_TIME** category of the current locale and by the values contained in the structure pointed to by *tptr*.

%a
: the locale's abbreviated weekday name.

%A
: the locale's full weekday name.

%b
: the locale's abbreviated month name.

%B
: the locale's full month name.

%c
: the locale's appropriate date and time representation.

%d
: the day of the month as decimal number (01-31).

%H
: the hour (24-hour clock) as a decimal number (00-23).

%I
: the hour (12-hour clock) as a decimal number (01-12).

%j
: the day of the year as decimal number (001-366).

%m
: the month as a decimal number (01-12).

%M
: the minute as a decimal number (00-59).

%P
: the locale's equivalent of the AM/PM designations associated with a 12-hour clock.

%S
: the second as a decimal number (00-59).

%U
: the week number of the year (first Sunday as first day of week 1) as a decimal number (00-53).

%w
: the weekday as a decimal number (0-6), where Sunday is 0.

%W
: the week number of the year (the first Monday as the first day of week 1) as a decimal number (00-53).

%x
: the locale's appropriate date representation.

%X
: the locale's appropriate time representation.

%y
: the year without century as a decimal number (00-99).

%Y
: the year with century as a decimal number.

%Z
: the time zone name or abbreviation, or by no characters if no time zone is determinable.

%%
: %.

For any other conversion specifier, the behavior is undefined.

INCLUDE FILES **time.h**

RETURNS The number of characters in *s*, not including the terminating null character -- or zero if the number of characters in *s*, including the null character, is more than *n* (in which case the contents of *s* are indeterminate).

SEE ALSO **ansiTime**

strlen( )

NAME **strlen( )** – determine the length of a string (ANSI)

SYNOPSIS

```
size_t strlen
    (
    const char * s            /* string */
    )
```

DESCRIPTION This routine returns the number of characters in *s*, not including EOS.

INCLUDE FILES **string.h**

RETURNS The number of non-null characters in the string.

SEE ALSO **ansiString**

strncat( )

NAME **strncat( )** – concatenate characters from one string to another (ANSI)

SYNOPSIS

```
char * strncat
    (
    char *       dst,         /* string to append to */
    const char * src,         /* string to append */
    size_t       n            /* max no. of characters to append */
    )
```

DESCRIPTION This routine appends up to *n* characters from string *src* to the end of string *dst*.

INCLUDE FILES **string.h**

S

RETURNS A pointer to the null-terminated string *s1*.

SEE ALSO **ansiString**

strncmp( )

NAME **strncmp( )** – compare the first *n* characters of two strings (ANSI)

SYNOPSIS

```
int strncmp
    (
    const char * s1,          /* string to compare */
    const char * s2,          /* string to compare s1 to */
    size_t       n            /* max no. of characters to compare */
    )
```

DESCRIPTION This routine compares up to *n* characters of string *s1* to string *s2*lexicographically.

INCLUDE FILES **string.h**

RETURNS An integer greater than, equal to, or less than 0, according to whether *s1* is lexicographically greater than, equal to, or less than *s2*, respectively.

SEE ALSO **ansiString**

strncpy( )

NAME **strncpy( )** – copy characters from one string to another (ANSI)

SYNOPSIS

```
char *strncpy
    (
    char *       s1,          /* string to copy to */
    const char * s2,          /* string to copy from */
    size_t       n            /* max no. of characters to copy */
    )
```

DESCRIPTION This routine copies *n* characters from string *s2* to string *s1*. If *n* is greater than the length of *s2*, nulls are added to *s1*. If *n* is less than or equal to the length of *s2*, the target string will not be null-terminated.

INCLUDE FILES **string.h**

RETURNS A pointer to *s1*.

SEE ALSO **ansiString**

---

strpbrk( )

NAME **strpbrk( )** – find the first occurrence in a string of a character from a given set (ANSI)

SYNOPSIS

```
char * strpbrk
    (
    const char * s1,          /* string to search */
    const char * s2           /* set of characters to look for in s1 */
    )
```

DESCRIPTION This routine locates the first occurrence in string *s1* of any character from string *s2*.

INCLUDE FILES **string.h**

RETURNS A pointer to the character found in *s1*, or **NULL** if no character from *s2* occurs in *s1*.

SEE ALSO **ansiString**, **strcspn( )**

S

---

strrchr( )

NAME **strrchr( )** – find the last occurrence of a character in a string (ANSI)

SYNOPSIS

```
char * strrchr
    (
    const char * s,           /* string to search */
    int          c            /* character to look for */
    )
```

DESCRIPTION This routine locates the last occurrence of *c* in the string pointed to by *s*. The terminating null is considered to be part of the string.

INCLUDE FILES **string.h**

RETURNS A pointer to the last occurrence of the character, or **NULL** if the character is not found.

SEE ALSO **ansiString**

strspn( )

NAME **strspn( )** – return the string length up to the first character not in a given set (ANSI)

SYNOPSIS

```
size_t strspn
    (
    const char * s,           /* string to search */
    const char * sep          /* set of characters to look for in s */
    )
```

DESCRIPTION This routine computes the length of the maximum initial segment of string *s* that consists entirely of characters from the string *sep*.

INCLUDE FILES **string.h**

RETURNS The length of the string segment.

SEE ALSO **ansiString**, **strcspn( )**

strstr( )

NAME **strstr( )** – find the first occurrence of a substring in a string (ANSI)

SYNOPSIS

```
char * strstr
    (
    const char * s,           /* string to search */
    const char * find         /* substring to look for */
    )
```

DESCRIPTION This routine locates the first occurrence in string *s* of the sequence of characters (excluding the terminating null character) in the string *find*.

INCLUDE FILES **string.h**

RETURNS A pointer to the located substring, or *s* if *find* points to a zero-length string, or **NULL** if the string is not found.

SEE ALSO **ansiString**

strtod( )

NAME **strtod( )** – convert the initial portion of a string to a double (ANSI)

SYNOPSIS

```
double strtod
    (
    const char * s,           /* string to convert */
    char * *     endptr       /* ptr to final string */
    )
```

DESCRIPTION This routine converts the initial portion of a specified string *s* to a double. First, it decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the **isspace( )** function); a subject sequence resembling a floating-point constant; and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then, it attempts to convert the subject sequence to a floating-point number, and returns the result.

The expected form of the subject sequence is an optional plus or minus decimal-point character, then an optional exponent part but no floating suffix. The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign, a digit, or a decimal-point character.

If the subject sequence has the expected form, the sequence of characters starting with the first digit or the decimal-point character (whichever occurs first) is interpreted as a floating constant, except that the decimal-point character is used in place of a period, and that if neither an exponent part nor a decimal-point character appears, a decimal point is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the value resulting form the conversion is negated. A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

S

In other than the "C" locale, additional implementation-defined subject sequence forms may be accepted. VxWorks supports only the "C" locale.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *s* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

INCLUDE FILES **stdlib.h**

RETURNS The converted value, if any. If no conversion could be performed, it returns zero. If the correct value is outside the range of representable values, it returns plus or minus **HUGE_VAL** (according to the sign of the value), and stores the value of the macro **ERANGE** in **errno**. If the correct value would cause underflow, it returns zero and stores the value of the macro **ERANGE** in **errno**.

SEE ALSO **ansiStdlib**

strtok( )

NAME **strtok( )** – break down a string into tokens (ANSI)

SYNOPSIS

```
char * strtok
    (
    char *       string,      /* string */
    const char * separator    /* separator indicator */
    )
```

DESCRIPTION A sequence of calls to this routine breaks the string *string* into a sequence of tokens, each of which is delimited by a character from the string *separator*. The first call in the sequence has *string* as its first argument, and is followed by calls with a null pointer as their first argument. The separator string may be different from call to call.

The first call in the sequence searches *string* for the first character that is not contained in the current separator string. If the character is not found, there are no tokens in *string* and **strtok( )** returns a null pointer. If the character is found, it is the start of the first token.

strtok( ) then searches from there for a character that is contained in the current separator string. If the character is not found, the current token expands to the end of the string pointed to by *string*, and subsequent searches for a token will return a null pointer. If the character is found, it is overwritten by a null character, which terminates the current token. **strtok( )** saves a pointer to the following character, from which the next search for a token will start. (Note that because the separator character is overwritten by a null character, the input string is modified as a result of this call.)

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

The implementation behaves as if **strtok( )** is called by no library functions.

REENTRANCY This routine is not reentrant; the reentrant form is **strtok_r( )**.

INCLUDE FILES **string.h**

RETURNS A pointer to the first character of a token, or a **NULL** pointer if there is no token.

SEE ALSO **ansiString**, **strtok_r( )**

---

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.

INCLUDE FILES **string.h**

S

RETURNS A pointer to the first character of a token, or a **NULL** pointer if there is no token.

SEE ALSO **ansiString**, **strtok( )**

strtol( )

NAME **strtol( )** – convert a string to a long integer (ANSI)

SYNOPSIS

```
long strtol
    (
    const char * nptr,        /* string to convert */
    char * *     endptr,      /* ptr to final string */
    int          base         /* radix */
    )
```

DESCRIPTION This routine converts the initial portion of a string *nptr* to **long int** representation. First, it decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by **isspace( )**); a subject sequence resembling an integer represented in some radix determined by the value of *base*; and a final string of one or more unrecognized characters, including the terminating **NULL** character of the input string. Then, it attempts to convert the subject sequence to an integer number, and returns the result.

If the value of *base* is zero, the expected form of the subject sequence is that of an integer constant, optionally preceded by a plus or minus sign, but not including an integer suffix. If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base* optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from a (or A) through to z (or Z) are ascribed the values 10 to 35; only letters whose ascribed values are less than *base* are permitted. If the value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of *base* is zero, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it is used as the *base* for conversion, ascribing to each latter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a **NULL** pointer.

In other than the "C" locale, additional implementation-defined subject sequence forms may be accepted. VxWorks supports only the "C" locale; it assumes that the upper- and lower-case alphabets and digits are each contiguous.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a **NULL** pointer.

INCLUDE FILES **stdlib.h**

RETURNS The converted value, if any. If no conversion could be performed, it returns zero. If the correct value is outside the range of representable values, it returns **LONG_MAX** or **LONG_MIN** (according to the sign of the value), and stores the value of the macro **ERANGE** in **errno**.

SEE ALSO **ansiStdlib**

strtoul( )

NAME **strtoul( )** – convert a string to an unsigned long integer (ANSI)

SYNOPSIS

```
ulong_t strtoul
    (
    const char * nptr,        /* string to convert */
    char * *     endptr,      /* ptr to final string */
    int          base         /* radix */
    )
```

DESCRIPTION This routine converts the initial portion of a string *nptr* to **unsigned long int** representation. First, it decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by **isspace( )**); a subject sequence resembling an unsigned integer represented in some radix determined by the value *base*; and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then, it attempts to convert the subject sequence to an unsigned integer, and returns the result.

If the value of *base* is zero, the expected form of the subject sequence is that of an integer constant, optionally preceded by a plus or minus sign, but not including an integer suffix. If the value of *base* is between 2 and 36, the expected form of the subject sequence is a

S

sequence of letters and digits representing an integer with the radix specified by letters from a (or A) through z (or Z) which are ascribed the values 10 to 35; only letters whose ascribed values are less than *base* are permitted. If the value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of *base* is zero, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it is used as the *base* for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

In other than the "C" locale, additional implementation-defined subject sequence forms may be accepted. VxWorks supports only the "C" locale; it assumes that the upper- and lower-case alphabets and digits are each contiguous.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

INCLUDE FILES

stdlib.h

RETURNS

The converted value, if any. If no conversion could be performed it returns zero. If the correct value is outside the range of representable values, it returns **ULONG_MAX**, and stores the value of the macro **ERANGE** in **errno**.

SEE ALSO

ansiStdlib

strxfrm( )

NAME

strxfrm( ) – transform up to *n* characters of *s2* into *s1* (ANSI)

SYNOPSIS

```
size_t strxfrm
    (
    char *       s1,          /* string out */
    const char * s2,          /* string in */
    size_t       n            /* size of buffer */
    )
```

DESCRIPTION

This routine transforms string *s2* and places the resulting string in *s1*. The transformation is such that if **strcmp( )** is applied to two transformed strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the **strcoll( )** function applied to the same two original strings. No more than *n* characters are placed into the resulting *s1*, including the terminating null character. If *n* is zero, *s1* is permitted to be a **NULL** pointer. If copying takes place between objects that overlap, the behavior is undefined.

INCLUDE FILES

string.h

RETURNS

The length of the transformed string, not including the terminating null character. If the value is *n* or more, the contents of *s1* are indeterminate.

SEE ALSO

ansiString, **strcmp( )**, **strcoll( )**

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.

It is an error for *nbytes* to be odd.

RETURNS 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 *name* and adds it to a specified symbol table *symTblId* with the specified parameters *value*, *type*, and *group*. 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 or there is insufficient memory for the symbol to be allocated.

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 value and type (*value* and *sType*). If there is no matching entry, it chooses the table entry with the next lower value (among entries with the same type). A pointer to the symbol name string (with terminating EOS) is returned into *pName*. The actual value and the type are copied to *pValue* and *pType*. The *mask* parameter can be used to match sub-classes of type.

pName is a pointer to memory allocated by **symByValueAndTypeFind( )**; the memory must be freed by the caller after the use of *pName*.

To search the global VxWorks symbol table, specify **sysSymTbl** as *symTblId*.

RETURNS

OK or **ERROR** if *symTblId* is invalid, *pName* is **NULL**, or *value* is less than the lowest value in the table.

SEE ALSO

symLib

S

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 matching a specified value. If there is no matching entry, it chooses the table entry with the next lower value. A pointer to a copy of the symbol name string (with terminating EOS) is returned into *pName*. The actual value and the type are copied to *pValue* and *pType*.

pName is a pointer to memory allocated by symByValueFind; the memory must be freed by the caller after the use of *pName*.

To search the global VxWorks symbol table, specify **sysSymTbl** as *symTblId*.

RETURNS

OK or **ERROR** if *symTblId* is invalid, *pName* is **NULL**, or *value* is less than the lowest value in the table.

SEE ALSO

symLib

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,  /* entry name                   */
    int       val,    /* value associated with entry */
    SYM_TYPE  type,   /* entry type                   */
    int       arg,    /* arbitrary user-supplied arg */
    UINT16    group   /* 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.

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 put symbol value */
    SYM_TYPE * pType          /* where to put symbol type */
    )
```

DESCRIPTION This routine searches a symbol table for a symbol matching a specified name. If the symbol is found, its value and type are copied to *pValue* and *pType*. If multiple symbols have the same name but differ in type, the routine chooses the matching symbol most recently added to the symbol table.

To search the global VxWorks symbol table, specify **sysSymTbl** as *symTblId*.

RETURNS **OK**, or **ERROR** if the symbol table ID is invalid or the symbol cannot be found.

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 matching both name and type (*name* and *sType*). If the symbol is found, its value and type are copied to *pValue* and *pType*. The *mask* parameter can be used to match sub-classes of type.

To search the global VxWorks symbol table, specify **sysSymTbl** as *symTblId*.

RETURNS

OK, or **ERROR** if the symbol table ID is invalid or the symbol is not found.

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( )**.

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 lower 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 *symTblId*.

RETURNS **OK**, or **ERROR** if *symTblId* is invalid or *value* is less than the lowest value in the table.

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( )**.

This routine searches a symbol table for a symbol matching both value and type (*value* and *sType*). If there is no matching entry, it chooses the table entry with the next lower value. The symbol name (with terminating EOS), the actual value, and the type are copied to *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 *symTblId*.

RETURNS **OK**, or **ERROR** if *symTblId* is invalid or *value* is less than the lowest value in the table. *

SEE ALSO **symLib**

symLibInit( )

NAME **symLibInit( )** – initialize the symbol table library

SYNOPSIS

```
STATUS symLibInit (void)
```

DESCRIPTION This routine initializes the symbol table package. If the configuration macro **INCLUDE_SYM_TBL** is defined, **symLibInit( )** is called by the root task, **usrRoot( )**, in **usrConfig.c**.

RETURNS **OK**, or **ERROR** if the library could not be initialized.

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 of 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.

SEE ALSO **symLib**

symSyncLibInit( )

NAME **symSyncLibInit( )** – initialize host/target symbol table synchronization

SYNOPSIS

```
void symSyncLibInit ()
```

DESCRIPTION This routine initializes host/target symbol table synchronization. To enable synchronization, it must be called before a target server is started. It is called automatically if the configuration macro **INCLUDE_SYM_TBL_SYNC** is defined.

RETURNS N/A

SEE ALSO **symSyncLib**

symSyncTimeoutSet( )

NAME **symSyncTimeoutSet( )** – set WTX timeout

SYNOPSIS

```
UINT32 symSyncTimeoutSet
    (
    UINT32 timeout            /* WTX timeout in milliseconds */
    )
```

DESCRIPTION This routine sets the WTX timeout between target server and synchronization task.

RETURNS If *timeout* is 0, the current timeout, otherwise the new timeout value in milliseconds.

SEE ALSO **symSyncLib**

S

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 *sameNameOk* is **FALSE**, attempting to add a symbol with the same name and type as an already-existing symbol results in an error.

Memory for storing symbols as they are added to the symbol table will be allocated from the memory partition *symPartId*. The ID of the system memory partition is stored in the global variable **memSysPartId**, which is declared in **memLib.h**.

RETURNS

Symbol table ID, or **NULL** if memory is insufficient.

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.

Deletion of a table that still contains symbols results in **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.

SEE ALSO **symLib**

sysAuxClkConnect( )

NAME **sysAuxClkConnect( )** – connect a routine to the auxiliary clock interrupt

SYNOPSIS

```
STATUS sysAuxClkConnect
    (
    FUNCPTR routine,  /* routine called at each aux clock interrupt */
    int     arg       /* argument to auxiliary clock interrupt routine */
    )
```

DESCRIPTION This routine specifies the interrupt service routine to be called at each auxiliary clock interrupt. It does not enable auxiliary clock interrupts.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS **OK**, or **ERROR** if the routine cannot be connected to the interrupt.

SEE ALSO **sysLib**, **intConnect( )**, **sysAuxClkEnable( )**, and BSP-specific reference pages for this routine.

S

sysAuxClkDisable( )

NAME **sysAuxClkDisable( )** – turn off auxiliary clock interrupts

SYNOPSIS

```
void sysAuxClkDisable (void)
```

DESCRIPTION This routine disables auxiliary clock interrupts.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS N/A

SEE ALSO **sysLib**, **sysAuxClkEnable( )**, and BSP-specific reference pages for this routine.

sysAuxClkEnable( )

NAME **sysAuxClkEnable( )** – turn on auxiliary clock interrupts

SYNOPSIS

```
void sysAuxClkEnable (void)
```

DESCRIPTION This routine enables auxiliary clock interrupts.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS N/A

SEE ALSO **sysLib**, **sysAuxClkConnect( )**, **sysAuxClkDisable( )**, **sysAuxClkRateSet( )**, and BSP-specific reference pages for this routine.

sysAuxClkRateGet( )

NAME **sysAuxClkRateGet( )** – get the auxiliary clock rate

SYNOPSIS

```
int sysAuxClkRateGet (void)
```

DESCRIPTION This routine returns the interrupt rate of the auxiliary clock.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS The number of ticks per second of the auxiliary clock.

SEE ALSO **sysLib**, **sysAuxClkEnable( )**, **sysAuxClkRateSet( )**, and BSP-specific reference pages for this routine.

sysAuxClkRateSet( )

NAME
sysAuxClkRateSet( ) – set the auxiliary clock rate

SYNOPSIS

```
STATUS sysAuxClkRateSet
    (
    int ticksPerSecond        /* number of clock interrupts per second */
    )
```

DESCRIPTION
This routine sets the interrupt rate of the auxiliary clock. It does not enable auxiliary clock interrupts.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS
OK, or **ERROR** if the tick rate is invalid or the timer cannot be set.

SEE ALSO
sysLib, **sysAuxClkEnable( )**, **sysAuxClkRateGet( )**, and BSP-specific reference pages for this routine.

sysBspRev( )

NAME
sysBspRev( ) – return the BSP version and revision number

SYNOPSIS

```
char * sysBspRev (void)
```

DESCRIPTION
This routine returns a pointer to a BSP version and revision number, for example, 1.0/1. **BSP_REV** is concatenated to **BSP_VERSION** and returned.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS
A pointer to the BSP version/revision string.

SEE ALSO
sysLib, and BSP-specific reference pages for this routine.

S

sysBusIntAck( )

NAME **sysBusIntAck( )** – acknowledge a bus interrupt

SYNOPSIS

```
int sysBusIntAck
    (
    int intLevel              /* interrupt level to acknowledge */
    )
```

DESCRIPTION This routine acknowledges a specified VME bus interrupt level.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS NULL.

SEE ALSO **sysLib**, **sysBusIntGen( )**, and BSP-specific reference pages for this routine.

sysBusIntGen( )

NAME **sysBusIntGen( )** – generate a bus interrupt

SYNOPSIS

```
STATUS sysBusIntGen
    (
    int intLevel,             /* bus interrupt level to generate */
    int vector                /* interrupt vector to generate (0-255) */
    )
```

DESCRIPTION This routine generates a bus interrupt for a specified level with a specified vector.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS **OK**, or **ERROR** if *intLevel* is out of range or the board cannot generate a bus interrupt.

SEE ALSO **sysLib**, **sysBusIntAck( )**, and BSP-specific reference pages for this routine.

sysBusTas( )

NAME

sysBusTas( ) – test and set a location across the bus

SYNOPSIS

```
BOOL sysBusTas
    (
    char * adrs                /* address to be tested and set */
    )
```

DESCRIPTION

This routine performs a test-and-set instruction across the backplane.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

NOTE: This routine is equivalent to **vxTas( )**.

RETURNS

TRUE if the value had not been set but is now, or **FALSE** if the value was set already.

SEE ALSO

sysLib, **vxTas( )**, and BSP-specific reference pages for this routine.

sysBusToLocalAdrs( )

NAME

sysBusToLocalAdrs( ) – convert a bus address to a local address

SYNOPSIS

```
STATUS sysBusToLocalAdrs
    (
    int    adrsSpace,        /* bus address space in which busAdrs resides */
    char * busAdrs,          /* bus address to convert */
    char * *pLocalAdrs       /* where to return local address */
    )
```

DESCRIPTION

This routine gets the local address that accesses a specified bus memory address.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS

OK, or **ERROR** if the address space is unknown or the mapping is not possible.

S

SEE ALSO **sysLib**, **sysLocalToBusAdrs( )**, and BSP-specific reference pages for this routine.

sysClkConnect( )

NAME **sysClkConnect( )** – connect a routine to the system clock interrupt

SYNOPSIS

```
STATUS sysClkConnect
    (
    FUNCPTR routine,          /* routine called at each system clock */
                              /* interrupt */
    int     arg               /* argument with which to call routine */
    )
```

DESCRIPTION This routine specifies the interrupt service routine to be called at each clock interrupt. Normally, it is called from **usrRoot( )** in **usrConfig.c** to connect **usrClock( )** to the system clock interrupt.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference entries for your BSP.

RETURN **OK**, or **ERROR** if the routine cannot be connected to the interrupt.

SEE ALSO **sysLib**, **intConnect( )**, **usrClock( )**, **sysClkEnable( )**, and BSP-specific reference pages for this routine.

sysClkDisable( )

NAME **sysClkDisable( )** – turn off system clock interrupts

SYNOPSIS

```
void sysClkDisable (void)
```

DESCRIPTION This routine disables system clock interrupts.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS N/A

SEE ALSO **sysLib**, **sysClkEnable( )**, and BSP-specific reference pages for this routine.

sysClkEnable( )

NAME **sysClkEnable( )** – turn on system clock interrupts

SYNOPSIS

```
void sysClkEnable (void)
```

DESCRIPTION This routine enables system clock interrupts.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS N/A

SEE ALSO **sysLib**, **sysClkConnect( )**, **sysClkDisable( )**, **sysClkRateSet( )**, and BSP-specific reference pages for this routine.

sysClkRateGet( )

NAME **sysClkRateGet( )** – get the system clock rate

SYNOPSIS

```
int sysClkRateGet (void)
```

DESCRIPTION This routine returns the system clock rate.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS The number of ticks per second of the system clock.

SEE ALSO **sysLib**, **sysClkEnable( )**, **sysClkRateSet( )**, and BSP-specific reference pages for this routine.

sysClkRateSet( )

NAME

sysClkRateSet( ) – set the system clock rate

SYNOPSIS

```
STATUS sysClkRateSet
    (
    int ticksPerSecond        /* number of clock interrupts per second */
    )
```

DESCRIPTION

This routine sets the interrupt rate of the system clock. It is called by **usrRoot( )** in **usrConfig.c**.

There may be interactions between this routine and the POSIX **clockLib** routines. Refer to the **clockLib** reference entry.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS

OK, or **ERROR** if the tick rate is invalid or the timer cannot be set.

SEE ALSO

sysLib, **sysClkEnable( )**, **sysClkRateGet( )**, **clockLib**, and BSP-specific reference pages for this routine.

sysHwInit( )

NAME

sysHwInit( ) – initialize the system hardware

SYNOPSIS

```
void sysHwInit (void)
```

DESCRIPTION

This routine initializes various features of the board. It is called from **usrInit( )** in **usrConfig.c**.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

NOTE: This routine should not be called directly by the user application.

RETURNS N/A

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine.

sysIntDisable( )

NAME **sysIntDisable( )** – disable a bus interrupt level

SYNOPSIS

```
STATUS sysIntDisable
    (
    int intLevel              /* interrupt level to disable */
    )
```

DESCRIPTION This routine disables a specified bus interrupt level.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS **OK**, or **ERROR** if *intLevel* is out of range.

SEE ALSO **sysLib**, **sysIntEnable( )**, and BSP-specific reference pages for this routine.

sysIntEnable( )

NAME **sysIntEnable( )** – enable a bus interrupt level

SYNOPSIS

```
STATUS sysIntEnable
    (
    int intLevel              /* interrupt level to enable (1-7) */
    )
```

DESCRIPTION This routine enables a specified bus interrupt level.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

S

RETURNS **OK**, or **ERROR** if *intLevel* is out of range.

SEE ALSO **sysLib**, **sysIntDisable( )**, and BSP-specific reference pages for this routine.

sysLocalToBusAdrs( )

NAME **sysLocalToBusAdrs( )** – convert a local address to a bus address

SYNOPSIS

```
STATUS sysLocalToBusAdrs
    (
    int    adrsSpace,       /* bus address space in which busAdrs resides */
    char * localAdrs,       /* local address to convert */
    char * *pBusAdrs        /* where to return bus address */
    )
```

DESCRIPTION This routine gets the bus address that accesses a specified local memory address.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS **OK**, or **ERROR** if the address space is unknown or not mapped.

SEE ALSO **sysLib**, **sysBusToLocalAdrs( )**, and BSP-specific reference pages for this routine.

sysMailboxConnect( )

NAME **sysMailboxConnect( )** – connect a routine to the mailbox interrupt

SYNOPSIS

```
STATUS sysMailboxConnect
    (
    FUNCPTR routine,          /* routine called at each mailbox interrupt */
    int     arg               /* argument with which to call routine */
    )
```

DESCRIPTION This routine specifies the interrupt service routine to be called at each mailbox interrupt.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS **OK**, or **ERROR** if the routine cannot be connected to the interrupt.

SEE ALSO **sysLib**, **intConnect( )**, **sysMailboxEnable( )**, and BSP-specific reference pages for this routine.

sysMailboxEnable( )

NAME **sysMailboxEnable( )** – enable the mailbox interrupt

SYNOPSIS

```
STATUS sysMailboxEnable
    (
    char * mailboxAdrs        /* address of mailbox (ignored) */
    )
```

DESCRIPTION This routine enables the mailbox interrupt.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS **OK**, always.

SEE ALSO **sysLib**, **sysMailboxConnect( )**, and BSP-specific reference pages for this routine.

sysMemTop( )

NAME **sysMemTop( )** – get the address of the top of logical memory

SYNOPSIS

```
char *sysMemTop (void)
```

DESCRIPTION This routine returns the address of the top of memory.

S

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS The address of the top of memory.

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine.

sysModel( )

NAME **sysModel( )** – return the model name of the CPU board

SYNOPSIS

```
char *sysModel (void)
```

DESCRIPTION This routine returns the model name of the CPU board.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS A pointer to a string containing the board name.

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine.

sysNanoDelay( )

NAME **sysNanoDelay( )** – delay for specified number of nanoseconds

SYNOPSIS

```
void sysNanoDelay
    (
    UINT32 nanoseconds        /* nanoseconds to delay */
    )
```

DESCRIPTION This is an optional API for BSPs to provide. Some, but not all, drivers do require the BSP to implement this function.

When implemented, this function implements a spin loop type delay for at least the specified number of nanoseconds. This is not a task delay, control of the processor is not given up to another task. The actual delay must be equal to or greater than the requested number of nanoseconds.

The purpose of this function is to provide a reasonably accurate time delay of very short duration. It should not be used for any delays that are much greater than two system clock ticks in length. For delays of a full clock tick, or more, the use of **taskDelay( )** is recommended.

This routine should be implemented as interrupt safe.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS N/A.

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine.

sysNvRamGet( )

NAME **sysNvRamGet( )** – get the contents of non-volatile RAM

SYNOPSIS

```
STATUS sysNvRamGet
    (
    char * string,            /* where to copy non-volatile RAM */
    int    strLen,            /* maximum number of bytes to copy */
    int    offset             /* byte offset into non-volatile RAM */
    )
```

DESCRIPTION This routine copies the contents of non-volatile memory into a specified string. The string will be terminated with an EOS.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS **OK**, or **ERROR** if access is outside the non-volatile RAM address range.

SEE ALSO **sysLib**, **sysNvRamSet( )**, and BSP-specific reference pages for this routine.

S

sysNvRamSet( )

NAME **sysNvRamSet( )** – write to non-volatile RAM

SYNOPSIS

```
STATUS sysNvRamSet
    (
    char * string,          /* string to be copied into non-volatile RAM */
    int    strLen,          /* maximum number of bytes to copy */
    int    offset           /* byte offset into non-volatile RAM */
    )
```

DESCRIPTION This routine copies a specified string into non-volatile RAM.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS **OK**, or **ERROR** if access is outside the non-volatile RAM address range.

SEE ALSO **sysLib**, **sysNvRamGet( )**, and BSP-specific reference pages for this routine.

sysPhysMemTop( )

NAME **sysPhysMemTop( )** – get the address of the top of memory

SYNOPSIS

```
char * sysPhysMemTop (void)
```

DESCRIPTION This routine returns the address of the first missing byte of memory, which indicates the top of memory.

Normally, the amount of physical memory is specified with the macro **LOCAL_MEM_SIZE**. BSPs that support run-time memory sizing do so only if the macro **LOCAL_MEM_AUTOSIZE** is defined. If not defined, then **LOCAL_MEM_SIZE** is assumed to be, and must be, the true size of physical memory.

NOTE: Do no adjust **LOCAL_MEM_SIZE** to reserve memory for application use. See **sysMemTop( )** for more information on reserving memory.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS

The address of the top of physical memory.

SEE ALSO

sysLib, **sysMemTop( )**, and BSP-specific reference pages for this routine.

sysProcNumGet( )

NAME

sysProcNumGet( ) – get the processor number

SYNOPSIS

```
int sysProcNumGet (void)
```

DESCRIPTION

This routine returns the processor number for the CPU board, which is set with **sysProcNumSet( )**.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS

The processor number for the CPU board.

SEE ALSO

sysLib, **sysProcNumSet( )**, and BSP-specific reference pages for this routine.

S

sysProcNumSet( )

NAME

sysProcNumSet( ) – set the processor number

SYNOPSIS

```
void sysProcNumSet
    (
    int procNum               /* processor number */
    )
```

DESCRIPTION

This routine sets the processor number for the CPU board. Processor numbers should be unique on a single backplane.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS N/A

SEE ALSO **sysLib**, **sysProcNumGet( )**, and BSP-specific reference pages for this routine.

sysScsiBusReset( )

NAME **sysScsiBusReset( )** – assert the RST line on the SCSI bus (Western Digital WD33C93 only)

SYNOPSIS

```
void sysScsiBusReset
    (
    WD_33C93_SCSI_CTRL * pSbic /* ptr to SBIC info */
    )
```

DESCRIPTION This routine asserts the RST line on the SCSI bus, which causes all connected devices to return to a quiescent state.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS N/A

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine.

sysScsiConfig( )

NAME **sysScsiConfig( )** – system SCSI configuration

SYNOPSIS

```
STATUS sysScsiConfig (void)
```

DESCRIPTION This is an example SCSI configuration routine.

Most of the code found here is an example of how to declare a SCSI peripheral configuration. You must edit this routine to reflect the actual configuration of your SCSI bus. This example can also be found in **src/config/usrScsi.c**.

If you are just getting started, you can test your hardware configuration by defining **SCSI_AUTO_CONFIG**, which will probe the bus and display all devices found. No device should have the same SCSI bus ID as your VxWorks SCSI port (default = 7), or the same as any other device. Check for proper bus termination.

There are three configuration examples here. They demonstrate configuration of a SCSI hard disk (any type), an OMTI 3500 floppy disk, and a tape drive (any type).

Hard Disk The hard disk is divided into two 32-Mbyte partitions and a third partition with the remainder of the disk. The first partition is initialized as a dosFs device. The second and third partitions are initialized as rt11Fs devices, each with 256 directory entries.

It is recommended that the first partition (**BLK_DEV**) on a block device be a dosFs device, if the intention is eventually to boot VxWorks from the device. This will simplify the task considerably.

Floppy Disk The floppy, since it is a removable medium device, is allowed to have only a single partition, and dosFs is the file system of choice for this device, since it facilitates media compatibility with IBM PC machines.

In contrast to the hard disk configuration, the floppy setup in this example is more intricate. Note that the **scsiPhysDevCreate( )** call is issued twice. The first time is merely to get a "handle" to pass to **scsiModeSelect( )**, since the default media type is sometimes inappropriate (in the case of generic SCSI-to-floppy cards). After the hardware is correctly configured, the handle is discarded via **scsiPhysDevDelete( )**, after which the peripheral is correctly configured by a second call to **scsiPhysDevCreate( )**. (Before the **scsiModeSelect( )** call, the configuration information was incorrect.) Note that after the **scsiBlkDevCreate( )** call, the correct values for *sectorsPerTrack* and *nHeads* must be set via **scsiBlkDevInit( )**. This is necessary for IBM PC compatibility.

Tape Drive The tape configuration is also somewhat complex because certain device parameters need to turned off within VxWorks and the fixed-block size needs to be defined, assuming that the tape supports fixed blocks.

The last parameter to the **dosFsDevInit( )** call is a pointer to a **DOS_VOL_CONFIG** structure. By specifying **NULL**, you are asking **dosFsDevInit( )** to read this information off the disk in the drive. This may fail if no disk is present or if the disk has no valid dosFs directory. Should this be the case, you can use the **dosFsMkfs( )** command to create a new directory on a disk. This routine uses default parameters (see **dosFsLib**) that may not be suitable for your application, in which case you should use **dosFsDevInit( )** with a pointer to a valid **DOS_VOL_CONFIG** structure that you have created and initialized. If **dosFsDevInit( )** is used, a **diskInit( )** call should be made to write a new directory on the disk, if the disk is blank or disposable.

NOTE The variable **pSbdFloppy** is global to allow the above calls to be made from the VxWorks shell, for example:

```
-> dosFsMkfs "/fd0/", pSbdFloppy
```

If a disk is new, use **diskFormat( )** to format it.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS **OK** or **ERROR**.

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine.

sysScsiInit( )

NAME **sysScsiInit( )** – initialize an on-board SCSI port

SYNOPSIS

```
STATUS sysScsiInit (void)
```

DESCRIPTION This routine creates and initializes a SCSI control structure, enabling use of the on-board SCSI port. It also connects the proper interrupt service routine to the desired vector, and enables the interrupt at the desired level.

If SCSI DMA is supported by the board and **INCLUDE_SCSI_DMA** is defined, the DMA is also initialized.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS **OK**, or **ERROR** if the control structure cannot be connected, the controller cannot be initialized, or the DMA's interrupt cannot be connected.

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine.

sysSerialChanGet( )

NAME **sysSerialChanGet( )** – get the **SIO_CHAN** device associated with a serial channel

SYNOPSIS

```
SIO_CHAN * sysSerialChanGet
    (
    int channel               /* serial channel */
    )
```

DESCRIPTION This routine gets the **SIO_CHAN** device associated with a specified serial channel.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS A pointer to the **SIO_CHAN** structure for the channel, or **ERROR** if the channel is invalid.

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine.

S

sysSerialHwInit( )

NAME **sysSerialHwInit( )** – initialize the BSP serial devices to a quiescent state

SYNOPSIS

```
void sysSerialHwInit (void)
```

DESCRIPTION This routine initializes the BSP serial device descriptors and puts the devices in a quiescent state. It is called from **sysHwInit( )** with interrupts locked.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS N/A

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine.

sysSerialHwInit2( )

NAME **sysSerialHwInit2( )** – connect BSP serial device interrupts

SYNOPSIS

```
void sysSerialHwInit2 (void)
```

DESCRIPTION This routine connects the BSP serial device interrupts. It is called from **sysHwInit2( )**. Serial device interrupts could not be connected in **sysSerialHwInit( )** because the kernel memory allocator was not initialized at that point, and **intConnect( )** calls **malloc( )**.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS N/A

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine.

sysSerialReset( )

NAME **sysSerialReset( )** – reset all SIO devices to a quiet state

SYNOPSIS

```
void sysSerialReset (void)
```

DESCRIPTION This routine is called from **sysToMonitor( )** to reset all SIO device and prevent them from generating interrupts or performing DMA cycles.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS N/A

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine.

system( )

NAME **system( )** – pass a string to a command processor (Unimplemented) (ANSI)

SYNOPSIS

```
int system
    (
    const char * string       /* pointer to string */
    )
```

DESCRIPTION This function is not applicable to VxWorks.

INCLUDE FILES **stdlib.h**

RETURNS **OK**, always.

SEE ALSO **ansiStdlib**

sysToMonitor( )

NAME **sysToMonitor( )** – transfer control to the ROM monitor

SYNOPSIS

```
STATUS sysToMonitor
    (
    int startType             /* parameter passed to ROM to tell it how */
                              /* to boot */
    )
```

DESCRIPTION This routine transfers control to the ROM monitor. Normally, it is called only by **reboot( )**--which services **CTRL+X**--and by bus errors at interrupt level. However, in some circumstances, the user may wish to introduce a *startType* to enable special boot ROM facilities.

NOTE: This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS Does not return.

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine.

S

tan( )

NAME
tan( ) – compute a tangent (ANSI)

SYNOPSIS
```
double tan
    (
    double x                  /* angle in radians */
    )
```

DESCRIPTION
This routine computes the tangent of *x* in double precision. The angle *x* is expressed in radians.

INCLUDE FILES
math.h

RETURNS
The double-precision tangent of *x*.

SEE ALSO
ansiMath, **mathALib**

tanf( )

NAME
tanf( ) – compute a tangent (ANSI)

SYNOPSIS
```
float tanf
    (
    float x    /* angle in radians */
    )
```

DESCRIPTION
This routine returns the tangent of *x* in single precision. The angle *x* is expressed in radians.

INCLUDE FILES
math.h

RETURNS
The single-precision tangent of *x*.

SEE ALSO
mathALib

tanh( )

NAME **tanh( )** – compute a hyperbolic tangent (ANSI)

SYNOPSIS

```
double tanh
    (
    double x                /* number whose hyperbolic tangent is required */
    )
```

DESCRIPTION This routine returns the hyperbolic tangent of *x* in double precision (IEEE double, 53 bits).

INCLUDE FILES **math.h**

RETURNS The double-precision hyperbolic tangent of *x*.

Special cases:
If *x* is NaN, **tanh( )** returns NaN.

SEE ALSO **ansiMath**, **mathALib**

tanhf( )

NAME **tanhf( )** – compute a hyperbolic tangent (ANSI)

SYNOPSIS

```
float tanhf
    (
    float x    /* number whose hyperbolic tangent is required */
    )
```

DESCRIPTION This routine returns the hyperbolic tangent of *x* in single precision.

INCLUDE FILES **math.h**

RETURNS The single-precision hyperbolic tangent of *x*.

SEE ALSO **mathALib**

tapeFsDevInit( )

NAME

tapeFsDevInit( ) – associate a sequential device with tape volume functions

SYNOPSIS

```
TAPE_VOL_DESC *tapeFsDevInit
    (
    char *         volName,    /* volume name */
    SEQ_DEV *      pSeqDev,    /* pointer to sequential device info */
    TAPE_CONFIG * pTapeConfig /* pointer to tape config info */
    )
```

DESCRIPTION

This routine takes a sequential device created by a device driver and defines it as a tape file system volume. As a result, when high-level I/O operations, such as **open( )** and **write( )**, are performed on the device, the calls will be routed through **tapeFsLib**.

This routine associates **volName** with a device and installs it in the VxWorks I/O system-device table. The driver number used when the device is added to the table is that which was assigned to the tape library during **tapeFsInit( )**. (The driver number is kept in the global variable **tapeFsDrvNum**.)

The **SEQ_DEV** structure specified by **pSeqDev** contains configuration data describing the device and the addresses of the routines which are called to read blocks, write blocks, write file marks, reset the device, check device status, perform other I/O control functions (**ioctl( )**), reserve and release devices, load and unload devices, and rewind devices. These routines are not called until they are required by subsequent I/O operations. The **TAPE_CONFIG** structure is used to define configuration parameters for the **TAPE_VOL_DESC**. The configuration parameters are defined and described in **tapeFsLib.h**.

RETURNS

A pointer to the volume descriptor (**TAPE_VOL_DESC**), or **NULL** if there is an error.

ERRNO

S_tapeFsLib_NO_SEQ_DEV, S_tapeFsLib_ILLEGAL_TAPE_CONFIG_PARM

SEE ALSO

tapeFsLib

tapeFsInit( )

NAME **tapeFsInit( )** – initialize the tape volume library

SYNOPSIS

```
STATUS tapeFsInit ()
```

DESCRIPTION This routine initializes the tape volume library. It must be called exactly once, before any other routine in the library. Only one file descriptor per volume is assumed.

This routine also installs tape volume library routines in the VxWorks I/O system driver table. The driver number assigned to **tapeFsLib** is placed in the global variable **tapeFsDrvNum**. This number is later associated with system file descriptors opened to tapeFs devices.

To enable this initialization, simply call the routine **tapeFsDevInit( )**, which automatically calls **tapeFsInit( )** in order to initialize the tape file system.

RETURNS **OK** or **ERROR**.

SEE ALSO **tapeFsLib**

tapeFsReadyChange( )

NAME **tapeFsReadyChange( )** – notify **tapeFsLib** of a change in ready status

SYNOPSIS

```
STATUS tapeFsReadyChange
    (
    TAPE_VOL_DESC * pTapeVol  /* pointer to volume descriptor */
    )
```

DESCRIPTION This routine sets the volume descriptor state to **TAPE_VD_READY_CHANGED**. It should be called whenever a driver senses that a device has come on-line or gone off-line (for example, that a tape has been inserted or removed).

After this routine has been called, the next attempt to use the volume results in an attempted remount.

RETURNS **OK** if the read change status is set, or **ERROR** if the file descriptor is in use.

ERRNO **S_tapeFsLib_FILE_DESCRIPTOR_BUSY**

SEE ALSO **tapeFsLib**

tapeFsVolUnmount( )

NAME

tapeFsVolUnmount( ) – disable a tape device volume

SYNOPSIS

```
STATUS tapeFsVolUnmount
    (
    TAPE_VOL_DESC * pTapeVol  /* pointer to volume descriptor */
    )
```

DESCRIPTION

This routine is called when I/O operations on a volume are to be discontinued. This is commonly done before changing removable tape. All buffered data for the volume is written to the device (if possible), any open file descriptors are marked obsolete, and the volume is marked not mounted.

Because this routine flushes data from memory to the physical device, it should not be used in situations where the tape-change is not recognized until after a new tape has been inserted. In these circumstances, use the ready-change mechanism. (See the manual entry for **tapeFsReadyChange( )**.)

This routine may also be called by issuing an **ioctl( )** call using the **FIOUNMOUNT** function code.

RETURNS

OK, or **ERROR** if the routine cannot access the volume.

ERRNO

S_tapeFsLib_VOLUME_NOT_AVAILABLE, **S_tapeFsLib_FILE_DESCRIPTOR_BUSY**, **S_tapeFsLib_SERVICE_NOT_AVAILABLE**

SEE ALSO

tapeFsLib, **tapeFsReadyChange( )**

tarArchive( )

NAME

tarArchive( ) – archive named file/dir onto tape in tar format

SYNOPSIS

```
STATUS tarArchive
    (
    char * pTape,            /* tape device name */
    int    bfactor,          /* requested blocking factor */
    BOOL   verbose,          /* if TRUE print progress info */
    char * pName             /* file/dir name to archive */
    )
```

DESCRIPTION

This function creates a UNIX compatible tar formatted archives which contain entire file hierarchies from disk file systems. Files and directories are archived with mode and time information as returned by **stat( )**.

The *tape* argument can be any tape drive device name or a name of any file that will be created if necessary, and will contain the archive. If *tape* is set to "-", standard output will be used. If *tape* is **NULL** (unspecified from Shell), the default archive file name stored in global variable *TAPE* will be used.

Each **write( )** of the archive file will be exactly *bfactor**512 bytes long, hence on tapes in variable mode, this will be the physical block size on the tape. With Fixed Mode tapes this is only a performance matter. If *bfactor* is 0, or unspecified from Shell, it will be set to the default value of 20.

The *verbose* argument is a boolean, if set to 1, will cause informative messages to be printed to standard error whenever an action is taken, otherwise, only errors are reported.

The *name* argument is the path of the hierarchy to be archived. if **NULL** (or unspecified from the Shell), the current directory path "." will be used. This is the path as seen from the target, not from the Tornado host.

All informative and error message are printed to standard error.

NOTE: Refrain from specifying absolute path names in *path*, such archives tend to be either difficult to extract or can cause unexpected damage to existing files if such exist under the same absolute name.

There is no way of specifying a number of hierarchies to dump.

SEE ALSO

tarLib

tarExtract( )

NAME

tarExtract( ) – extract all files from a tar formatted tape

SYNOPSIS

```
STATUS tarExtract
    (
    char * pTape,              /* tape device name */
    int    bfactor,            /* requested blocking factor */
    BOOL   verbose             /* if TRUE print progress info */
    )
```

DESCRIPTION

This is a UNIX-tar compatible utility that extracts entire file hierarchies from tar-formatted archive. The files are extracted with their original names and modes. In some cases a file cannot be created on disk, for example if the name is too long for regular DOS file name conventions, in such cases entire files are skipped, and this program will continue with the next file. Directories are created in order to be able to create all files on tape.

The *tape* argument may be any tape device name or file name that contains a tar formatted archive. If *tape* is equal "-", standard input is used. If *tape* is **NULL** (or unspecified from Shell) the default archive file name stored in global variable *TAPE* is used.

The *bfactor* dictates the blocking factor the tape was written with. If 0, or unspecified from the shell, a default of 20 is used.

The *verbose* argument is a boolean, if set to 1, will cause informative messages to be printed to standard error whenever an action is taken, otherwise, only errors are reported.

All informative and error message are printed to standard error.

There is no way to selectively extract tar archives with this utility. It extracts entire archives.

SEE ALSO **tarLib**

tarToc( )

NAME **tarToc( )** – display all contents of a tar formatted tape

SYNOPSIS

```
STATUS tarToc
    (
    char * tape,              /* tape device name */
    int    bfactor            /* requested blocking factor */
    )
```

DESCRIPTION This is a UNIX-tar compatible utility that displays entire file hierarchies from tar-formatted media, *e.g.* tape.

The *tape* argument may be any tape device name or file name that contains a tar formatted archive. If *tape* is equal "-", standard input is used. If *tape* is **NULL** (or unspecified from Shell) the default archive file name stored in global variable *TAPE* is used.

The *bfactor* dictates the blocking factor the tape was written with. If 0, or unspecified from Shell, default of 20 is used.

Archive contents are displayed on standard output, while all informative and error message are printed to standard error.

SEE ALSO **tarLib**

taskActivate( )

NAME **taskActivate( )** – activate a task that has been initialized

SYNOPSIS

```
STATUS taskActivate
    (
    int tid                    /* task ID of task to activate */
    )
```

DESCRIPTION This routine activates tasks created by **taskInit( )**. Without activation, a task is ineligible for CPU allocation by the scheduler.

The *tid* (task ID) argument is simply the address of the **WIND_TCB** for the task (the **taskInit( )** *pTcb* argument), cast to an integer:

```
tid = (int) pTcb;
```

The **taskSpawn( )** routine is built from **taskActivate( )** and **taskInit( )**. Tasks created by **taskSpawn( )** do not require explicit task activation.

RETURNS **OK**, or **ERROR** if the task cannot be activated.

SEE ALSO **taskLib**, **taskInit( )**

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 routine should be declared as follows:

```
void createHook
    (
    WIND_TCB *pNewTcb     /* pointer to new task's TCB */
    )
```

T

RETURNS **OK**, or **ERROR** if the table of task create routines 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.

SEE ALSO **taskHookLib**, **taskCreateHookAdd( )**

taskCreateHookShow( )

NAME **taskCreateHookShow( )** – show the list of task create routines

SYNOPSIS

```
void taskCreateHookShow (void)
```

DESCRIPTION This routine shows all the task create routines installed in the task create hook table, in the order in which they were installed.

RETURNS N/A

SEE ALSO **taskHookShow**, **taskCreateHookAdd( )**

taskDelay( )

NAME **taskDelay( )** – delay a task from executing

SYNOPSIS

```
STATUS taskDelay
    (
    int ticks                 /* number of ticks to delay task */
    )
```

DESCRIPTION This routine causes the calling task to relinquish the CPU for the duration specified (in ticks). This is commonly referred to as manual rescheduling, but it is also useful when waiting for some external condition that does not have an interrupt associated with it.

If the calling task receives a signal that is not being blocked or ignored, **taskDelay( )** returns **ERROR** and sets **errno** to **EINTR** after the signal handler is run.

RETURNS **OK**, or **ERROR** if called from interrupt level or if the calling task receives a signal that is not blocked or ignored.

ERRNO **S_intLib_NOT_ISR_CALLABLE, EINTR**

SEE ALSO **taskLib**

taskDelete( )

NAME **taskDelete( )** – delete a task

SYNOPSIS

```
STATUS taskDelete
    (
    int tid                   /* task ID of task to delete */
    )
```

DESCRIPTION This routine causes a specified task to cease to exist and deallocates the stack and **WIND_TCB** memory resources. Upon deletion, all routines specified by **taskDeleteHookAdd( )** will be called in the context of the deleting task. This routine is the companion routine to **taskSpawn( )**.

RETURNS **OK**, or **ERROR** if the task cannot be deleted.

ERRNO **S_intLib_NOT_ISR_CALLABLE, S_objLib_OBJ_DELETED, S_objLib_OBJ_UNAVAILABLE, S_objLib_OBJ_ID_ERROR**

SEE ALSO **taskLib**, **excLib**, **taskDeleteHookAdd( )**, **taskSpawn( )**, *VxWorks Programmer's Guide: Basic OS*

taskDeleteForce( )

NAME **taskDeleteForce( )** – delete a task without restriction

SYNOPSIS

```
STATUS taskDeleteForce
    (
    int tid                  /* task ID of task to delete */
    )
```

DESCRIPTION This routine deletes a task even if the task is protected from deletion. It is similar to **taskDelete( )**. Upon deletion, all routines specified by **taskDeleteHookAdd( )** will be called in the context of the deleting task.

WARNING: This routine is intended as a debugging aid, and is generally inappropriate for applications. Disregarding a task's deletion protection could leave the system in an unstable state or lead to system deadlock.

The system does not protect against simultaneous **taskDeleteForce( )** calls. Such a situation could leave the system in an unstable state.

RETURNS **OK**, or **ERROR** if the task cannot be deleted.

ERRNO **S_intLib_NOT_ISR_CALLABLE, S_objLib_OBJ_DELETED, S_objLib_OBJ_UNAVAILABLE, S_objLib_OBJ_ID_ERROR**

SEE ALSO **taskLib**, **taskDeleteHookAdd( )**, **taskDelete( )**

taskDeleteHookAdd( )

NAME

taskDeleteHookAdd( ) – add a routine to be called at every task delete

SYNOPSIS

```
STATUS taskDeleteHookAdd
    (
    FUNCPTR deleteHook      /* routine to be called when a task is deleted */
    )
```

DESCRIPTION

This routine adds a specified routine to a list of routines that will be called whenever a task is deleted. The routine should be declared as follows:

```
    void deleteHook
        (
        WIND_TCB *pTcb        /* pointer to deleted task's WIND_TCB */
        )
```

RETURNS

OK, or **ERROR** if the table of task delete routines is full.

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.

SEE ALSO

taskHookLib, **taskDeleteHookAdd( )**

T

taskDeleteHookShow( )

NAME **taskDeleteHookShow( )** – show the list of task delete routines

SYNOPSIS

```
void taskDeleteHookShow (void)
```

DESCRIPTION This routine shows all the delete routines installed in the task delete hook table, in the order in which they were installed. Note that the delete routines will be run in reverse of the order in which they were installed.

RETURNS N/A

SEE ALSO **taskHookShow**, **taskDeleteHookAdd( )**

taskHookInit( )

NAME **taskHookInit( )** – initialize task hook facilities

SYNOPSIS

```
void taskHookInit (void)
```

DESCRIPTION This routine is a **NULL** routine called to configure the task hook package into the system. It is called automatically if the configuration macro **INCLUDE_TASK_HOOKS** is defined.

RETURNS N/A

SEE ALSO **taskHookLib**

taskHookShowInit( )

NAME

taskHookShowInit( ) – initialize the task hook show facility

SYNOPSIS

```
void taskHookShowInit (void)
```

DESCRIPTION

This routine links the task hook show facility into the VxWorks system. It is called automatically when the task hook show facility is configured into VxWorks using either of the following methods:

– If you use the configuration header files, define **INCLUDE_SHOW_ROUTINES** in **config.h**.

– If you use the Tornado project facility, select **INCLUDE_TASK_HOOK_SHOW**.

RETURNS

N/A

SEE ALSO

taskHookShow

taskIdDefault( )

NAME

taskIdDefault( ) – set the default task ID

SYNOPSIS

```
int taskIdDefault
    (
    int tid                  /* user supplied task ID; if 0, return default */
    )
```

DESCRIPTION

This routine maintains a global default task ID. This ID is used by libraries that want to allow a task ID argument to take on a default value if the user did not explicitly supply one.

If *tid* is not zero (*i.e.*, the user did specify a task ID), the default ID is set to that value, and that value is returned. If *tid* is zero (*i.e.*, the user did not specify a task ID), the default ID is not changed and its value is returned. Thus the value returned is always the last task ID the user specified.

RETURNS

The most recent non-zero task ID.

SEE ALSO

taskInfo, **dbgLib**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

T

taskIdListGet( )

NAME **taskIdListGet( )** – get a list of active task IDs

SYNOPSIS

```
int taskIdListGet
    (
    int idList[],             /* array of task IDs to be filled in */
    int maxTasks              /* max tasks idList can accommodate */
    )
```

DESCRIPTION This routine provides the calling task with a list of all active tasks. An unsorted list of task IDs for no more than *maxTasks* tasks is put into *idList*.

WARNING: Kernel rescheduling is disabled with **taskLock( )** while tasks are filled into the *idList*. There is no guarantee that all the tasks are valid or that new tasks have not been created by the time this routine returns.

RETURNS The number of tasks put into the ID list.

SEE ALSO **taskInfo**

taskIdSelf( )

NAME **taskIdSelf( )** – get the task ID of a running task

SYNOPSIS

```
int taskIdSelf (void)
```

DESCRIPTION This routine gets the task ID of the calling task. The task ID will be invalid if called at interrupt level.

RETURNS The task ID of the calling task.

SEE ALSO **taskLib**

taskIdVerify( )

NAME

taskIdVerify( ) – verify the existence of a task

SYNOPSIS

```
STATUS taskIdVerify
    (
    int tid                    /* task ID */
    )
```

DESCRIPTION

This routine verifies the existence of a specified task by validating the specified ID as a task ID. Note that an exception occurs if the task ID parameter points to an address not located in physical memory.

RETURNS

OK, or **ERROR** if the task ID is invalid.

ERRNO

S_objLib_OBJ_ID_ERROR

SEE ALSO

taskLib

taskInfoGet( )

NAME

taskInfoGet( ) – get information about a task

SYNOPSIS

```
STATUS taskInfoGet
    (
    int         tid,           /* ID of task for which to get info */
    TASK_DESC * pTaskDesc      /* task descriptor to be filled in */
    )
```

DESCRIPTION

This routine fills in a specified task descriptor (**TASK_DESC**) for a specified task. The information in the task descriptor is, for the most part, a copy of information kept in the task control block (**WIND_TCB**). The **TASK_DESC** structure is useful for common information and avoids dealing directly with the unwieldy **WIND_TCB**.

NOTE: Examination of **WIND_TCBs** should be restricted to debugging aids.

RETURNS

OK, or **ERROR** if the task ID is invalid.

SEE ALSO

taskShow

taskInit( )

NAME **taskInit( )** – initialize a task with a stack at a specified address

SYNOPSIS

```
STATUS taskInit
    (
    WIND_TCB * pTcb,          /* address of new task's TCB */
    char *     name,          /* name of new task (stored at pStackBase) */
    int        priority,      /* priority of new task */
    int        options,       /* task option word */
    char *     pStackBase,    /* base of new task's stack */
    int        stackSize,     /* size (bytes) of stack needed */
    FUNCPTR    entryPt,       /* entry point of new task */
    int        arg1,          /* first of ten task args to pass to func */
    int        arg2,
    int        arg3,
    int        arg4,
    int        arg5,
    int        arg6,
    int        arg7,
    int        arg8,
    int        arg9,
    int        arg10
    )
```

DESCRIPTION This routine initializes user-specified regions of memory for a task stack and control block instead of allocating them from memory as **taskSpawn( )** does. This routine will utilize the specified pointers to the **WIND_TCB** and stack as the components of the task. This allows, for example, the initialization of a static **WIND_TCB** variable. It also allows for special stack positioning as a debugging aid.

As in **taskSpawn( )**, a task may be given a name. While **taskSpawn( )** automatically names unnamed tasks, **taskInit( )** permits the existence of tasks without names. The task ID required by other task routines is simply the address *pTcb*, cast to an integer.

Note that the task stack may grow up or down from *pStackBase*, depending on the target architecture.

Other arguments are the same as in **taskSpawn( )**. Unlike **taskSpawn( )**, **taskInit( )** does not activate the task. This must be done by calling **taskActivate( )** after calling **taskInit( )**.

Normally, tasks should be started using **taskSpawn( )** rather than **taskInit( )**, except when additional control is required for task memory allocation or a separate task activation is desired.

RETURNS **OK**, or **ERROR** if the task cannot be initialized.

ERRNO **S_intLib_NOT_ISR_CALLABLE**, **S_objLib_OBJ_ID_ERROR**, **S_taskLib_ILLEGAL_PRIORITY**

SEE ALSO **taskLib**, **taskActivate( )**, **taskSpawn( )**

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**.

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**.

SEE ALSO **taskInfo**

T

taskLock( )

NAME **taskLock( )** – disable task rescheduling

SYNOPSIS

```
STATUS taskLock (void)
```

DESCRIPTION This routine disables task context switching. The task that calls this routine will be the only task that is allowed to execute, unless the task explicitly gives up the CPU by making itself no longer ready. Typically this call is paired with **taskUnlock( )**; together they surround a critical section of code. These preemption locks are implemented with a counting variable that allows nested preemption locks. Preemption will not be unlocked until **taskUnlock( )** has been called as many times as **taskLock( )**.

This routine does not lock out interrupts; use **intLock( )** to lock out interrupts.

A **taskLock( )** is preferable to **intLock( )** as a means of mutual exclusion, because interrupt lock-outs add interrupt latency to the system.

A **semTake( )** is preferable to **taskLock( )** as a means of mutual exclusion, because preemption lock-outs add preemptive latency to the system.

The **taskLock( )** routine is not callable from interrupt service routines.

RETURNS **OK** or **ERROR**.

ERRNO **S_objLib_OBJ_ID_ERROR**, **S_intLib_NOT_ISR_CALLABLE**

SEE ALSO **taskLib**, **taskUnlock( )**, **intLock( )**, **taskSafe( )**, **semTake( )**

taskName( )

NAME **taskName( )** – get the name associated with a task ID

SYNOPSIS

```
char *taskName
    (
    int tid                  /* ID of task whose name is to be found */
    )
```

DESCRIPTION This routine returns a pointer to the name of a task of a specified ID, if the task has a name. If the task has no name, it returns an empty string.

RETURNS A pointer to the task name, or **NULL** if the task ID is invalid.

SEE ALSO **taskInfo**

taskNameToId( )

NAME **taskNameToId( )** – look up the task ID associated with a task name

SYNOPSIS

```
int taskNameToId
    (
    char * name              /* task name to look up */
    )
```

DESCRIPTION This routine returns the ID of the task matching a specified name. Referencing a task in this way is inefficient, since it involves a search of the task list.

RETURNS The task ID, or **ERROR** if the task is not found.

ERRNO **S_taskLib_NAME_NOT_FOUND**

SEE ALSO **taskInfo**

taskOptionsGet( )

NAME **taskOptionsGet( )** – examine task options

SYNOPSIS

```
STATUS taskOptionsGet
    (
    int   tid,               /* task ID */
    int * pOptions           /* task's options */
    )
```

DESCRIPTION This routine gets the current execution options of the specified task. The option bits returned by this routine indicate the following modes:

VX_FP_TASK
execute with floating-point coprocessor support.

T

VX_PRIVATE_ENV
include private environment support (see **envLib**).

VX_NO_STACK_FILL
do not fill the stack for use by **checkstack( )**.

VX_UNBREAKABLE
do not allow breakpoint debugging.

For definitions, see **taskLib.h**.

RETURNS **OK**, or **ERROR** if the task ID is invalid.

SEE ALSO **taskInfo**, **taskOptionsSet( )**

taskOptionsSet( )

NAME **taskOptionsSet( )** – change task options

SYNOPSIS

```
STATUS taskOptionsSet
    (
    int tid,                  /* task ID */
    int mask,                 /* bit mask of option bits to unset */
    int newOptions            /* bit mask of option bits to set */
    )
```

DESCRIPTION This routine changes the execution options of a task. The only option that can be changed after a task has been created is:

VX_UNBREAKABLE
do not allow breakpoint debugging.

For definitions, see **taskLib.h**.

RETURNS **OK**, or **ERROR** if the task ID is invalid.

SEE ALSO **taskInfo**, **taskOptionsGet( )**

taskPriorityGet( )

NAME

taskPriorityGet( ) – examine the priority of a task

SYNOPSIS

```
STATUS taskPriorityGet
    (
    int   tid,                /* task ID */
    int * pPriority           /* return priority here */
    )
```

DESCRIPTION

This routine determines the current priority of a specified task. The current priority is copied to the integer pointed to by *pPriority*.

RETURNS

OK, or **ERROR** if the task ID is invalid.

ERRNO

S_objLib_OBJ_ID_ERROR

SEE ALSO

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.

RETURNS

OK, or **ERROR** if the task ID is invalid.

ERRNO

S_taskLib_ILLEGAL_PRIORITY, **S_objLib_OBJ_ID_ERROR**

SEE ALSO

taskLib, **taskPriorityGet( )**

taskRegsGet( )

NAME **taskRegsGet( )** – get a task's registers from the TCB

SYNOPSIS

```
STATUS taskRegsGet
    (
    int       tid,            /* task ID */
    REG_SET * pRegs           /* put register contents here */
    )
```

DESCRIPTION This routine gathers task information kept in the TCB. It copies the contents of the task's registers to the register structure *pRegs*.

NOTE: This routine only works well if the task is known to be in a stable, non-executing state. Self-examination, for instance, is not advisable, as results are unpredictable.

RETURNS **OK**, or **ERROR** if the task ID is invalid.

SEE ALSO **taskInfo**, **taskSuspend( )**, **taskRegsSet( )**

taskRegsSet( )

NAME **taskRegsSet( )** – set a task's registers

SYNOPSIS

```
STATUS taskRegsSet
    (
    int       tid,            /* task ID */
    REG_SET * pRegs           /* get register contents from here */
    )
```

DESCRIPTION This routine loads a specified register set *pRegs* into a specified task's TCB.

NOTE: This routine only works well if the task is known not to be in the ready state. Suspending the task before changing the register set is recommended.

RETURNS **OK**, or **ERROR** if the task ID is invalid.

SEE ALSO **taskInfo**, **taskSuspend( )**, **taskRegsGet( )**

taskRegsShow( )

NAME

taskRegsShow( ) – display the contents of a task's registers

SYNOPSIS

```
void taskRegsShow
    (
    int tid                    /* task ID */
    )
```

DESCRIPTION

This routine displays the register contents of a specified task on standard output.

EXAMPLE

The following example displays the register of the shell task (68000 family):

```
-> taskRegsShow (taskNameToId ("tShell"))
d0     =        0   d1     =        0   d2     =    578fe   d3     =        1
d4     =   3e84e1   d5     =   3e8568   d6     =        0   d7     = ffffffff
a0     =        0   a1     =        0   a2     =    4f06c   a3     =    578d0
a4     =   3fffc4   a5     =        0   fp     =   3e844c   sp     =   3e842c
sr     =     3000   pc     =    4f0f2
value = 0 = 0x0
```

RETURNS

N/A

SEE ALSO

taskShow

taskRestart( )

NAME

taskRestart( ) – restart a task

SYNOPSIS

```
STATUS taskRestart
    (
    int tid                    /* task ID of task to restart */
    )
```

DESCRIPTION

This routine "restarts" a task. The task is first terminated, and then re-initialized 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 the exception task. The shell utilizes this routine to restart itself when aborted.

NOTE: If the task has modified any of its start-up parameters, the restarted task will start with the changed values.

RETURNS **OK**, or **ERROR** if the task ID is invalid or the task could not be restarted.

ERRNO **S_intLib_NOT_ISR_CALLABLE, S_objLib_OBJ_DELETED, S_objLib_OBJ_UNAVAILABLE, S_objLib_OBJ_ID_ERROR, S_smObjLib_NOT_INITIALIZED, S_memLib_NOT_ENOUGH_MEMORY, S_memLib_BLOCK_ERROR, S_taskLib_ILLEGAL_PRIORITY**

SEE ALSO **taskLib**

taskResume( )

NAME **taskResume( )** – resume a task

SYNOPSIS

```
STATUS taskResume
    (
    int tid                    /* task ID of task to resume */
    )
```

DESCRIPTION This routine resumes a specified task. Suspension is cleared, and the task operates in the remaining state.

RETURNS **OK**, or **ERROR** if the task cannot be resumed.

ERRNO **S_objLib_OBJ_ID_ERROR**

SEE ALSO **taskLib**

taskSafe( )

NAME **taskSafe( )** – make the calling task safe from deletion

SYNOPSIS

```
STATUS taskSafe (void)
```

DESCRIPTION

This routine protects the calling task from deletion. Tasks that attempt to delete a protected task will block until the task is made unsafe, using **taskUnsafe( )**. When a task becomes unsafe, the deleter will be unblocked and allowed to delete the task.

The **taskSafe( )** primitive utilizes a count to keep track of nested calls for task protection. When nesting occurs, the task becomes unsafe only after the outermost **taskUnsafe( )** is executed.

RETURNS

OK.

SEE ALSO

taskLib, **taskUnsafe( )**, *VxWorks Programmer's Guide: Basic OS*

taskShow( )

NAME

taskShow( ) – display task information from TCBs

SYNOPSIS

```
STATUS taskShow
    (
    int tid,                    /* task ID */
    int level                   /* 0 = summary, 1 = details, 2 = all tasks */
    )
```

DESCRIPTION

This routine displays the contents of a task control block (TCB) for a specified task. If *level* is 1, it also displays task options and registers. If *level* is 2, it displays all tasks.

The TCB display contains the following fields:

Field	Meaning
NAME	Task name
ENTRY	Symbol name or address where task began execution
TID	Task ID
PRI	Priority
STATUS	Task status, as formatted by **taskStatusString( )**
PC	Program counter
SP	Stack pointer
ERRNO	Most recent error code for this task
DELAY	If task is delayed, number of clock ticks remaining in delay (0 otherwise)

EXAMPLE

The following example shows the TCB contents for the shell task:

```
-> taskShow tShell, 1
```

```
  NAME         ENTRY     TID     PRI  STATUS       PC        SP     ERRNO  DELAY
---------- --------- -------- --- --------- -------- -------- ------ -----
tShell     _shell     20efcac   1 READY      201dc90  20ef980      0     0
stack: base 0x20efcac  end 0x20ed59c  size 9532   high 1452   margin 8080
options: 0x1e
VX_UNBREAKABLE      VX_DEALLOC_STACK    VX_FP_TASK         VX_STDIO
VxWorks Events
--------------
Events Pended on    : Not Pended
Received Events     : 0x0
Options             : N/A

D0 =       0   D4 =       0   A0 =       0   A4 =        0
D1 =       0   D5 =       0   A1 =       0   A5 =  203a084   SR =     3000
D2 =       0   D6 =       0   A2 =       0   A6 =  20ef9a0   PC =  2038614
D3 =       0   D7 =       0   A3 =       0   A7 =  20ef980
value = 34536868 = 0x20efda4
```

RETURNS N/A

SEE ALSO **taskShow**, **taskStatusString( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

taskShowInit( )

NAME **taskShowInit( )** – initialize the task show routine facility

SYNOPSIS

```
void taskShowInit (void)
```

DESCRIPTION This routine links the task show routines into the VxWorks system. It is called automatically when the task show facility is configured into VxWorks using either of the following methods:

– If you use the configuration header files, define **INCLUDE_SHOW_ROUTINES** in **config.h**.

– If you use the Tornado project facility, select **INCLUDE_TASK_SHOW**.

RETURNS N/A

SEE ALSO **taskShow**

taskSpawn( )

NAME

taskSpawn( ) – spawn a task

SYNOPSIS

```
int taskSpawn
    (
    char *   name,        /* name of new task (stored at pStackBase) */
    int      priority,    /* priority of new task */
    int      options,     /* task option word */
    int      stackSize,   /* size (bytes) of stack needed plus name */
    FUNCPTR  entryPt,     /* entry point of new task */
    int      arg1,        /* 1st of 10 req'd task args to pass to func */
    int      arg2,
    int      arg3,
    int      arg4,
    int      arg5,
    int      arg6,
    int      arg7,
    int      arg8,
    int      arg9,
    int      arg10
    )
```

DESCRIPTION

This routine creates and activates a new task with a specified priority and options and returns a system-assigned ID. See **taskInit( )** and **taskActivate( )** for the building blocks of this routine.

A task may be assigned a name as a debugging aid. This name will appear in displays generated by various system information facilities such as **i( )**. The name may be of arbitrary length and content, but the current VxWorks convention is to limit task names to ten characters and prefix them with a "t". If *name* is specified as **NULL**, an ASCII name will be assigned to the task of the form "t*n*" where *n* is an integer which increments as new tasks are spawned.

The only resource allocated to a spawned task is a stack of a specified size *stackSize*, which is allocated from the system memory partition. Stack size should be an even integer. A task control block (TCB) is carved from the stack, as well as any memory required by the task name. The remaining memory is the task's stack and every byte is filled with the value 0xEE for the **checkStack( )** facility. See the manual entry for **checkStack( )** for stack-size checking aids.

The entry address *entryPt* is the address of the "main" routine of the task. The routine will be called once the C environment has been set up. The specified routine will be called with the ten given arguments. Should the specified main routine return, a call to **exit( )** will automatically be made.

Note that ten (and only ten) arguments must be passed for the spawned function.

Bits in the options argument may be set to run with the following modes:

VX_FP_TASK (0x0008)
: execute with floating-point coprocessor support. A task which performs floating point operations or calls any functions which either return or take a floating point value as arguments must be created with this option. Some routines perform floating point operations internally. The VxWorks documentation for these clearly state the need to use the **VX_FP_TASK** option.

VX_PRIVATE_ENV (0x0080)
: include private environment support (see **envLib**).

VX_NO_STACK_FILL (0x0100)
: do not fill the stack for use by **checkStack( )**.

VX_UNBREAKABLE (0x0002)
: do not allow breakpoint debugging.

See the definitions in **taskLib.h**.

RETURNS The task ID, or **ERROR** if memory is insufficient or the task cannot be created.

ERRNO **S_intLib_NOT_ISR_CALLABLE**, **S_objLib_OBJ_ID_ERROR**, **S_smObjLib_NOT_INITIALIZED**, **S_memLib_NOT_ENOUGH_MEMORY**, **S_memLib_BLOCK_ERROR**, **S_taskLib_ILLEGAL_PRIORITY**

SEE ALSO **taskLib**, **taskInit( )**, **taskActivate( )**, **sp( )**, *VxWorks Programmer's Guide: Basic OS*

---

taskSRInit( )

NAME **taskSRInit( )** – initialize the default task status register (MIPS)

SYNOPSIS

```
ULONG taskSRInit
    (
    ULONG newSRValue          /* new default task status register */
    )
```

DESCRIPTION This routine sets the default status register for system-wide tasks. All tasks are spawned with the status register set to this value; thus, it must be called before **kernelInit( )**.

RETURNS The previous value of the default status register.

SEE ALSO **taskArchLib**

taskSRSet( )

NAME

taskSRSet( ) – set the task status register (68K, MIPS, x86)

SYNOPSIS

```
STATUS taskSRSet
    (
    int    tid,              /* task ID */
    UINT16 sr                /* new SR */
    )
```

DESCRIPTION

This routine sets the status register of a task that is not running (*i.e.*, the TCB must not be that of the calling task). Debugging facilities use this routine to set the trace bit in the status register of a task that is being single-stepped.

x86:

The second parameter represents EFLAGS register and the size is 32 bit.

RETURNS

OK, or **ERROR** if the task ID is invalid.

SEE ALSO

taskArchLib

taskStatusString( )

NAME

taskStatusString( ) – get a task's status as a string

SYNOPSIS

```
STATUS taskStatusString
    (
    int    tid,              /* task to get string for */
    char * pString           /* where to return string */
    )
```

DESCRIPTION

This routine deciphers the WIND task status word in the TCB for a specified task, and copies the appropriate string to *pString*.

The formatted string is one of the following:

String	Meaning
READY	Task is not waiting for any resource other than the CPU.
PEND	Task is blocked due to the unavailability of some resource.
DELAY	Task is asleep for some duration.
SUSPEND	Task is unavailable for execution (but not suspended, delayed, or pended).

T

String	Meaning
DELAY+S	Task is both delayed and suspended.
PEND+S	Task is both pended and suspended.
PEND+T	Task is pended with a timeout.
PEND+S+T	Task is pended with a timeout, and also suspended.
...+I	Task has inherited priority (+I may be appended to any string above).
DEAD	Task no longer exists.

EXAMPLE

```
-> taskStatusString (taskNameToId ("tShell"), xx=malloc (10))
new symbol "xx" added to symbol table.
value = 0 = 0x0
-> printf ("shell status = <%s>\n", xx)
shell status = <READY>
value = 2 = 0x2
```

RETURNS

OK, or **ERROR** if the task ID is invalid.

SEE ALSO

taskShow

taskSuspend( )

NAME

taskSuspend( ) – suspend a task

SYNOPSIS

```
STATUS taskSuspend
    (
    int tid                   /* task ID of task to suspend */
    )
```

DESCRIPTION

This routine suspends a specified task. A task ID of zero results in the suspension of the calling task. Suspension is additive, thus tasks can be delayed and suspended, or pended and suspended. Suspended, delayed tasks whose delays expire remain suspended. Likewise, suspended, pended tasks that unblock remain suspended only.

Care should be taken with asynchronous use of this facility. The specified task is suspended regardless of its current state. The task could, for instance, have mutual exclusion to some system resource, such as the network * or system memory partition. If suspended during such a time, the facilities engaged are unavailable, and the situation often ends in deadlock.

This routine is the basis of the debugging and exception handling packages. However, as a synchronization mechanism, this facility should be rejected in favor of the more general semaphore facility.

RETURNS **OK**, or **ERROR** if the task cannot be suspended.

ERRNO **S_objLib_OBJ_ID_ERROR**

SEE ALSO **taskLib**

taskSwitchHookAdd( )

NAME **taskSwitchHookAdd( )** – add a routine to be called at every task switch

SYNOPSIS

```
STATUS taskSwitchHookAdd
    (
    FUNCPTR switchHook        /* routine to be called at every task switch */
    )
```

DESCRIPTION This routine adds a specified routine to a list of routines that will be called at every task switch. The routine should be declared as follows:

```
    void switchHook
        (
        WIND_TCB *pOldTcb,    /* pointer to old task's WIND_TCB */
        WIND_TCB *pNewTcb     /* pointer to new task's WIND_TCB */
        )
```

NOTE User-installed switch hooks are called within the kernel context. Therefore, switch hooks do not have access to all VxWorks facilities. The following routines can be called from within a task switch hook:

Library	Routines
bLib	All routines
fppArchLib	**fppSave( )**, **fppRestore( )**
intLib	**intContext( )**, **intCount( )**, **intVecSet( )**, **intVecGet( )**
lstLib	All routines
mathALib	All routines, if **fppSave( )**/**fppRestore( )** are used
rngLib	All routines except **rngCreate( )**
taskLib	**taskIdVerify( )**, **taskIdDefault( )**, **taskIsReady( )**, **taskIsSuspended( )**, **taskTcb( )**
vxLib	**vxTas( )**

RETURNS **OK**, or **ERROR** if the table of task switch routines is full.

SEE ALSO **taskHookLib**, **taskSwitchHookDelete( )**

taskSwitchHookDelete( )

NAME **taskSwitchHookDelete( )** – delete a previously added task switch routine

SYNOPSIS

```
STATUS taskSwitchHookDelete
    (
    FUNCPTR switchHook        /* routine to be deleted from list */
    )
```

DESCRIPTION This routine removes the specified routine from the list of routines to be called at each task switch.

RETURNS **OK**, or **ERROR** if the routine is not in the table of task switch routines.

SEE ALSO **taskHookLib**, **taskSwitchHookAdd( )**

taskSwitchHookShow( )

NAME **taskSwitchHookShow( )** – show the list of task switch routines

SYNOPSIS

```
void taskSwitchHookShow (void)
```

DESCRIPTION This routine shows all the switch routines installed in the task switch hook table, in the order in which they were installed.

RETURNS N/A

SEE ALSO **taskHookShow**, **taskSwitchHookAdd( )**

taskTcb( )

NAME **taskTcb( )** – get the task control block for a task ID

SYNOPSIS

```
WIND_TCB *taskTcb
    (
    int tid                    /* task ID */
    )
```

DESCRIPTION This routine returns a pointer to the task control block (**WIND_TCB**) for a specified task. Although all task state information is contained in the TCB, users must not modify it directly. To change registers, for instance, use **taskRegsSet( )** and **taskRegsGet( )**.

RETURNS A pointer to a **WIND_TCB**, or **NULL** if the task ID is invalid.

ERRNO **S_objLib_OBJ_ID_ERROR**

SEE ALSO **taskLib**

taskUnlock( )

NAME **taskUnlock( )** – enable task rescheduling

SYNOPSIS

```
STATUS taskUnlock (void)
```

DESCRIPTION This routine decrements the preemption lock count. Typically this call is paired with **taskLock( )** and concludes a critical section of code. Preemption will not be unlocked until **taskUnlock( )** has been called as many times as **taskLock( )**. When the lock count is decremented to zero, any tasks that were eligible to preempt the current task will execute.

The **taskUnlock( )** routine is not callable from interrupt service routines.

RETURNS **OK** or **ERROR**.

ERRNO **S_intLib_NOT_ISR_CALLABLE**

SEE ALSO **taskLib**, **taskLock( )**

taskUnsafe( )

NAME

taskUnsafe( ) – make the calling task unsafe from deletion

SYNOPSIS

```
STATUS taskUnsafe (void)
```

DESCRIPTION

This routine removes the calling task's protection from deletion. Tasks that attempt to delete a protected task will block until the task is unsafe. When a task becomes unsafe, the deleter will be unblocked and allowed to delete the task.

The **taskUnsafe( )** primitive utilizes a count to keep track of nested calls for task protection. When nesting occurs, the task becomes unsafe only after the outermost **taskUnsafe( )** is executed.

RETURNS

OK.

SEE ALSO

taskLib, **taskSafe( )**, *VxWorks Programmer's Guide: Basic OS*

taskVarAdd( )

NAME

taskVarAdd( ) – add a task variable to a task

SYNOPSIS

```
STATUS taskVarAdd
    (
    int   tid,                /* ID of task to have new variable */
    int * pVar              /* pointer to variable to be switched for task */
    )
```

DESCRIPTION

This routine adds a specified variable *pVar* (4-byte memory location) to a specified task's context. After calling this routine, the variable will be private to the task. The task can access and modify the variable, but the modifications will not appear to other tasks, and other tasks' modifications to that variable will not affect the value seen by the task. This is accomplished by saving and restoring the variable's initial value each time a task switch occurs to or from the calling task.

This facility can be used when a routine is to be spawned repeatedly as several independent tasks. Although each task will have its own stack, and thus separate stack variables, they will all share the same static and global variables. To make a variable *not* shareable, the routine can call **taskVarAdd( )** to make a separate copy of the variable for each task, but all at the same physical address.

Note that task variables increase the task switch time to and from the tasks that own them. Therefore, it is desirable to limit the number of task variables that a task uses. One efficient way to use task variables is to have a single task variable that is a pointer to a dynamically allocated structure containing the task's private data.

EXAMPLE

Assume that three identical tasks were spawned with a routine called **operator( )**. All three use the structure **OP_GLOBAL** for all variables that are specific to a particular incarnation of the task. The following code fragment shows how this is set up:

```
OP_GLOBAL *opGlobal;  /* ptr to operator task's global variables */
void operator
    (
    int opNum         /* number of this operator task */
    )
    {
    if (taskVarAdd (0, (int *)&opGlobal) != OK)
        {
        printErr ("operator%d: can't taskVarAdd opGlobal\n", opNum);
        taskSuspend (0);
        }
    if ((opGlobal = (OP_GLOBAL *) malloc (sizeof (OP_GLOBAL))) == NULL)
        {
        printErr ("operator%d: can't malloc opGlobal\n", opNum);
        taskSuspend (0);
        }
    ...
    }
```

RETURNS

OK, or **ERROR** if memory is insufficient for the task variable descriptor.

SEE ALSO

taskVarLib, **taskVarDelete( )**, **taskVarGet( )**, **taskVarSet( )**

taskVarDelete( )

NAME **taskVarDelete( )** – remove a task variable from a task

SYNOPSIS

```
STATUS taskVarDelete
    (
    int   tid,             /* ID of task whose variable is to be removed */
    int * pVar             /* pointer to task variable to be removed */
    )
```

DESCRIPTION This routine removes a specified task variable, *pVar*, from the specified task's context. The private value of that variable is lost.

RETURNS **OK**, or **ERROR** if the task variable does not exist for the specified task.

SEE ALSO **taskVarLib**, **taskVarAdd( )**, **taskVarGet( )**, **taskVarSet( )**

taskVarGet( )

NAME **taskVarGet( )** – get the value of a task variable

SYNOPSIS

```
int taskVarGet
    (
    int   tid,      /* ID of task whose task variable is to be retrieved */
    int * pVar      /* pointer to task variable */
    )
```

DESCRIPTION This routine returns the private value of a task variable for a specified task. The specified task is usually not the calling task, which can get its private value by directly accessing the variable. This routine is provided primarily for debugging purposes.

RETURNS The private value of the task variable, or **ERROR** if the task is not found or it does not own the task variable.

SEE ALSO **taskVarLib**, **taskVarAdd( )**, **taskVarDelete( )**, **taskVarSet( )**

taskVarInfo( )

NAME **taskVarInfo( )** – get a list of task variables of a task

SYNOPSIS

```
int taskVarInfo
    (
    int      tid,          /* ID of task whose task variable is to be set */
    TASK_VAR varList[],    /* array to hold task variable addresses */
    int      maxVars       /* maximum variables varList can accommodate */
    )
```

DESCRIPTION This routine provides the calling task with a list of all of the task variables of a specified task. The unsorted array of task variables is copied to *varList*.

WARNING: Kernel rescheduling is disabled with **taskLock( )** while task variables are looked up. There is no guarantee that all the task variables are still valid or that new task variables have not been created by the time this routine returns.

RETURNS The number of task variables in the list.

SEE ALSO **taskVarLib**

taskVarInit( )

NAME **taskVarInit( )** – initialize the task variables facility

SYNOPSIS

```
STATUS taskVarInit (void)
```

DESCRIPTION This routine initializes the task variables facility. It installs task switch and delete hooks used for implementing task variables. If **taskVarInit( )** is not called explicitly, **taskVarAdd( )** will call it automatically when the first task variable is added.

After the first invocation of this routine, subsequent invocations have no effect.

WARNING: Order dependencies in task delete hooks often involve task variables. If a facility uses task variables and has a task delete hook that expects to use those task variables, the facility's delete hook must run before the task variables' delete hook. Otherwise, the task variables will be deleted by the time the facility's delete hook runs.

T

VxWorks is careful to run the delete hooks in reverse of the order in which they were installed. Any facility that has a delete hook that will use task variables can guarantee proper ordering by calling **taskVarInit( )** before adding its own delete hook.

Note that this is not an issue in normal use of task variables. The issue only arises when adding another task delete hook that uses task variables.

Caution should also be taken when adding task variables from within create hooks. If the task variable package has not been installed via **taskVarInit( )**, the create hook attempts to create a create hook, and that may cause system failure. To avoid this situation, **taskVarInit( )** should be called during system initialization from the root task, **usrRoot( )**, in **usrConfig.c**.

RETURNS

OK, or **ERROR** if the task switch/delete hooks could not be installed.

SEE ALSO

taskVarLib

taskVarSet( )

NAME

taskVarSet( ) – set the value of a task variable

SYNOPSIS

```
STATUS taskVarSet
    (
    int   tid,    /* ID of task whose task variable is to be set */
    int * pVar,   /* pointer to task variable to be set for this task */
    int   value   /* new value of task variable */
    )
```

DESCRIPTION

This routine sets the private value of the task variable for a specified task. The specified task is usually not the calling task, which can set its private value by directly modifying the variable. This routine is provided primarily for debugging purposes.

RETURNS

OK, or **ERROR** if the task is not found or it does not own the task variable.

SEE ALSO

taskVarLib, **taskVarAdd( )**, **taskVarDelete( )**, **taskVarGet( )**

tcpDebugShow( )

NAME **tcpDebugShow( )** – display debugging information for the TCP protocol

SYNOPSIS
```
void tcpDebugShow
    (
    int numPrint,            /* no. of entries to print, default (0) = 20 */
    int verbose              /* 1 = verbose */
    )
```

DESCRIPTION This routine displays debugging information for the TCP protocol. To include TCP debugging facilities, define **INCLUDE_TCP_DEBUG** when building the system image. To enable information gathering, turn on the **SO_DEBUG** option for the relevant socket(s).

RETURNS N/A

SEE ALSO **tcpShow**

tcpShowInit( )

NAME **tcpShowInit( )** – initialize TCP show routines

SYNOPSIS
```
void tcpShowInit (void)
```

DESCRIPTION This routine links the TCP show facility into the VxWorks system. These routines are included automatically if **INCLUDE_TCP_SHOW** is defined.

RETURNS N/A

SEE ALSO **tcpShow**

T

tcpstatShow( )

NAME **tcpstatShow( )** – display all statistics for the TCP protocol

SYNOPSIS

```
void tcpstatShow (void)
```

DESCRIPTION This routine displays detailed statistics for the TCP protocol.

RETURNS N/A

SEE ALSO **tcpShow**

td( )

NAME **td( )** – delete a task

SYNOPSIS

```
void td
    (
    int taskNameOrId          /* task name or task ID */
    )
```

DESCRIPTION This command deletes a specified task. It simply calls **taskDelete( )**.

RETURNS N/A

SEE ALSO **usrLib**, **taskDelete( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

telnetdExit( )

NAME **telnetdExit( )** – close an active **telnet** session

SYNOPSIS
```
void telnetdExit
    (
    UINT32 sessionId          /* identifies the session to be deleted */
    )
```

DESCRIPTION This routine supports the session exit command for a command interpreter (such as **logout( )** for the VxWorks shell). Depending on the **TELNETD_TASKFLAG** setting, it causes the associated input and output tasks to restart or exit. *sessionId* must match a value provided to the command interpreter with the **REMOTE_START** option.

RETURNS N/A.

SEE ALSO **telnetdLib**

telnetdInit( )

NAME **telnetdInit( )** – initialize the **telnet** services

SYNOPSIS
```
STATUS telnetdInit
    (
    int  numClients,  /* maximum number of simultaneous sessions */
    BOOL staticFlag   /* TRUE: create all tasks in advance of any clients */
    )
```

DESCRIPTION This routine initializes the **telnet** server, which supports remote login to VxWorks via the **telnet** protocol. It is called automatically when the configuration macro **INCLUDE_TELNET** is defined. The **telnet** server supports simultaneous client sessions up to the limit specified by the **TELNETD_MAX_CLIENTS** setting provided in the *numClients* argument. The *staticFlag* argument is equal to the **TELNETD_TASKFLAG** setting. It allows the server to create all of the secondary input and output tasks and allocate all required resources in advance of any connection. The default value of **FALSE** causes the server to spawn a task pair and create the associated data structures after each new connection.

RETURNS **OK**, or **ERROR** if initialization fails

SEE ALSO **telnetdLib**

T

telnetdParserSet( )

NAME

telnetdParserSet( ) – specify a command interpreter for **telnet** sessions

SYNOPSIS

```
STATUS telnetdParserSet
    (
    FUNCPTR pParserCtrlRtn    /* provides parser's file descriptors */
    )
```

DESCRIPTION

This routine provides the ability to handle **telnet** connections using a custom command interpreter or the default VxWorks shell. It is called automatically during system startup (when the configuration macro **INCLUDE_TELNET** is defined) to connect clients to the command interpreter specified in the **TELNETD_PARSER_HOOK** parameter. The command interpreter in use when the **telnet** server start scan never be changed.

The *pParserCtrlRtn* argument provides a routine using the following interface:

```
STATUS parserControlRtn
    (
    int telnetdEvent,/* start or stop a telnet session */
    UINT32 sessionId,/* a unique session identifier */
    int ioFd         /* file descriptor for character i/o */
    )
```

The **telnet** server calls the control routine with a *telnetdEvent* parameter of **REMOTE_INIT** during initialization. The **telnet** server then calls the control routine with a *telnetdEvent* parameter of **REMOTE_START** when a client establishes a new connection. The *sessionId* parameter provides a unique identifier for the session.

In the default configuration, the **telnet** server calls the control routine with a *telnetdEvent* parameter of **REMOTE_STOP** when a session ends.

The **telnet** server does not call the control routine when a session ends if it is configured to spawn all tasks and allocate all resources in advance of any connections. The associated file descriptors will be reused by later clients and cannot be released. In that case, the **REMOTE_STOP** operation only occurs to allow the command interpreter to close those files when the server encounters a fatal error.

RETURNS

OK if parser control routine installed, or **ERROR** otherwise.

SEE ALSO

telnetdLib

telnetdStart( )

NAME

telnetdStart( ) – initialize the **telnet** services

SYNOPSIS

```
STATUS telnetdStart
    (
    int port                  /* target port for accepting connections */
    )
```

DESCRIPTION

Following the **telnet** server initialization, this routine creates a socket for accepting remote connections and spawns the primary **telnet** server task. It executes automatically during system startup when the **INCLUDE_TELNET** configuration macro is defined since a parser control routine is available. The server will not accept connections otherwise.

By default, the server will spawn a pair of secondary input and output tasks after each client connection. Changing the **TELNETD_TASKFLAG** setting to **TRUE** causes this routine to create all of those tasks in advance of any connection. In that case, it calls the current parser control routine repeatedly to obtain file descriptors for each possible client based on the *numClients* argument to the initialization routine. The server will not start if the parser control routine returns **ERROR**.

The **TELNETD_PORT** constant provides the *port* argument, which assigns the port where the server accepts connections. The default value is the standard setting of 23.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

OK, or **ERROR** if startup fails

SEE ALSO

telnetdLib

telnetdStaticTaskInitializationGet( )

NAME

telnetdStaticTaskInitializationGet( ) – report whether tasks were pre-started by **telnetd**

SYNOPSIS

```
BOOL telnetdStaticTaskInitializationGet ()
```

DESCRIPTION

This routine is called by a custom shell partser library to determine whether a shell is to be spawned at the time a connection is requested.

RETURNS

TRUE, if all tasks are pre-spawned; **FALSE**, if tasks are spawned at the time a connection is requested.

SEE ALSO

telnetdLib, **telnetdInit( )**, **telnetdParserSet( )**

tffsBootImagePut( )

NAME

tffsBootImagePut( ) – write to the boot-image region of the flash device

SYNOPSIS

```
STATUS tffsBootImagePut
    (
    int    driveNo,           /* TFFS drive number */
    int    offset,            /* offset in the flash chip/card */
    char * filename           /* binary format of the bootimage */
    )
```

DESCRIPTION

This routine writes an input stream to the boot-image region (if any) of a flash memory device. Typically, the input stream contains a boot image, such as the VxWorks boot image, but you are free to use this function to write any data needed. The size of the boot-image region is set by the **tffsDevFormat( )** call (or the **sysTffsFormat( )** call, a BSP-specific helper function that calls **tffsDevFormat( )** internally) that formats the flash device for use with TrueFFS.

If **tffsBootImagePut( )** is used to put a VxWorks boot image in flash, you should not use the s-record version of the boot image typically produced by make. Instead, you should take the pre s-record version (usually called **bootrom** instead of **bootrom.hex**), and filter out its loader header information using an *xxx***ToBin** utility. For example:

```
elfToBin < bootrom > bootrom.bin
```

Use the resulting **bootrom.bin** as input to **tffsBootImagePut( )**.

The discussion above assumes that you want only to use the flash device to store a VxWorks image that is retrieved from the flash device and then run out of RAM. However, because it is possible to map many flash devices directly into the target's memory, it is also possible run the VxWorks image from flash memory, although there are some restrictions:

– The flash device must be non-NAND.

– Only the text segment of the VxWorks image (**vxWorks.res_rom**) may run out

of flash memory. The data segment of the image must reside in standard RAM.

– No part of the flash device may be erased while the VxWorks image is running

from flash memory.

Because TrueFFS garbage collection triggers an erase, this last restriction means that you cannot run a VxWorks boot image out of a flash device that must also support a writable file system (although a read-only file system is **OK**).

This last restriction arises from the way in which flash devices are constructed. The current physical construction of flash memory devices does not allow access to the device while an erase is in progress anywhere on the flash device. As a result, if TrueFFS tries to erase a portion of the flash device, the entire device becomes inaccessible to all other users. If that other user happens to be the VxWorks image looking for its next instruction, the VxWorks image crashes.

RETURNS **OK** or **ERROR**

SEE ALSO **tffsConfig**

tffsDevCreate( )

NAME **tffsDevCreate( )** – create a TrueFFS block device suitable for use with dosFs

SYNOPSIS

```
BLK_DEV * tffsDevCreate
    (
    int tffsDriveNo,          /* TFFS drive number (0 - DRIVES-1) */
    int removableMediaFlag    /* 0 - nonremovable flash media */
    )
```

DESCRIPTION This routine creates a TFFS block device on top of a flash device. It takes as arguments a drive number, determined from the order in which the socket components were registered, and a flag integer that indicates whether the medium is removable or not. A zero indicates a non removable medium. A one indicates a removable medium. If you intend to mount dosFs on this block device, you probably do not want to call **tffsDevCreate( )**, but should call **usrTffsConfig( )** instead. Internally, **usrTffsConfig( )** calls **tffsDevCreate( )** for you. It then does everything necessary (such as calling the **dosFsDevInit( )** routine) to mount dosFs on the just created block device.

RETURNS **BLK_DEV** pointer, or **NULL** if it failed.

SEE ALSO **tffsDrv**

tffsDevFormat( )

NAME **tffsDevFormat( )** – format a flash device for use with TrueFFS

SYNOPSIS

```
STATUS tffsDevFormat
    (
    int tffsDriveNo,          /* TrueFFS drive number (0 - DRIVES-1) */
    int arg                   /* pointer to tffsDevFormatParams structure */
    )
```

DESCRIPTION This routine formats a flash device for use with TrueFFS. It takes two parameters, a drive number and a pointer to a device format structure. This structure describes how the volume should be formatted. The structure is defined in **dosformt.h**. The drive number is assigned in the order that the socket component for the device was registered.

The format process marks each erase unit with an Erase Unit Header (EUH) and creates the physical and virtual Block Allocation Maps (BAM) for the device. The erase units reserved for the "boot-image" are skipped and the first EUH is placed at number (boot-image length - 1). To write to the boot-image region, call **tffsBootImagePut( )**.

WARNING: If any of the erase units in the boot-image region contains an erase unit header from a previous format call (this can happen if you reformat a flash device specifying a larger boot region) TrueFFS fails to mount the device. To fix this problem, use **tffsRawio( )** to erase the problem erase units (thus removing the outdated EUH).

The macro **TFFS_STD_FORMAT_PARAMS** defines the default values used for formatting a flask disk device. If the second argument to this routine is zero, **tffsDevFormat( )** uses these default values.

RETURNS **OK**, or **ERROR** if it failed.

SEE ALSO **tffsDrv**

tffsDevOptionsSet( )

NAME **tffsDevOptionsSet( )** – set TrueFFS volume options

SYNOPSIS

```
STATUS tffsDevOptionsSet
    (
    TFFS_DEV * pTffsDev        /* pointer to device descriptor */
    )
```

DESCRIPTION This routine is intended to set various TrueFFS volume options. At present it only disables FAT monitoring. If VxWorks long file names are to be used with TrueFFS, FAT monitoring must be turned off.

RETURNS **OK**, or **ERROR** if it failed.

SEE ALSO **tffsDrv**

tffsDrv( )

NAME **tffsDrv( )** – initialize the TrueFFS system

SYNOPSIS

```
STATUS tffsDrv (void)
```

DESCRIPTION This routine sets up the structures, the global variables, and the mutual exclusion semaphore needed to manage TrueFFS. This call also registers socket component drivers for all the flash devices attached to your target.

Because **tffsDrv( )** is the call that initializes the TrueFFS system, this function must be called (exactly once) before calling any other TrueFFS utilities, such as **tffsDevFormat( )** or **tffsDevCreate( )**. Typically, the call to **tffsDrv( )** is handled for you automatically. If you defined **INCLUDE_TFFS** in your BSP's **config.h**, the call to **tffsDrv( )** is made from **usrRoot( )**. If your BSP's **config.h** defines **INCLUDE_PCMCIA**, the call to **tffsDrv( )** is made from **pccardTffsEnabler( )**.

RETURNS **OK**, or **ERROR** if it fails.

SEE ALSO **tffsDrv**

tffsRawio( )

NAME

tffsRawio( ) – low level I/O access to flash components

SYNOPSIS

```
STATUS tffsRawio
    (
    int tffsDriveNo,          /* TrueFFS drive number (0 - DRIVES-1) */
    int functionNo,           /* TrueFFS function code */
    int arg0,                 /* argument 0 */
    int arg1,                 /* argument 1 */
    int arg2                  /* argument 2 */
    )
```

DESCRIPTION

Use the utilities provided by this routine with the utmost care. If you use these routines carelessly, you risk data loss as well as permanent physical damage to the flash device.

This routine is a gateway to a series of utilities (listed below). Functions such as **mkbootTffs( )** and **tffsBootImagePut( )** use these **tffsRawio( )** utilities to write boot sector information. The functions for physical read, write, and erase are made available with the intention that they be used on erase units allocated to the boot-image region by **tffsDevFormat( )**. Using these functions elsewhere could be dangerous.

The *arg0*, *arg1*, and *arg2* parameters to **tffsRawio( )** are interpreted differently depending on the function number you specify for *functionNo*. The drive number is determined by the order in which the socket components were registered.

Function Name	arg0	arg1	arg2
TFFS_GET_PHYSICAL_INFO	user buffer address	N/A	N/A
TFFS_PHYSICAL_READ	address to read	byte count	user buffer address
TFFS_PHYSICAL_WRITE	address to write	byte count	user buffer address
TFFS_PHYSICAL_ERASE	first unit	number of units	N/A
TFFS_ABS_READ	sector number	number of sectors	user buffer address
TFFS_ABS_WRITE	sector number	number of sectors	user buffer address
TFFS_ABS_DELETE	sector number	number of sectors	N/A
TFFS_DEFRAGMENT_VOLUME	number of sectors	user buffer address	N/A

TFFS_GET_PHYSICAL_INFO writes the flash type, erasable block size, and media size to the user buffer specified in *arg0*.

TFFS_PHYSICAL_READ reads *arg1* bytes from *arg0* and writes them to the buffer specified by *arg2*.

TFFS_PHYSICAL_WRITE copies *arg1* bytes from the *arg2* buffer and writes them to the flash memory location specified by *arg0*. This aborts if the volume is already mounted to prevent the versions of translation data in memory and in flash from going out of synchronization.

TFFS_PHYSICAL_ERASE erases *arg1* erase units, starting at the erase unit specified in *arg0*. This aborts if the volume is already mounted to prevent the versions of translation data in memory and in flash from going out of synchronization.

TFFS_ABS_READ reads *arg1* sectors, starting at sector *arg0*, and writes them to the user buffer specified in *arg2*.

TFFS_ABS_WRITE takes data from the *arg2* user buffer and writes *arg1* sectors of it to the flash location starting at sector *arg0*.

TFFS_ABS_DELETE deletes *arg1* sectors of data starting at sector *arg0*.

TFFS_DEFRAGMENT_VOLUME calls the defragmentation routine with the minimum number of sectors to be reclaimed, *arg0*, and writes the actual number reclaimed in the user buffer by *arg1*. Calling this function through some low priority task will make writes more deterministic. No validation is done of the user specified address fields, so the functions assume they are writable. If the address is invalid, you could see bus errors or segmentation faults.

RETURNS **OK**, or **ERROR** if it failed.

SEE ALSO **tffsDrv**

---

tffsShow( )

NAME **tffsShow( )** – show device information on a specific socket interface

SYNOPSIS

```
void tffsShow
    (
    int driveNo                /* TFFS drive number */
    )
```

DESCRIPTION This routine prints device information on the specified socket interface. This information is particularly useful when trying to determine the number of Erase Units required to contain a boot image. The field called *unitSize* reports the size of an Erase Unit.

If the process of getting physical information fails, an error code is printed. The error codes can be found in **flbase.h**.

RETURNS N/A

SEE ALSO **tffsConfig**

T

tffsShowAll( )

NAME

tffsShowAll( ) – show device information on all socket interfaces

SYNOPSIS

```
void tffsShowAll (void)
```

DESCRIPTION

This routine prints device information on all socket interfaces.

RETURNS

N/A

SEE ALSO

tffsConfig

tftpCopy( )

NAME

tftpCopy( ) – transfer a file via TFTP

SYNOPSIS

```
STATUS tftpCopy
    (
    char * pHost,             /* host name or address */
    int    port,              /* optional port number */
    char * pFilename,         /* remote filename */
    char * pCommand,          /* TFTP command */
    char * pMode,             /* TFTP transfer mode */
    int    fd                 /* fd to put/get data */
    )
```

DESCRIPTION

This routine transfers a file using the TFTP protocol to or from a remote system. *pHost* is the remote server name or Internet address. A non-zero value for *port* specifies an alternate TFTP server port (zero means use default TFTP port number (69)). *pFilename* is the remote file name. *pCommand* specifies the TFTP command, which can be either "put" or "get". *pMode* specifies the mode of transfer, which can be "ascii", "netascii", "binary", "image", or "octet".

fd is a file descriptor from which to read/write the data from or to the remote system. For example, if the command is "get", the remote data will be written to *fd*. If the command is "put", the data to be sent is read from *fd*. The caller is responsible for managing *fd*. That is, *fd* must be opened prior to calling **tftpCopy( )** and closed up on completion.

EXAMPLE

The following sequence gets an ASCII file **/folk/vw/xx.yy** on host "congo" and stores it to a local file called **localfile**:

```
-> fd = open ("localfile", 0x201, 0644)
-> tftpCopy ("congo", 0, "/folk/vw/xx.yy", "get", "ascii", fd)
-> close (fd)
```

RETURNS

OK, or **ERROR** if unsuccessful.

ERRNO

S_tftpLib_INVALID_COMMAND

SEE ALSO

tftpLib, **ftpLib**

tftpdDirectoryAdd( )

NAME

tftpdDirectoryAdd( ) – add a directory to the access list

SYNOPSIS

```
STATUS tftpdDirectoryAdd
    (
    char * fileName           /* name of directory to add to access list */
    )
```

DESCRIPTION

This routine adds the specified directory name to the access list for the TFTP server.

RETURNS

N/A

SEE ALSO

tftpdLib

T

tftpdDirectoryRemove( )

NAME

tftpdDirectoryRemove( ) – delete a directory from the access list

SYNOPSIS

```
STATUS tftpdDirectoryRemove
    (
    char * fileName           /* name of directory to add to access list */
    )
```

DESCRIPTION

This routine deletes the specified directory name from the access list for the TFTP server.

RETURNS N/A

SEE ALSO **tftpdLib**

tftpdInit( )

NAME **tftpdInit( )** – initialize the TFTP server task

SYNOPSIS

```
STATUS tftpdInit
    (
    int    stackSize,         /* stack size for the tftpdTask */
    int    nDirectories,      /* number of directories allowed read */
    char * *directoryNames,   /* array of dir names */
    BOOL   noControl,         /* TRUE if no access control required */
    int    maxConnections
    )
```

DESCRIPTION

This routine will spawn a new TFTP server task, if one does not already exist. If a TFTP server task is running already, **tftpdInit( )** will simply return an **ERROR** value without creating a new task.

To change the default stack size for the TFTP server task, use the *stackSize* parameter. The task stack size should be set to a large enough value for the needs of your application - use **checkStack( )** to evaluate your stack usage. The default size is set in the global variable **tftpdTaskStackSize**. Setting *stackSize* to zero will result in the stack size being set to this default.

To set the maximum number of simultaneous TFTP connections (each with its own transfer identifier or TID), set the *maxConnections* parameter. More information on this is found in RFC 1350 ("The TFTP Protocol (Revision 2)"). Setting *maxConnections* to zero will result in the maximum number of connections being set to the default, which is 10.

If *noControl* is **TRUE**, the server will be set up to transfer any file in any location. Otherwise, it will only transfer files in the directories in **/tftpboot** or the *nDirectories* directories in the *directoryNames* list, and will send an access violation error to clients that attempt to access files outside of these directories.

By default, *noControl* is **FALSE**, *directoryNames* is empty, *nDirectories* is zero, and access is restricted to the **/tftpboot** directory.

Directories can be added to the access list after initialization by using the **tftpdDirectoryAdd( )** routine.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

OK, or **ERROR** if a new TFTP task cannot be created.

SEE ALSO

tftpdLib

---

tftpdTask( )

NAME

tftpdTask( ) – TFTP server daemon task

SYNOPSIS

```
STATUS tftpdTask
    (
    int     nDirectories,      /* number of dirs allowed access */
    char * *directoryNames,   /* array of directory names */
    int     maxConnections     /* max number of simultan. connects */
    )
```

DESCRIPTION

This routine processes incoming TFTP client requests by spawning a new task for each connection that is set up. This routine is called by **tftpdInit( )**.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

OK, or **ERROR** if the task returns unexpectedly.

SEE ALSO

tftpdLib

tftpGet( )

NAME

tftpGet( ) – get a file from a remote system

SYNOPSIS

```
STATUS tftpGet
    (
    TFTP_DESC * pTftpDesc,     /* TFTP descriptor */
    char *      pFilename,     /* remote filename */
    int         fd,            /* file descriptor */
    int         clientOrServer /* which side is calling */
    )
```

DESCRIPTION

This routine gets a file from a remote system via TFTP. *pFilename* is the filename. *fd* is the file descriptor to which the data is written. *pTftpDesc* is a pointer to the TFTP descriptor. The **tftpPeerSet( )** routine must be called prior to calling this routine.

RETURNS

OK, or **ERROR** if unsuccessful.

ERRNO

S_tftpLib_INVALID_DESCRIPTOR
S_tftpLib_INVALID_ARGUMENT
S_tftpLib_NOT_CONNECTED

SEE ALSO

tftpLib

tftpInfoShow( )

NAME

tftpInfoShow( ) – get TFTP status information

SYNOPSIS

```
STATUS tftpInfoShow
    (
    TFTP_DESC * pTftpDesc     /* TFTP descriptor */
    )
```

DESCRIPTION

This routine prints information associated with TFTP descriptor *pTftpDesc*.

EXAMPLE

A call to **tftpInfoShow( )** might look like:

```
-> tftpInfoShow (tftpDesc)
       Connected to yuba [69]
       Mode: netascii  Verbose: off  Tracing: off
       Rexmt-interval: 5 seconds, Max-timeout: 25 seconds
```

RETURNS **OK**, or **ERROR** if unsuccessful.

ERRNO **S_tftpLib_INVALID_DESCRIPTOR**

SEE ALSO **tftpLib**

tftpInit( )

NAME **tftpInit( )** – initialize a TFTP session

SYNOPSIS

```
TFTP_DESC * tftpInit (void)
```

DESCRIPTION This routine initializes a TFTP session by allocating and initializing a TFTP descriptor. It sets the default transfer mode to "netascii".

RETURNS A pointer to a TFTP descriptor if successful, otherwise **NULL**.

SEE ALSO **tftpLib**

tftpModeSet( )

NAME **tftpModeSet( )** – set the TFTP transfer mode

SYNOPSIS

```
STATUS tftpModeSet
    (
    TFTP_DESC * pTftpDesc,     /* TFTP descriptor */
    char *      pMode          /* TFTP transfer mode */
    )
```

DESCRIPTION This routine sets the transfer mode associated with the TFTP descriptor *pTftpDesc*. *pMode* specifies the transfer mode, which can be "netascii", "binary", "image", or "octet". Although recognized, these modes actually translate into either octet or netascii.

RETURNS **OK**, or **ERROR** if unsuccessful.

ERRNO **S_tftpLib_INVALID_DESCRIPTOR**, **S_tftpLib_INVALID_ARGUMENT**, **S_tftpLib_INVALID_MODE**

SEE ALSO **tftpLib**

T

tftpPeerSet( )

NAME

tftpPeerSet( ) – set the TFTP server address

SYNOPSIS

```
STATUS tftpPeerSet
    (
    TFTP_DESC * pTftpDesc,    /* TFTP descriptor */
    char *      pHostname,    /* server name/address */
    int         port          /* port number */
    )
```

DESCRIPTION

This routine sets the TFTP server (peer) address associated with the TFTP descriptor *pTftpDesc*. *pHostname* is either the TFTP server name (*e.g.*, "congo") or the server Internet address (*e.g.*, "90.3"). A non-zero value for *port* specifies the server port number (zero means use the default TFTP server port number (69)).

RETURNS

OK, or **ERROR** if unsuccessful.

ERRNO

S_tftpLib_INVALID_DESCRIPTOR
S_tftpLib_INVALID_ARGUMENT
S_tftpLib_UNKNOWN_HOST

SEE ALSO

tftpLib

tftpPut( )

NAME

tftpPut( ) – put a file to a remote system

SYNOPSIS

```
STATUS tftpPut
    (
    TFTP_DESC * pTftpDesc,     /* TFTP descriptor */
    char *      pFilename,     /* remote filename */
    int         fd,            /* file descriptor */
    int         clientOrServer /* which side is calling */
    )
```

DESCRIPTION

This routine puts data from a local file (descriptor) to a file on the remote system. *pTftpDesc* is a pointer to the TFTP descriptor. *pFilename* is the remote filename. *fd* is the file descriptor from which it gets the data. A call to **tftpPeerSet( )** must be made prior to calling this routine.

RETURNS **OK**, or **ERROR** if unsuccessful.

ERRNO **S_tftpLib_INVALID_DESCRIPTOR**
S_tftpLib_INVALID_ARGUMENT
S_tftpLib_NOT_CONNECTED

SEE ALSO **tftpLib**

tftpQuit( )

NAME **tftpQuit( )** – quit a TFTP session

SYNOPSIS

```
STATUS tftpQuit
    (
    TFTP_DESC * pTftpDesc     /* TFTP descriptor */
    )
```

DESCRIPTION This routine closes a TFTP session associated with the TFTP descriptor *pTftpDesc*.

RETURNS **OK**, or **ERROR** if unsuccessful.

ERRNO **S_tftpLib_INVALID_DESCRIPTOR**

SEE ALSO **tftpLib**

tftpSend( )

NAME **tftpSend( )** – send a TFTP message to the remote system

SYNOPSIS

```
int tftpSend
    (
    TFTP_DESC * pTftpDesc,    /* TFTP descriptor */
    TFTP_MSG *  pTftpMsg,     /* TFTP send message */
    int         sizeMsg,      /* send message size */
    TFTP_MSG *  pTftpReply,   /* TFTP reply message */
    int         opReply,      /* reply opcode */
    int         blockReply,   /* reply block number */
    int *       pPort         /* return port number */
    )
```

DESCRIPTION
This routine sends *sizeMsg* bytes of the passed message *pTftpMsg* to the remote system associated with the TFTP descriptor *pTftpDesc*. If *pTftpReply* is not **NULL**, **tftpSend( )** tries to get a reply message with a block number *blockReply* and an opcode *opReply*. If *pPort* is **NULL**, the reply message must come from the same port to which the message was sent. If *pPort* is not **NULL**, the port number from which the reply message comes is copied to this variable.

RETURNS
The size of the reply message, or **ERROR**.

ERRNO
S_tftpLib_TIMED_OUT
S_tftpLib_TFTP_ERROR

SEE ALSO
tftpLib

tftpXfer( )

NAME
tftpXfer( ) – transfer a file via TFTP using a stream interface

SYNOPSIS

```
STATUS tftpXfer
    (
    char * pHost,              /* host name or address */
    int    port,               /* port number */
    char * pFilename,          /* remote filename */
    char * pCommand,           /* TFTP command */
    char * pMode,              /* TFTP transfer mode */
    int *  pDataDesc,          /* return data desc. */
    int *  pErrorDesc          /* return error desc. */
    )
```

DESCRIPTION
This routine initiates a transfer to or from a remote file via TFTP. It spawns a task to perform the TFTP transfer and returns a descriptor from which the data can be read (for "get") or to which it can be written (for "put") interactively. The interface for this routine is similar to **ftpXfer( )** in **ftpLib**.

pHost is the server name or Internet address. A non-zero value for *port* specifies an alternate TFTP server port number (zero means use default TFTP port number (69)). *pFilename* is the remote filename. *pCommand* specifies the TFTP command. The command can be either "put" or "get".

The **tftpXfer( )** routine returns a data descriptor, in *pDataDesc*, from which the TFTP data is read (for "get") or to which is it is written (for "put"). An error status descriptor is returned in the variable *pErrorDesc*. If an error occurs during the TFTP transfer, an error string can be read from this descriptor. After returning successfully from **tftpXfer( )**, the calling application is responsible for closing both descriptors.

If there are delays in reading or writing the data descriptor, it is possible for the TFTP transfer to time out.

EXAMPLE

The following code demonstrates how **tftpXfer( )** may be used:

```
#include "tftpLib.h"
#define BUFFERSIZE        512
int  dataFd;
int  errorFd;
int  num;
char buf [BUFFERSIZE + 1];
if (tftpXfer ("congo", 0, "/usr/fred", "get", "ascii", &dataFd,
              &errorFd) == ERROR)
    return (ERROR);
while ((num = read (dataFd, buf, sizeof (buf))) > 0)
    {
    ....
    }
close (dataFd);
num = read (errorFd, buf, BUFFERSIZE);
if (num > 0)
    {
    buf [num] = '\0';
    printf ("YIKES! An error occurred!:%s\n", buf);
    .....
    }
close (errorFd);
```

RETURNS

OK, or **ERROR** if unsuccessful.

ERRNO

S_tftpLib_INVALID_ARGUMENT

SEE ALSO

tftpLib, **ftpLib**

ti( )

NAME

ti( ) – print complete information from a task's TCB

SYNOPSIS

```
void ti
    (
    int taskNameOrId          /* task name or task ID; 0 = use default */
    )
```

DESCRIPTION

This command prints the task control block (TCB) contents, including registers, for a specified task. If *taskNameOrId* is omitted or zero, the last task referenced is assumed.

The **ti( )** routine uses **taskShow( )**; see the documentation for **taskShow( )** for a description of the output format.

EXAMPLE

The following shows the TCB contents for the shell task:

```
-> ti
  NAME      ENTRY     TID     PRI  STATUS      PC       SP     ERRNO  DELAY
---------- --------- -------- --- --------- -------- -------- ------ -----
tShell     _shell     20efcac   1 READY      201dc90  20ef980      0     0
stack: base 0x20efcac  end 0x20ed59c  size 9532   high 1452   margin 8080
options: 0x1e
VX_UNBREAKABLE      VX_DEALLOC_STACK    VX_FP_TASK         VX_STDIO
D0 =        0   D4 =        0   A0 =        0   A4 =        0
D1 =        0   D5 =        0   A1 =        0   A5 =  203a084   SR =     3000
D2 =        0   D6 =        0   A2 =        0   A6 =  20ef9a0   PC =  2038614
D3 =        0   D7 =        0   A3 =        0   A7 =  20ef980
value = 34536868 = 0x20efda4
```

RETURNS

N/A

SEE ALSO

usrLib, **taskShow( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

tickAnnounce( )

NAME **tickAnnounce( )** – announce a clock tick to the kernel

SYNOPSIS

```
void tickAnnounce (void)
```

DESCRIPTION This routine informs the kernel of the passing of time. It should be called from an interrupt service routine that is connected to the system clock. The most common frequencies are 60Hz or 100Hz. Frequencies in excess of 600Hz are an inefficient use of processor power because the system will spend most of its time advancing the clock. By default, this routine is called by **usrClock( )** in **usrConfig.c**.

RETURNS N/A

SEE ALSO **tickLib**, **kernelLib**, **taskLib**, **semLib**, **wdLib**, *VxWorks Programmer's Guide: Basic OS*

tickGet( )

NAME **tickGet( )** – get the value of the kernel's tick counter

SYNOPSIS

```
ULONG tickGet (void)
```

DESCRIPTION This routine returns the current value of the tick counter. This value is set to zero at startup, incremented by **tickAnnounce( )**, and can be changed using **tickSet( )**.

RETURNS The most recent **tickSet( )** value, plus all **tickAnnounce( )** calls since.

SEE ALSO **tickLib**, **tickSet( )**, **tickAnnounce( )**

T

tickSet( )

NAME

tickSet( ) – set the value of the kernel's tick counter

SYNOPSIS

```
void tickSet
    (
    ULONG ticks                /* new time in ticks */
    )
```

DESCRIPTION

This routine sets the internal tick counter to a specified value in ticks. The new count will be reflected by **tickGet( )**, but will not change any delay fields or timeouts selected for any tasks. For example, if a task is delayed for ten ticks, and this routine is called to advance time, the delayed task will still be delayed until ten **tickAnnounce( )** calls have been made.

RETURNS

N/A

SEE ALSO

tickLib, **tickGet( )**, **tickAnnounce( )**

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 to which *timer* points.

INCLUDE FILES

time.h

RETURNS

The current calendar time in seconds, or **ERROR** (-1) if the calendar time is not available.

SEE ALSO

ansiTime, **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: Non-POSIX.

RETURNS 0 (**OK**), or -1 (**ERROR**) if *timerid* is invalid.

ERRNO **EINVAL**

SEE ALSO **timerLib**

timer_connect( )

NAME **timer_connect( )** – connect a user routine to the timer signal

SYNOPSIS

```
int timer_connect
    (
    timer_t     timerid,      /* timer ID */
    VOIDFUNCPTR routine,      /* user routine */
    int         arg           /* user argument */
    )
```

DESCRIPTION This routine sets the specified *routine* to be invoked with *arg* when fielding a signal indicated by the timer's *evp* signal number, or if *evp* is **NULL**, when fielding the default signal (**SIGALRM**).

The signal handling routine should be declared as:

```
    void my_handler
        (
        timer_t timerid,      /* expired timer ID */
        int arg               /* user argument    */
        )
```

T

NOTE: Non-POSIX.

RETURNS 0 (**OK**), or -1 (**ERROR**) if the timer is invalid or cannot bind the signal handler.

ERRNO EINVAL

SEE ALSO **timerLib**

timer_create( )

NAME **timer_create( )** – allocate a timer using the specified clock for a timing base (POSIX)

SYNOPSIS

```
int timer_create
    (
    clockid_t          clock_id, /* clock ID (always CLOCK_REALTIME) */
    struct sigevent * evp,       /* user event handler */
    timer_t *          pTimer    /* ptr to return value */
    )
```

DESCRIPTION This routine returns a value in *pTimer* that identifies the timer in subsequent timer requests. The *evp* argument, if non-**NULL**, points to a **sigevent** structure, which is allocated by the application and defines the signal number and application-specific data to be sent to the task when the timer expires. If *evp* is **NULL**, a default signal (**SIGALRM**) is queued to the task, and the signal data is set to the timer ID. Initially, the timer is disarmed.

RETURNS 0 (**OK**), or -1 (**ERROR**) if too many timers already are allocated or the signal number is invalid.

ERRNO EMTIMERS, EINVAL, ENOSYS, EAGAIN, S_memLib_NOT_ENOUGH_MEMORY

SEE ALSO **timerLib**, **timer_delete( )**

timer_delete( )

NAME **timer_delete( )** – remove a previously created timer (POSIX)

SYNOPSIS

```
int timer_delete
    (
    timer_t timerid           /* timer ID */
    )
```

DESCRIPTION This routine removes a timer.

RETURNS 0 (**OK**), or -1 (**ERROR**) if *timerid* is invalid.

ERRNO EINVAL

SEE ALSO **timerLib**, **timer_create( )**

timer_getoverrun( )

NAME **timer_getoverrun( )** – return the timer expiration overrun (POSIX)

SYNOPSIS

```
int timer_getoverrun
    (
    timer_t timerid           /* timer ID */
    )
```

DESCRIPTION This routine returns the timer expiration overrun count for *timerid*, when called from a timer expiration signal catcher. The overrun count is the number of extra timer expirations that have occurred, up to the implementation-defined maximum **_POSIX_DELAYTIMER_MAX**. If the count is greater than the maximum, it returns the maximum.

RETURNS The number of overruns, or **_POSIX_DELAYTIMER_MAX** if the count equals or is greater than **_POSIX_DELAYTIMER_MAX**, or -1 (**ERROR**) if *timerid* is invalid.

ERRNO EINVAL, ENOSYS

SEE ALSO **timerLib**

timer_gettime( )

NAME

timer_gettime( ) – get the remaining time before expiration and the reload value (POSIX)

SYNOPSIS

```
int timer_gettime
    (
    timer_t              timerid, /* timer ID */
    struct itimerspec * value    /* where to return remaining time */
    )
```

DESCRIPTION

This routine gets the remaining time and reload value of a specified timer. Both values are copied to the *value* structure.

RETURNS

0 (**OK**), or -1 (**ERROR**) if *timerid* is invalid.

ERRNO

EINVAL

SEE ALSO

timerLib

timer_settime( )

NAME

timer_settime( ) – set the time until the next expiration and arm timer (POSIX)

SYNOPSIS

```
int timer_settime
    (
    timer_t                  timerid, /* timer ID */
    int                      flags,   /* absolute or relative */
    const struct itimerspec * value,  /* time to be set */
    struct itimerspec *      ovalue   /* 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 task that sets the timer receives the signal; in other words, the *taskId* is noted. If a timer is set by an ISR, the signal is delivered to the task that created the timer.

The reload value of the timer is set to the value specified by the **.it_interval** field of *value*. When a timer is armed with a nonzero **.it_interval** a periodic timer is set up.

Time values that are between two consecutive non-negative integer multiples of the resolution of the specified timer are rounded up to the larger multiple of the resolution.

If *ovalue* is non-**NULL**, the routine stores a value representing the previous amount of time before the timer would have expired. Or if the timer is disarmed, the routine stores zero, together with the previous timer reload value. The *ovalue* parameter is the same value as that returned by **timer_gettime( )** and is subject to the timer resolution.

WARNING: If **clock_settime( )** is called to reset the absolute clock time after a timer has been set with **timer_settime( )**, and if *flags* is equal to **TIMER_ABSTIME**, then the timer will behave unpredictably. If you must reset the absolute clock time after setting a timer, do not use *flags* equal to **TIMER_ABSTIME**.

RETURNS

0 (**OK**), or -1 (**ERROR**) if *timerid* is invalid, the number of nanoseconds specified by *value* is less than 0 or greater than or equal to 1,000,000,000, or the time specified by *value* exceeds the maximum allowed by the timer.

ERRNO

EINVAL

SEE ALSO

timerLib

timex( )

NAME

timex( ) – time a single execution of a function or functions

SYNOPSIS

```
void timex
    (
    FUNCPTR func,            /* function to time (optional) */
    int     arg1,            /* first of up to 8 args to call function */
                             /* with (optional) */
    int     arg2,
    int     arg3,
    int     arg4,
    int     arg5,
    int     arg6,
    int     arg7,
    int     arg8
    )
```

DESCRIPTION

This routine times a single execution of a specified function with up to eight of the function's arguments. If no function is specified, it times the execution of the current list of functions to be timed, which is created using **timexFunc( )**, **timexPre( )**, and **timexPost( )**. If **timex( )** is executed with a function argument, the entire current list is replaced with the single specified function.

When execution is complete, **timex( )** displays the execution time. If the execution was so fast relative to the clock rate that the time is meaningless (error> 50%), a warning message is printed instead. In such cases, use **timexN( )**.

RETURNS

N/A

SEE ALSO

timexLib, **timexFunc( )**, **timexPre( )**, **timexPost( )**, **timexN( )**

timexClear( )

NAME

timexClear( ) – clear the list of function calls to be timed

SYNOPSIS

```
void timexClear (void)
```

DESCRIPTION

This routine clears the current list of functions to be timed.

RETURNS

N/A

SEE ALSO

timexLib

timexFunc( )

NAME

timexFunc( ) – specify functions to be timed

SYNOPSIS

```
void timexFunc
    (
    int     i,           /* function number in list (0..3) */
    FUNCPTR func,        /* function to be added (NULL if to be deleted) */
    int     arg1,        /* first of up to 8 args to call function with */
    int     arg2,
    int     arg3,
    int     arg4,
```

```
int     arg5,
int     arg6,
int     arg7,
int     arg8
)
```

DESCRIPTION This routine adds or deletes functions in the list of functions to be timed as a group by calls to **timex( )** or **timexN( )**. Up to four functions can be included in the list. The argument *i* specifies the function's position in the sequence of execution (0, 1, 2, or 3). A function is deleted by specifying its sequence number *i* and **NULL** for the function argument *func*.

RETURNS N/A

SEE ALSO **timexLib**, **timex( )**, **timexN( )**

---

timexHelp( )

NAME **timexHelp( )** – display synopsis of execution timer facilities

SYNOPSIS

```
void timexHelp (void)
```

DESCRIPTION This routine displays the following summary of the available execution timer functions:

```
timexHelp                       Print this list.
timex       [func,[args...]]    Time a single execution.
timexN      [func,[args...]]    Time repeated executions.
timexClear                      Clear all functions.
timexFunc   i,func,[args...]    Add timed function number i (0,1,2,3).
timexPre    i,func,[args...]    Add pre-timing function number i.
timexPost   i,func,[args...]    Add post-timing function number i.
timexShow                       Show all functions to be called.
Notes:
  1) timexN() will repeat calls enough times to get
     timing accuracy to approximately 2%.
  2) A single function can be specified with timex() and timexN();
     or, multiple functions can be pre-set with timexFunc().
  3) Up to 4 functions can be pre-set with timexFunc(),
     timexPre(), and timexPost(), i.e., i in the range 0 - 3.
  4) timexPre() and timexPost() allow locking/unlocking, or
     raising/lowering priority before/after timing.
```

T

RETURNS N/A

SEE ALSO **timexLib**

timexInit( )

NAME **timexInit( )** – include the execution timer library

SYNOPSIS

```
void timexInit (void)
```

DESCRIPTION This null routine is provided so that **timexLib** can be linked into the system. If the configuration macro **INCLUDE_TIMEX** is defined, it is called by the root task, **usrRoot( )**, in **usrConfig.c**.

RETURNS N/A

SEE ALSO **timexLib**

timexN( )

NAME **timexN( )** – time repeated executions of a function or group of functions

SYNOPSIS

```
void timexN
    (
    FUNCPTR func,        /* function to time (optional) */
    int     arg1,        /* first of up to 8 args to call function with */
    int     arg2,
    int     arg3,
    int     arg4,
    int     arg5,
    int     arg6,
    int     arg7,
    int     arg8
    )
```

DESCRIPTION This routine times the execution of the current list of functions to be timed in the same manner as **timex( )**; however, the list of functions is called a variable number of times until sufficient resolution is achieved to establish the time with an error less than 2%. (Since each iteration of the list may be measured to a resolution of +/- 1 clock tick, repetitive timings decrease this error to 1/N ticks, where N is the number of repetitions.)

RETURNS N/A

SEE ALSO **timexLib**, **timexFunc( )**, **timex( )**

---

timexPost( )

NAME **timexPost( )** – specify functions to be called after timing

SYNOPSIS

```
void timexPost
    (
    int      i,           /* function number in list (0..3) */
    FUNCPTR func,         /* function to be added (NULL if to be deleted) */
    int      arg1,        /* first of up to 8 args to call function with */
    int      arg2,
    int      arg3,
    int      arg4,
    int      arg5,
    int      arg6,
    int      arg7,
    int      arg8
    )
```

DESCRIPTION This routine adds or deletes functions in the list of functions to be called immediately following the timed functions. A maximum of four functions may be included. Up to eight arguments may be passed to each function.

RETURNS N/A

SEE ALSO **timexLib**

timexPre( )

NAME
timexPre( ) – specify functions to be called prior to timing

SYNOPSIS

```
void timexPre
    (
    int     i,           /* function number in list (0..3) */
    FUNCPTR func,        /* function to be added (NULL if to be deleted) */
    int     arg1,        /* first of up to 8 args to call function with */
    int     arg2,
    int     arg3,
    int     arg4,
    int     arg5,
    int     arg6,
    int     arg7,
    int     arg8
    )
```

DESCRIPTION
This routine adds or deletes functions in the list of functions to be called immediately prior to the timed functions. A maximum of four functions may be included. Up to eight arguments may be passed to each function.

RETURNS
N/A

SEE ALSO
timexLib

timexShow( )

NAME
timexShow( ) – display the list of function calls to be timed

SYNOPSIS

```
void timexShow (void)
```

DESCRIPTION
This routine displays the current list of function calls to be timed. These lists are created by calls to **timexPre( )**, **timexFunc( )**, and **timexPost( )**.

RETURNS
N/A

SEE ALSO
timexLib, **timexPre( )**, **timexFunc( )**, **timexPost( )**

tmpfile( )

NAME **tmpfile( )** – create a temporary binary file (Unimplemented) (ANSI)

SYNOPSIS

```
FILE * tmpfile (void)
```

DESCRIPTION This routine is not be implemented because VxWorks does not close all open files at task exit.

INCLUDE FILES **stdio.h**

RETURNS NULL

SEE ALSO **ansiStdio**

tmpnam( )

NAME **tmpnam( )** – generate a temporary file name (ANSI)

SYNOPSIS

```
char * tmpnam
    (
    char * s                  /* name buffer */
    )
```

DESCRIPTION This routine generates a string that is a valid file name and not the same as the name of an existing file. It generates a different string each time it is called, up to **TMP_MAX** times.

If the argument is a null pointer, **tmpnam( )** leaves its result in an internal static object and returns a pointer to that object. Subsequent calls to **tmpnam( )** may modify the same object. If the argument is not a null pointer, it is assumed to point to an array of at least **L_tmpnam** chars; **tmpnam( )** writes its result in that array and returns the argument as its value.

INCLUDE FILES **stdio.h**

RETURNS A pointer to the file name.

SEE ALSO **ansiStdio**

tolower( )

NAME

tolower( ) – convert an upper-case letter to its lower-case equivalent (ANSI)

SYNOPSIS

```
int tolower
    (
    int c                      /* character to convert */
    )
```

DESCRIPTION

This routine converts an upper-case letter to the corresponding lower-case letter.

INCLUDE FILES

ctype.h

RETURNS

If *c* is an upper-case letter, it returns the lower-case equivalent; otherwise, it returns the argument unchanged.

SEE ALSO

ansiCtype

toupper( )

NAME

toupper( ) – convert a lower-case letter to its upper-case equivalent (ANSI)

SYNOPSIS

```
int toupper
    (
    int c                      /* character to convert */
    )
```

DESCRIPTION

This routine converts a lower-case letter to the corresponding upper-case letter.

INCLUDE FILES

ctype.h

RETURNS

If *c* is a lower-case letter, it returns the upper-case equivalent; otherwise, it returns the argument unchanged.

SEE ALSO

ansiCtype

tr( )

NAME **tr( )** – resume a task

SYNOPSIS

```
void tr
    (
    int taskNameOrId          /* task name or task ID */
    )
```

DESCRIPTION This command resumes the execution of a suspended task. It simply calls **taskResume( )**.

RETURNS N/A

SEE ALSO **usrLib**, **ts( )**, **taskResume( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

trgAdd( )

NAME **trgAdd( )** – add a new trigger to the trigger list

SYNOPSIS

```
TRIGGER_ID trgAdd
    (
    event_t   event,
    int       status,
    int       contextType,
    UINT32    contextId,
    OBJ_ID    objId,
    int       conditional,
    int       condType,
    int *     condEx1,
    int       condOp,
    int       condEx2,
    BOOL      disable,
    TRIGGER * chain,
    int       actionType,
    FUNCPTR   actionFunc,
    BOOL      actionDef,
    int       actionArg
    )
```

T

DESCRIPTION

This routine creates a new trigger and adds it to the proper trigger list. Parameters:

event
: as defined in **eventP.h** for WindView, if given.

status
: the initial status of the trigger (enabled or disabled).

contextType
: the type of context where the event occurs.

contextId
: the ID (if any) of the context where the event occurs.

objectId
: if given and applicable.

conditional
: the indicator that there is a condition on the trigger.

condType
: the indicator that the condition is either a variable or a function.

condEx1
: the first element in the comparison.

condOp
: the type of operator (==, !=, <, <=, <, <=, |, &).

condEx2
: the second element in the comparison (a constant).

disable
: the indicator of whether the trigger must be disabled once it is hit.

chain
: a pointer to another trigger associated to this one (if any).

actionType
: the type of action associated with the trigger (none, func, lib).

actionFunc
: the action associated with the trigger (the function).

actionDef
: the indicator of whether the action can be deferred (deferred is the default).

actionArg
: the argument passed to the function, if any.

Calling **trgAdd( )** while triggering is enabled is not allowed and will return **NULL**.

RETURNS

TRIGGER_ID, or **NULL** if either the trigger ID can not be allocated, or if called whilst triggering is enabled.

SEE ALSO **trgLib**, **trgDelete( )**

trgChainSet( )

NAME **trgChainSet( )** – chains two triggers

SYNOPSIS

```
STATUS trgChainSet
    (
    TRIGGER_ID fromId,
    TRIGGER_ID toId
    )
```

DESCRIPTION This routine chains two triggers together. When the first trigger fires, it calls **trgEnable( )** for the second trigger. The second trigger must be created disabled in order to maintain the correct sequence.

RETURNS **OK** or **ERROR**.

SEE ALSO **trgLib**, **trgEnable( )**

trgDelete( )

NAME **trgDelete( )** – delete a trigger from the trigger list

SYNOPSIS

```
STATUS trgDelete
    (
    TRIGGER_ID trgId
    )
```

DESCRIPTION This routine deletes a trigger by removing it from the trigger list. It also checks that no other triggers are still active. If there are no active triggers and triggering is still on, it turns triggering off.

RETURNS **OK**, or **ERROR** if the trigger is not found.

SEE ALSO **trgLib**, **trgAdd( )**

T

trgDisable( )

NAME
trgDisable( ) – turn a trigger off

SYNOPSIS

```
STATUS trgDisable
    (
    TRIGGER_ID trgId
    )
```

DESCRIPTION
This routine disables a trigger. It also checks to see if there are triggers still active. If this is the last active trigger it sets triggering off.

RETURNS
OK, or **ERROR** if the trigger ID is not found.

SEE ALSO
trgLib, **trgEnable( )**

trgEnable( )

NAME
trgEnable( ) – enable a trigger

SYNOPSIS

```
STATUS trgEnable
    (
    TRIGGER_ID trgId
    )
```

DESCRIPTION
This routine enables a trigger that has been created with **trgAdd( )**. A counter is incremented to keep track of the total number of enabled triggers so that **trgDisable( )** knows when to set triggering off. If the maximum number of enabled triggers is reached, an error is returned.

RETURNS
OK, or **ERROR** if the trigger ID is not found or if the maximum number of triggers has already been enabled.

SEE ALSO
trgLib, **trgDisable( )**

trgEvent( )

NAME **trgEvent( )** – trigger a user-defined event

SYNOPSIS
```
void trgEvent
    (
    event_t evtId             /* event */
    )
```

DESCRIPTION This routine triggers a user event. A trigger must exist and triggering must have been started with **trgOn( )** or from the triggering GUI to use this routine. The *evtId* should be in the range 40000-65535.

RETURNS N/A

SEE ALSO **trgLib**, **dbgLib**, **e( )**

trgLibInit( )

NAME **trgLibInit( )** – initialize the triggering library

SYNOPSIS
```
STATUS trgLibInit (void)
```

DESCRIPTION This routine initializes the trigger class. Triggers are VxWorks objects and therefore require a class to be initialized.

RETURNS **OK** or **ERROR**.

SEE ALSO **trgLib**

trgOff( )

NAME **trgOff( )** – set triggering off

SYNOPSIS

```
void trgOff (void)
```

DESCRIPTION This routine turns triggering off. From this time on, when an event point is hit, no search on triggers is performed.

RETURNS N/A

SEE ALSO **trgLib**, **trgOn( )**

trgOn( )

NAME **trgOn( )** – set triggering on

SYNOPSIS

```
STATUS trgOn (void)
```

DESCRIPTION This routine activates triggering. From this time on, any time an event point is hit, a check for the presence of possible triggers is performed. Start triggering only when needed since some overhead is introduced.

NOTE: If **trgOn( )** is called when there are no triggers in the trigger list, it immediately sets triggering off again. If **trgOn( )** is called with at least one trigger in the list, triggering begins. Triggers should not be added to the list while triggering is on since this can create instability.

RETURNS **OK** or **ERROR**.

SEE ALSO **trgLib**, **trgOff( )**

trgShow( )

NAME **trgShow( )** – show trigger information

SYNOPSIS

```
STATUS trgShow
    (
    TRIGGER_ID trgId,
    int        level
    )
```

DESCRIPTION This routine displays trigger information. If *trgId* is passed, only the summary for that trigger is displayed. If no parameter is passed, the list of existing triggers is displayed with a summary of their state. For example:

```
trgID      Status   EvtID    ActType Action     Dis    Chain
------------------------------------------------------------
0xffedfc   disabled  101          3 0x14e7a4     Y 0xffe088
0xffe088   enabled    55          1 0x10db58     Y      0x0
```

If *level* is 1, then more detailed information is displayed.

EXAMPLE

```
-> trgShow trgId, 1
```

RETURNS **OK**.

SEE ALSO **trgShow**, **trgLib**

trgShowInit( )

NAME **trgShowInit( )** – initialize the trigger show facility

SYNOPSIS

```
void trgShowInit (void)
```

DESCRIPTION This routine links the trigger show facility into the VxWorks system. These routines are included automatically when **INCLUDE_TRIGGER_SHOW** is defined.

RETURNS N/A

SEE ALSO **trgShow**

T

trgWorkQReset( )

NAME

trgWorkQReset( ) – reset the trigger work queue task and queue

SYNOPSIS

```
STATUS trgWorkQReset (void)
```

DESCRIPTION

When a trigger fires, if the associated action requires a function to be called in "safe" mode, a pointer to the required function will be placed on a queue known as the "triggering work queue". A system task "**tActDef**" is spawned to action these requests at task level. Should the user have need to reset this work queue (*e.g.*, if a called task causes an exception which causes the **trgActDef** task to be SUSPENDED, or if the queue gets out of sync and becomes unresponsive), **trgWorkQReset( )** may be called.

Its effect is to delete the trigger work queue task and its associated resources and then recreate them. Any entries pending on the triggering work queue will be lost. Calling this function with triggering on will result in triggering being turned off before the queue reset takes place. It is the responsibility of the user to turn triggering back on.

RETURNS

OK, or **ERROR** if the triggering task and its associated resources cannot be deleted and recreated.

SEE ALSO

trgLib

trunc( )

NAME

trunc( ) – truncate to integer

SYNOPSIS

```
double trunc
    (
    double x  /* value to truncate */
    )
```

DESCRIPTION

This routine discards the fractional part of a double-precision value *x*.

INCLUDE FILES

math.h

RETURNS

The integer portion of *x*, represented in double-precision.

SEE ALSO

mathALib

truncf( )

NAME **truncf( )** – truncate to integer

SYNOPSIS

```
float truncf
    (
    float x   /* value to truncate */
    )
```

DESCRIPTION This routine discards the fractional part of a single-precision value *x*.

INCLUDE FILES **math.h**

RETURNS The integer portion of *x*, represented in single precision.

SEE ALSO **mathALib**

ts( )

NAME **ts( )** – suspend a task

SYNOPSIS

```
void ts
    (
    int taskNameOrId          /* task name or task ID */
    )
```

DESCRIPTION This command suspends the execution of a specified task. It simply calls **taskSuspend( )**.

RETURNS N/A

SEE ALSO **usrLib**, **tr( )**, **taskSuspend( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

T

tsfsUploadPathClose( )

NAME **tsfsUploadPathClose( )** – close the TSFS-socket upload path (Windview)

SYNOPSIS

```
void tsfsUploadPathClose
    (
    UPLOAD_ID upId            /* generic upload-path descriptor */
    )
```

DESCRIPTION This routine closes the TSFS-socket connection to the event receiver on the host.

RETURNS N/A

SEE ALSO **wvTsfsUploadPathLib**, **tsfsUploadPathCreate( )**

tsfsUploadPathCreate( )

NAME **tsfsUploadPathCreate( )** – open an upload path to the host using a TSFS socket (Windview)

SYNOPSIS

```
UPLOAD_ID tsfsUploadPathCreate
    (
    char * ipAddress,         /* server's IP address in .-notation */
    short  port               /* port number to bind to */
    )
```

DESCRIPTION This routine opens a TSFS socket to the host to be used for uploading event data. After successfully establishing this connection, an **UPLOAD_ID** is returned which points to the **TSFS_UPLOAD_DESC** that is passed to **open( )**, **close( )**, **read( )**, etc. for future operations.

RETURNS The **UPLOAD_ID**, or **NULL** if the connection cannot be completed or not enough memory is available.

SEE ALSO **wvTsfsUploadPathLib**, **tsfsUploadPathClose( )**

tsfsUploadPathLibInit( )

NAME **tsfsUploadPathLibInit( )** – initialize **wvTsfsUploadPathLib** library (Windview)

SYNOPSIS

```
STATUS tsfsUploadPathLibInit (void)
```

DESCRIPTION This routine initializes **wvTsfsUploadPathLib** by pulling in the routines in this file for use with WindView. It is called during system configuration from **usrWindview.c**.

RETURNS **OK**.

SEE ALSO **wvTsfsUploadPathLib**

tsfsUploadPathWrite( )

NAME **tsfsUploadPathWrite( )** – write to the TSFS upload path (Windview)

SYNOPSIS

```
int tsfsUploadPathWrite
    (
    UPLOAD_ID upId,           /* generic upload-path descriptor */
    char *    pStart,         /* address of data to write */
    size_t    size            /* number of bytes of data at pStart */
    )
```

DESCRIPTION This routine writes *size* bytes of data beginning at *pStart* to the upload path connecting the target with the host receiver.

RETURNS The number of bytes written, or **ERROR**.

SEE ALSO **wvTsfsUploadPathLib**, **tsfsUploadPathCreate( )**

T

tt( )

NAME

tt( ) – display a stack trace of a task

SYNOPSIS

```
STATUS tt
    (
    int taskNameOrId          /* task name or task ID */
    )
```

DESCRIPTION

This routine displays a list of the nested routine calls that the specified task is in. Each routine call and its parameters are shown.

If *taskNameOrId* is not specified or zero, the last task referenced is assumed. The **tt( )** routine can only trace the stack of a task other than itself. For instance, when **tt( )** is called from the shell, it cannot trace the shell's stack.

EXAMPLE

```
-> tt "logTask"
 3ab92 _vxTaskEntry   +10 : _logTask (0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
  ee6e _logTask       +12 : _read (5, 3f8a10, 20)
  d460 _read          +10 : _iosRead (5, 3f8a10, 20)
  e234 _iosRead       +9c : _pipeRead (3fce1c, 3f8a10, 20)
 23978 _pipeRead      +24 : _semTake (3f8b78)
value = 0 = 0x0
```

This indicates that **logTask( )** is currently in **semTake( )** (with one parameter) and was called by **pipeRead( )** (with three parameters), which was called by **iosRead( )** (with three parameters), and so on.

WARNING: In order to do the trace, some assumptions are made. In general, the trace will work for all C language routines and for assembly language routines that start with a LINK instruction. Some C compilers require specific flags to generate the LINK first. Most VxWorks assembly language routines include LINK instructions for this reason. The trace facility may produce inaccurate results or fail completely if the routine is written in a language other than C, the routine's entry point is non-standard, or the task's stack is corrupted. Also, all parameters are assumed to be 32-bit quantities, so structures passed as parameters will be displayed as *long* integers.

RETURNS

OK, or **ERROR** if the task does not exist.

SEE ALSO

dbgLib, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

ttyDevCreate( )

NAME **ttyDevCreate( )** – create a VxWorks device for a serial channel

SYNOPSIS

```
STATUS ttyDevCreate
    (
    char *     name,          /* name to use for this device */
    SIO_CHAN * pSioChan,      /* pointer to core driver structure */
    int        rdBufSize,     /* read buffer size, in bytes */
    int        wrtBufSize     /* write buffer size, in bytes */
    )
```

DESCRIPTION This routine creates a device on a specified serial channel. Each channel to be used should have exactly one device associated with it by calling this routine.

For instance, to create the device **/tyCo/0**, with buffer sizes of 512 bytes, the proper call would be:

```
    ttyDevCreate ("/tyCo/0", pSioChan, 512, 512);
```

Where *pSioChan* is the address of the underlying **SIO_CHAN** serial channel descriptor (defined in **sioLib.h**). This routine is typically called by **usrRoot( )** in **usrConfig.c**

RETURNS **OK**, or **ERROR** if the driver is not installed, or the device already exists.

SEE ALSO **ttyDrv**

ttyDrv( )

NAME **ttyDrv( )** – initialize the tty driver

SYNOPSIS

```
STATUS ttyDrv (void)
```

DESCRIPTION This routine initializes the tty driver, which is the OS interface to core serial channel(s). Normally, it is called by **usrRoot( )** in **usrConfig.c**.

After this routine is called, **ttyDevCreate( )** is typically called to bind serial channels to VxWorks devices.

RETURNS **OK**, or **ERROR** if the driver cannot be installed.

SEE ALSO **ttyDrv**

tyAbortFuncSet( )

NAME

tyAbortFuncSet( ) – set the abort function

SYNOPSIS

```
void tyAbortFuncSet
    (
    FUNCPTR func                /* routine to call when abort char received */
    )
```

DESCRIPTION

This routine sets the function that will be called when the abort character is received on a tty. There is only one global abort function, used for any tty on which **OPT_ABORT** is enabled. When the abort character is received from a tty with **OPT_ABORT** set, the function specified in *func* will be called, with no parameters, from interrupt level.

Setting an abort function of **NULL** will disable the abort function.

RETURNS

N/A

SEE ALSO

tyLib, **tyAbortSet( )**

tyAbortSet( )

NAME

tyAbortSet( ) – change the abort character

SYNOPSIS

```
void tyAbortSet
    (
    char ch                     /* char to be abort */
    )
```

DESCRIPTION

This routine sets the abort character to *ch*. The default abort character is CTRL-C.

Typing the abort character to any device whose **OPT_ABORT** option is set will cause the shell task to be killed and restarted. Note that the character set by this routine applies to all devices whose handlers use the standard tty package **tyLib**.

RETURNS

N/A

SEE ALSO

tyLib, **tyAbortFuncSet( )**

tyBackspaceSet( )

NAME
tyBackspaceSet( ) – change the backspace character

SYNOPSIS

```
void tyBackspaceSet
    (
    char ch                  /* char to be backspace */
    )
```

DESCRIPTION
This routine sets the backspace character to *ch*. The default backspace character is CTRL-H.

Typing the backspace character to any device operating in line protocol mode (**OPT_LINE** set) will cause the previous character typed to be deleted, up to the beginning of the current line. Note that the character set by this routine applies to all devices whose handlers use the standard tty package **tyLib**.

RETURNS
N/A

SEE ALSO
tyLib

tyDeleteLineSet( )

NAME
tyDeleteLineSet( ) – change the line-delete character

SYNOPSIS

```
void tyDeleteLineSet
    (
    char ch                  /* char to be line-delete */
    )
```

DESCRIPTION
This routine sets the line-delete character to *ch*. The default line-delete character is CTRL-U.

Typing the delete character to any device operating in line protocol mode (**OPT_LINE** set) will cause all characters in the current line to be deleted. Note that the character set by this routine applies to all devices whose handlers use the standard tty package **tyLib**.

RETURNS
N/A

SEE ALSO
tyLib

T

tyDevInit( )

NAME

tyDevInit( ) – initialize the tty device descriptor

SYNOPSIS

```
STATUS tyDevInit
    (
    TY_DEV_ID pTyDev,         /* ptr to tty dev descriptor to init */
    int       rdBufSize,      /* size of read buffer in bytes */
    int       wrtBufSize,     /* size of write buffer in bytes */
    FUNCPTR   txStartup       /* device transmit start-up routine */
    )
```

DESCRIPTION

This routine initializes a tty device descriptor according to the specified parameters. The initialization includes allocating read and write buffers of the specified sizes from the memory pool, and initializing their respective buffer descriptors. The semaphores are initialized and the write semaphore is given to enable writers. Also, the transmitter start-up routine pointer is set to the specified routine. All other fields in the descriptor are zeroed.

This routine should be called only by serial drivers.

RETURNS

OK, or **ERROR** if there is not enough memory to allocate data structures.

SEE ALSO

tyLib

tyDevRemove( )

NAME

tyDevRemove( ) – remove the tty device descriptor

SYNOPSIS

```
STATUS tyDevRemove
    (
    TY_DEV_ID pTyDev          /* ptr to tty dev descriptor to remove */
    )
```

DESCRIPTION

This routine removes an existing tty device descriptor. It releases the read and write buffers and the descriptor data structure.

RETURNS

OK, or **ERROR** if expected data structures are not found

SEE ALSO

tyLib

tyEOFSet( )

NAME **tyEOFSet( )** – change the end-of-file character

SYNOPSIS

```
void tyEOFSet
    (
    char ch                    /* char to be EOF */
    )
```

DESCRIPTION This routine sets the EOF character to *ch*. The default EOF character is CTRL-D.

Typing the EOF character to any device operating in line protocol mode (**OPT_LINE** set) will cause no character to be entered in the current line, but will cause the current line to be terminated (thus without a newline character). The line is made available to reading tasks. Thus, if the EOF character is the first character input on a line, a line length of zero characters is returned to the reader. This is the standard end-of-file indication on a read call. Note that the EOF character set by this routine will apply to all devices whose handlers use the standard tty package **tyLib**.

RETURNS N/A

SEE ALSO **tyLib**

tyIoctl( )

NAME **tyIoctl( )** – handle device control requests

SYNOPSIS

```
STATUS tyIoctl
    (
    TY_DEV_ID pTyDev,          /* ptr to device to control */
    int       request,         /* request code */
    int       arg              /* some argument */
    )
```

DESCRIPTION This routine handles **ioctl( )** requests for tty devices. The I/O control functions for tty devices are described in the manual entry for **tyLib**.

BUGS In line protocol mode (**OPT_LINE** option set), the **FIONREAD** function actually returns the number of characters available plus the number of lines in the buffer. Thus, if five lines consisting of just NEWLINEs were in the input buffer, the **FIONREAD** function would return the value ten (five characters + five lines).

RETURNS OK or ERROR.

SEE ALSO tyLib

tyIRd( )

NAME **tyIRd( )** – interrupt-level input

SYNOPSIS

```
STATUS tyIRd
    (
    TY_DEV_ID pTyDev,         /* ptr to tty device descriptor */
    char      inchar          /* character read */
    )
```

DESCRIPTION This routine handles interrupt-level character input for tty devices. A device driver calls this routine when it has received a character. This routine adds the character to the ring buffer for the specified device, and gives a semaphore if a task is waiting for it.

This routine also handles all the special characters, as specified in the option word for the device, such as X-on, X-off, NEWLINE, or backspace.

RETURNS **OK**, or **ERROR** if the ring buffer is full.

SEE ALSO **tyLib**

tyITx( )

NAME **tyITx( )** – interrupt-level output

SYNOPSIS

```
STATUS tyITx
    (
    TY_DEV_ID pTyDev,         /* pointer to tty device descriptor */
    char *    pChar           /* where to put character to be output */
    )
```

DESCRIPTION This routine gets a single character to be output to a device. It looks at the ring buffer for *pTyDev* and gives the caller the next available character, if there is one. The character to be output is copied to *pChar*.

RETURNS **OK** if there are more characters to send, or **ERROR** if there are no more characters.

SEE ALSO **tyLib**

tyMonitorTrapSet( )

NAME **tyMonitorTrapSet( )** – change the trap-to-monitor character

SYNOPSIS

```
void tyMonitorTrapSet
    (
    char ch                    /* char to be monitor trap */
    )
```

DESCRIPTION This routine sets the trap-to-monitor character to *ch*. The default trap-to-monitor character is CTRL-X.

Typing the trap-to-monitor character to any device whose **OPT_MON_TRAP** option is set will cause the resident ROM monitor to be entered, if one is present. Once the ROM monitor is entered, the normal multitasking system is halted.

Note that the trap-to-monitor character set by this routine will apply to all devices whose handlers use the standard tty package **tyLib**. Also note that not all systems have a monitor trap available.

RETURNS N/A

SEE ALSO **tyLib**

T

tyRead( )

NAME **tyRead( )** – do a task-level read for a tty device

SYNOPSIS

```
int tyRead
    (
    TY_DEV_ID pTyDev,          /* device to read */
    char *    buffer,          /* buffer to read into */
    int       maxbytes         /* maximum length of read */
    )
```

DESCRIPTION This routine handles the task-level portion of the tty handler's read function. It reads into the buffer up to *maxbytes* available bytes.

This routine should only be called from serial device drivers.

RETURNS The number of bytes actually read into the buffer.

SEE ALSO **tyLib**

tyWrite( )

NAME **tyWrite( )** – do a task-level write for a tty device

SYNOPSIS

```
int tyWrite
    (
    TY_DEV_ID pTyDev,         /* ptr to device structure */
    char *    buffer,         /* buffer of data to write */
    int       nbytes          /* number of bytes in buffer */
    )
```

DESCRIPTION This routine handles the task-level portion of the tty handler's write function.

RETURNS The number of bytes actually written to the device.

SEE ALSO **tyLib**

udpShowInit( )

NAME **udpShowInit( )** – initialize UDP show routines

SYNOPSIS

```
void udpShowInit (void)
```

DESCRIPTION This routine links the UDP show facility into the VxWorks system. These routines are included automatically if **INCLUDE_NET_SHOW** and **INCLUDE_UDP** are defined.

RETURNS N/A

SEE ALSO **udpShow**

udpstatShow( )

NAME **udpstatShow( )** – display statistics for the UDP protocol

SYNOPSIS

```
void udpstatShow (void)
```

DESCRIPTION This routine displays statistics for the UDP protocol.

RETURNS N/A

SEE ALSO **udpShow**

ungetc( )

NAME **ungetc( )** – push a character back into an input stream (ANSI)

SYNOPSIS

```
int ungetc
    (
    int    c,                  /* character to push */
    FILE * fp                  /* input stream */
    )
```

DESCRIPTION This routine pushes a character *c* (converted to an **unsigned char**) back into the specified input stream. The pushed-back characters will be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call on the stream to a file positioning function (**fseek( )**, **fsetpos( )**, or **rewind( )**) discards any pushed-back characters for the stream. The external storage corresponding to the stream is unchanged.

One character of push-back is guaranteed. If **ungetc( )** is called too many times on the same stream without an intervening read or file positioning operation, the operation may fail.

If the value of *c* equals EOF, the operation fails and the input stream is unchanged.

A successful call to **ungetc( )** clears the end-of-file indicator for the stream. The value of the file position indicator for the stream after reading or discarding all pushed-back characters is the same as it was before the character were pushed back. For a text stream, the value of its file position indicator after a successful call to **ungetc( )** is unspecified until all pushed-back characters are read or discarded. For a binary stream, the file position indicator is decremented by each successful call to **ungetc( )**; if its value was zero before a call, it is indeterminate after the call.

INCLUDE **stdio.h**

RETURNS The pushed-back character after conversion, or EOF if the operation fails.

SEE ALSO **ansiStdio**, **getc( )**, **fgetc( )**

unixDiskDevCreate( )

NAME **unixDiskDevCreate( )** – create a UNIX disk device

SYNOPSIS

```
BLK_DEV *unixDiskDevCreate
    (
    char * unixFile,          /* name of the UNIX file */
    int    bytesPerBlk,       /* number of bytes per block */
    int    blksPerTrack,      /* number of blocks per track */
    int    nBlocks            /* number of blocks on this device */
    )
```

DESCRIPTION This routine creates a UNIX disk device.

The *unixFile* parameter specifies the name of the UNIX file to use for the disk device.

The *bytesPerBlk* parameter specifies the size of each logical block on the disk. If *bytesPerBlk* is zero, 512 is the default.

The *blksPerTrack* parameter specifies the number of blocks on each logical track of the disk. If *blksPerTrack* is zero, the count of blocks per track is set to *nBlocks* (*i.e.*, the disk is defined as having only one track).

The *nBlocks* parameter specifies the size of the disk, in blocks. If *nBlocks* is zero, a default size is used. The default is calculated as the size of the UNIX disk divided by the number of bytes per block.

This routine is only applicable to VxSim for Solaris and VxSim for HP.

RETURNS A pointer to block device (**BLK_DEV**) structure, or **NULL**, if unable to open the UNIX disk.

SEE ALSO **unixDrv**

unixDiskInit( )

NAME **unixDiskInit( )** – initialize a dosFs disk on top of UNIX

SYNOPSIS

```
void unixDiskInit
    (
    char * unixFile,          /* UNIX file name */
    char * volName,           /* dosFs name */
    int    diskSize           /* number of bytes */
    )
```

DESCRIPTION This routine provides some convenience for a user wanting to create a UNIX disk-based dosFs file system under VxWorks. The user only specifies the UNIX file to use, the dosFs volume name, and the size of the volume in bytes, if the UNIX file needs to be created.

This routine is only applicable to VxSim for Solaris and VxSim for HP.

RETURNS N/A

SEE ALSO **unixDrv**

U

unixDrv( )

NAME

unixDrv( ) – install UNIX disk driver

SYNOPSIS

```
STATUS unixDrv (void)
```

DESCRIPTION

Used in **usrConfig.c** to cause the UNIX disk driver to be linked in when building VxWorks. Otherwise, it is not necessary to call this routine before using the UNIX disk driver.

This routine is only applicable to VxSim for Solaris and VxSim for HP.

RETURNS

OK (always).

SEE ALSO

unixDrv

unld( )

NAME

unld( ) – unload an object module by specifying a file name or module ID

SYNOPSIS

```
STATUS unld
    (
    void * nameOrId,          /* name or ID of the object module file */
    int    options
    )
```

DESCRIPTION

This routine unloads the specified object module from the system. The module can be specified by name or by module ID. For a.out and ECOFF format modules, unloading does the following:

(1) It frees the space allocated for text, data, and BSS segments, unless **loadModuleAt( )** was called with specific addresses, in which case the user is responsible for freeing the space.
(2) It removes all symbols associated with the object module from the system symbol table.
(3) It removes the module descriptor from the module list.

For other modules of other formats, unloading has similar effects.

Before any modules are unloaded, all breakpoints in the system are deleted. If you need to keep breakpoints, set the options parameter to **UNLD_KEEP_BREAKPOINTS**. No breakpoints can be set in code that is unloaded.

This routine is a **shell command**. That is, it is designed to be used only in the shell, and not in code running on the target. In future releases, calling **unld( )** directly from code may not be supported.

RETURNS **OK** or **ERROR**.

SEE ALSO **unldLib**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

unldByGroup( )

NAME **unldByGroup( )** – unload an object module by specifying a group number

SYNOPSIS

```
STATUS unldByGroup
    (
    UINT16 group,             /* group number to unload */
    int    options            /* options, currently unused */
    )
```

DESCRIPTION This routine unloads an object module that has a group number matching *group*.

See the manual entries for **unld( )** or **unldLib** for more information on module unloading.

RETURNS **OK** or **ERROR**.

SEE ALSO **unldLib**, **unld( )**

unldByModuleId( )

NAME **unldByModuleId( )** – unload an object module by specifying a module ID

SYNOPSIS

```
STATUS unldByModuleId
    (
    MODULE_ID moduleId,       /* module ID to unload */
    int       options
    )
```

DESCRIPTION This routine unloads an object module that has a module ID matching *moduleId*.

See the manual entries for **unld( )** or **unldLib** for more information on module unloading.

RETURNS **OK** or **ERROR**.

SEE ALSO **unldLib**, **unld( )**

unldByNameAndPath( )

NAME

unldByNameAndPath( ) – unload an object module by specifying a name and path

SYNOPSIS

```
STATUS unldByNameAndPath
    (
    char * name,            /* name of the object module to unload */
    char * path,            /* path to the object module to unload */
    int    options          /* options, currently unused */
    )
```

DESCRIPTION

This routine unloads an object module specified by *name* and *path*.

See the manual entries for **unld( )** or **unldLib** for more information on module unloading.

RETURNS

OK or **ERROR**.

SEE ALSO

unldLib, **unld( )**

unlink( )

NAME

unlink( ) – delete a file (POSIX)

SYNOPSIS

```
STATUS unlink
    (
    char * name             /* name of the file to remove */
    )
```

DESCRIPTION

This routine deletes a specified file. It performs the same function as **remove( )** and is provided for POSIX compatibility.

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**.

SEE ALSO

ioLib, **remove( )**

usrAtaConfig( )

NAME

usrAtaConfig( ) – mount a DOS file system from an ATA hard disk or a CDROM

SYNOPSIS

```
STATUS usrAtaConfig
    (
    int    ctrl,              /* 0: primary address, 1: secondary address */
    int    drive,             /* drive number of hard disk (0 or 1) */
    char * devNames           /* mount points for each partition */
    )
```

DESCRIPTION

file system from an ATAPI CDROM drive

This routine mounts a DOS file system from an ATA hard disk. Parameters:

drive
: the drive number of the hard disk; 0 is **C:** and 1 is **D:**.

devName
: the mount point for all partitions which are expected to be present on the disk, separated with commas, for example "/ata0,/ata1" or "C:,D:". Blanks are not allowed in this string. If the drive is an ATAPI CDROM drive, then the CDROM file system is specified by appending "(cdrom)" after the mount point name. For example, a CDROM drive could be specified as "/cd(cdrom)".

NOTE: Because VxWorks does not support creation of partition tables, hard disks formatted and initialized on VxWorks are not compatible with DOS machines. This routine does not refuse to mount a hard disk that was initialized on VxWorks. Up to 8 disk partitions are supported.

RETURNS

OK or **ERROR**.

SEE ALSO

usrAta, **src/config/usrAta.c**, *VxWorks Programmer's Guide: I/O System, Local File Systems, Intel i386/i486/Pentium*

usrAtaInit( )

NAME **usrAtaInit( )** – initialize the hard disk driver

SYNOPSIS

```
void usrAtaInit (void)
```

DESCRIPTION This routine is called from **usrConfig.c** to initialize the hard drive.

SEE ALSO **usrAta**

usrClock( )

NAME **usrClock( )** – user-defined system clock interrupt routine

SYNOPSIS

```
void usrClock ()
```

DESCRIPTION This routine is called at interrupt level on each clock interrupt. It is installed by **usrRoot( )** with a **sysClkConnect( )** call. It calls all the other packages that need to know about clock ticks, including the kernel itself.

If the application needs anything to happen at the system clock interrupt level, it can be added to this routine.

RETURNS N/A

SEE ALSO **usrConfig**

usrFdConfig( )

NAME **usrFdConfig( )** – mount a DOS file system from a floppy disk

SYNOPSIS

```
STATUS usrFdConfig
    (
    int    drive,             /* drive number of floppy disk (0 - 3) */
    int    type,              /* type of floppy disk */
    char * fileName           /* mount point */
    )
```

DESCRIPTION

This routine mounts a DOS file system from a floppy disk device.

The *drive* parameter is the drive number of the floppy disk; valid values are 0 to 3.

The *type* parameter specifies the type of diskette, which is described in the structure table **fdTypes[]** in **sysLib.c**. *type* is an index to the table. Currently the table contains two diskette types:

– A *type* of 0 indicates the first entry in the table (3.5" 2HD, 1.44MB);

– A *type* of 1 indicates the second entry in the table (5.25" 2HD, 1.2MB).

The *fileName* parameter is the mount point, *e.g.*, **/fd0/**.

RETURNS

OK or **ERROR**.

SEE ALSO

usrFd, *VxWorks Programmer's Guide: I/O System, Local File Systems, Intel i386/i486 Appendix*

---

usrFdiskPartCreate( )

NAME

usrFdiskPartCreate( ) – create an FDISK-like partition table on a disk

SYNOPSIS

```
STATUS usrFdiskPartCreate
    (
    CBIO_DEV_ID cDev,     /* device representing the entire disk */
    int         nPart,    /* how many partitions needed, default=1, max=4 */
    int         size1,    /* space percentage for second partition */
    int         size2,    /* space percentage for third partition */
    int         size3     /* space percentage for fourth partition */
    )
```

DESCRIPTION

This function may be used to create a basic PC partition table. Such partition table however is not intended to be compatible with other operating systems, it is intended for disks connected to a VxWorks target, but without the access to a PC which may be used to create the partition table.

This function is capable of creating only one partition table - the MBR, and will not create any Bootable or Extended partitions. Therefore, 4 partitions are supported.

dev is a CBIO device handle for an entire disk, *e.g.*, a handle returned by **dcacheDevCreate( )**, or if dpartCbio is used, it can be either the Master partition manager handle, or the one of the 0th partition if the disk does not contain a partition table at all.

The *nPart* argument contains the number of partitions to create. If *nPart* is 0 or 1, then a single partition covering the entire disk is created. If *nPart* is between 2 and 4, then the arguments *size1*, *size2*and *size3* contain the *percentage* of disk space to be assigned to the

2nd, 3rd, and 4th partitions respectively. The first partition (partition 0) will be assigned the remainder of space left (space hog).

Partition sizes will be round down to be multiple of whole tracks so that partition Cylinder/Head/Track fields will be initialized as well as the LBA fields. Although the CHS fields are written they are not used in VxWorks, and can not be guaranteed to work correctly on other systems.

RETURNS
OK or **ERROR** writing a partition table to disk

SEE ALSO
usrFdiskPartLib

usrFdiskPartRead( )

NAME
usrFdiskPartRead( ) – read an FDISK-style partition table

SYNOPSIS

```
STATUS usrFdiskPartRead
    (
    CBIO_DEV_ID        cDev,     /* device from which to read blocks */
    PART_TABLE_ENTRY * pPartTab, /* table where to fill results */
    int                nPart     /* # of entries in pPartTable */
    )
```

DESCRIPTION
This function will read and decode a PC formatted partition table on a disk, and fill the appropriate partition table array with the resulting geometry, which should be used by the dpartCbio partition manager to access a partitioned disk with a shared disk cache.

EXAMPLE
The following example shows how a hard disk which is expected to have up to two partitions might be configured, assuming the physical level initialization resulted in the *blkIoDevId* handle:

```
devCbio = dcacheDevCreate( blkIoDevId, 0, 0x20000, "Hard Disk");
mainDevId = dpartDevCreate( devCbio, 2, usrFdiskPartRead )
dosFsDevCreate(  "/disk0a", dpartPartGet (mainDevId, 0), 0,0,0);
dosFsDevCreate(  "/disk0b", dpartPartGet (mainDevId, 1), 0,0,0);
```

RETURNS
OK or **ERROR** if partition table is corrupt

SEE ALSO
usrFdiskPartLib

usrFdiskPartShow( )

NAME

usrFdiskPartShow( ) – parse and display partition data

SYNOPSIS

```
STATUS usrFdiskPartShow
    (
    CBIO_DEV_ID cbio,           /* device CBIO handle */
    block_t     extPartOffset, /* user should pass zero */
    block_t     currentOffset, /* user should pass zero */
    int         extPartLevel   /* user should pass zero */
    )
```

DESCRIPTION

This routine is intended to be user callable.

A device dependent partition table show routine. This routine outputs formatted data for all partition table fields for every partition table found on a given disk, starting with the MBR sectors partition table. This code can be removed to reduce code size by undefining: **INCLUDE_PART_SHOW** and rebuilding this library and linking to the new library.

This routine takes three arguments. First, a CBIO pointer (assigned for the entire physical disk) usually obtained from **dcacheDevCreate( )**. It also takes two **block_t** type arguments and one signed int, the user shall pass zero in these parameters.

For example:

```
sp usrFdiskPartShow (pCbio,0,0,0)
```

Developers may use size*arch* to view code size.

RETURNS

OK or **ERROR**

SEE ALSO

usrFdiskPartLib

U

usrIdeConfig( )

NAME

usrIdeConfig( ) – mount a DOS file system from an IDE hard disk

SYNOPSIS

```
STATUS usrIdeConfig
    (
    int    drive,              /* drive number of hard disk (0 or 1) */
    char * fileName            /* mount point */
    )
```

DESCRIPTION

This routine mounts a DOS file system from an IDE hard disk.

The *drive* parameter is the drive number of the hard disk; 0 is **C:** and 1 is **D:**.

The *fileName* parameter is the mount point, *e.g.*, **/ide0/**.

NOTE: Because VxWorks does not support partitioning, hard disks formatted and initialized on VxWorks are not compatible with DOS machines. This routine does not refuse to mount a hard disk that was initialized on VxWorks. The hard disk is assumed to have only one partition with a partition record in sector 0.

RETURNS

OK or **ERROR**.

SEE ALSO

usrIde, *VxWorks Programmer's Guide: I/O System, Local File Systems, Intel i386/i486 Appendix*

usrInit( )

NAME

usrInit( ) – user-defined system initialization routine

SYNOPSIS

```
void usrInit
    (
    int startType
    )
```

DESCRIPTION

This is the first C code executed after the system boots. This routine is called by the assembly language start-up routine **sysInit( )** which is in the **sysALib** module of the target-specific directory. It is called with interrupts locked out. The kernel is not multitasking at this point.

This routine starts by clearing BSS; thus all variables are initialized to 0, as per the C specification. It then initializes the hardware by calling **sysHwInit( )**, sets up the interrupt/exception vectors, and starts kernel multitasking with **usrRoot( )** as the root task.

RETURNS N/A

SEE ALSO **usrConfig**, **kernelLib**

usrRoot( )

NAME **usrRoot( )** – the root task

SYNOPSIS

```
void usrRoot
    (
    char *   pMemPoolStart,   /* start of system memory partition */
    unsigned memPoolSize      /* initial size of mem pool */
    )
```

DESCRIPTION This is the first task to run under the multitasking kernel. It performs all final initialization and then starts other tasks.

It initializes the I/O system, installs drivers, creates devices, and sets up the network, etc., as necessary for a particular configuration. It may also create and load the system symbol table, if one is to be included. It may then load and spawn additional tasks as needed. In the default configuration, it simply initializes the VxWorks shell.

RETURNS N/A

SEE ALSO **usrConfig**

usrScsiConfig( )

NAME **usrScsiConfig( )** – configure SCSI peripherals

SYNOPSIS

```
STATUS usrScsiConfig (void)
```

DESCRIPTION This code configures the SCSI disks and other peripherals on a SCSI controller chain.

The macro **SCSI_AUTO_CONFIG** will include code to scan all possible device/lun id's and to configure a scsiPhysDev structure for each device found. Of course this doesn't include final configuration for disk partitions, floppy configuration parameters, or tape system setup. All of these actions must be performed by user code, either through **sysScsiConfig( )**, the startup script, or by the application program.

The user may customize this code on a per BSP basis using the **SYS_SCSI_CONFIG** macro. If defined, then this routine will call the routine **sysScsiConfig( )**. That routine is to be provided by the BSP, either in **sysLib.c** or **sysScsi.c**. If **SYS_SCSI_CONFIG** is not defined, then **sysScsiConfig( )** will not be called as part of this routine.

An example **sysScsiConfig( )** routine can be found in **target/src/config/usrScsi.c**. The example code contains sample configurations for a hard disk, a floppy disk and a tape unit.

RETURNS
OK or **ERROR**.

SEE ALSO
usrScsi, *VxWorks Programmer's Guide: I/O System, Local File Systems*

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.

It is an error for *nbytes* to be odd.

RETURNS
N/A

SEE ALSO
bLib, **swab( )**

utime( )

NAME

utime( ) – update time on a file

SYNOPSIS

```
int utime
    (
    char *            file,
    struct utimbuf * newTimes
    )
```

DESCRIPTION

RETURNS

OK or ERROR.

SEE ALSO

dirLib, **stat( )**, **fstat( )**, **ls( )**

va_arg( )

NAME

va_arg( ) – expand to an expression having the type and value of the call's next argument

SYNOPSIS

```
void va_arg
    (
     ap,                        /* list of type va_list */
     type                       /* type */
    )
```

DESCRIPTION

Each invocation of this macro modifies an object of type **va_list** (*ap*) so that the values of successive arguments are returned in turn. The parameter *type* is a type name specified such that the type of a pointer to an object that has the specified type can be obtained simply by postfixing a * to *type*. If there is no actual next argument, or if *type* is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), the behavior is undefined.

RETURNS

The first invocation of **va_arg( )** after **va_start( )** returns the value of the argument after that specified by *parmN* (the rightmost parameter). Successive invocations return the value of the remaining arguments in succession.

SEE ALSO

ansiStdarg

va_end( )

NAME

va_end( ) – facilitate a normal return from a routine using a **va_list** object

SYNOPSIS

```
void va_end
    (
     ap                         /* list of type va_list */
    )
```

DESCRIPTION

This macro facilitates a normal return from the function whose variable argument list was referred to by the expansion of **va_start( )** that initialized the **va_list** object.

va_end( ) may modify the **va_list** object so that it is no longer usable (without an intervening invocation of **va_start( )**). If there is no corresponding invocation of the **va_start( )** macro, or if the **va_end( )** macro is not invoked before the return, the behavior is undefined.

RETURNS

N/A

SEE ALSO

ansiStdarg

va_start( )

NAME **va_start( )** – initialize a **va_list** object for use by **va_arg( )** and **va_end( )**

SYNOPSIS

```
void va_start
    (
     ap,                     /* list of type va_list */
     parmN                   /* rightmost parameter */
    )
```

DESCRIPTION This macro initializes an object of type **va_list** (*ap*) for subsequent use by **va_arg( )** and **va_end( )**. The parameter *parmN* is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the, ...). If *parmN* is declared with the register storage class with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions, the behavior is undefined.

RETURNS N/A

SEE ALSO **ansiStdarg**

valloc( )

NAME **valloc( )** – allocate memory on a page boundary

SYNOPSIS

```
void * valloc
    (
    unsigned size            /* number of bytes to allocate */
    )
```

DESCRIPTION This routine allocates a buffer of *size* bytes from the system memory 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 **S_memLib_PAGE_SIZE_UNAVAILABLE**

SEE ALSO **memLib**

version( )

NAME
version( ) – print VxWorks version information

SYNOPSIS

```
void version (void)
```

DESCRIPTION
This command prints the VxWorks version number, the date this copy of VxWorks was made, and other pertinent information.

EXAMPLE

```
-> version
VxWorks (for Mizar 7170) version 5.1
Kernel: WIND version 2.1.
Made on Tue Jul 27 20:26:23 CDT 1997.
Boot line:
enp(0,0)host:/usr/wpwr/target/config/mz7170/vxWorks e=90.0.0.50 h=90.0.0.4
u=target
value = 1 = 0x1
```

RETURNS
N/A

SEE ALSO
usrLib, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

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.

SEE ALSO
fioLib, **fdprintf( )**

vfprintf( )

NAME

vfprintf( ) – write a formatted string to a stream (ANSI)

SYNOPSIS

```
int vfprintf
    (
    FILE *       fp,           /* stream to write to */
    const char * fmt,          /* format string */
    va_list      vaList        /* arguments to format string */
    )
```

DESCRIPTION

This routine is equivalent to **fprintf( )**, except that it takes the variable arguments to be formatted from a list *vaList* of type **va_list** rather than from in-line arguments.

INCLUDE FILES

stdio.h

RETURNS

The number of characters written, or a negative value if an output error occurs.

SEE ALSO

ansiStdio, **fprintf( )**

vmBaseGlobalMapInit( )

NAME

vmBaseGlobalMapInit( ) – initialize global mapping

SYNOPSIS

```
VM_CONTEXT_ID vmBaseGlobalMapInit
    (
    PHYS_MEM_DESC * pMemDescArray,      /* pointer to array of mem descs */
    int           numDescArrayElements, /* no. of elements in pMemDescArray */
    BOOL          enable                /* enable virtual memory */
    )
```

DESCRIPTION

This routine creates and installs a virtual memory context with mappings defined for each contiguous memory segment defined in *pMemDescArray*. In the standard VxWorks configuration, an instance of **PHYS_MEM_DESC** (called **sysPhysMemDesc**) is defined in **sysLib.c**; the variable is passed to **vmBaseGlobalMapInit( )** by the system configuration mechanism.

The physical memory descriptor also contains state information used to initialize the state information in the MMU's translation table for that memory segment. The following state bits may be or'ed together:

V

VM_STATE_VALID	**VM_STATE_VALID_NOT**	valid/invalid
VM_STATE_WRITABLE	**VM_STATE_WRITABLE_NOT**	writable/write-protected
VM_STATE_CACHEABLE	**VM_STATE_CACHEABLE_NOT**	cacheable/not-cacheable

Additionally, mask bits are or'ed together in the **initialStateMask** structure element to describe which state bits are being specified in the **initialState** structure element:

VM_STATE_MASK_VALID
VM_STATE_MASK_WRITABLE
VM_STATE_MASK_CACHEABLE

If *enable* is **TRUE**, the MMU is enabled upon return.

RETURNS A pointer to a newly created virtual memory context, or **NULL** if memory cannot be mapped.

SEE ALSO **vmBaseLib**, **vmBaseLibInit( )**

vmBaseLibInit( )

NAME **vmBaseLibInit( )** – initialize base virtual memory support

SYNOPSIS

```
STATUS vmBaseLibInit
    (
    int pageSize              /* size of page */
    )
```

DESCRIPTION This routine initializes the virtual memory context class and module-specific data structures. It is called only once during system initialization, and should be followed with a call to **vmBaseGlobalMapInit( )**, which initializes and enables the MMU.

RETURNS **OK**.

SEE ALSO **vmBaseLib**, **vmBaseGlobalMapInit( )**

vmBasePageSizeGet( )

NAME **vmBasePageSizeGet( )** – return the page size

SYNOPSIS

```
int vmBasePageSizeGet (void)
```

DESCRIPTION This routine returns the architecture-dependent page size.

This routine is callable from interrupt level.

RETURNS The page size of the current architecture.

SEE ALSO **vmBaseLib**

vmBaseStateSet( )

NAME **vmBaseStateSet( )** – change the state of a block of virtual memory

SYNOPSIS

```
STATUS vmBaseStateSet
    (
    VM_CONTEXT_ID context,     /* context - NULL == currentContext */
    void *        pVirtual,    /* virtual address to modify state of */
    int           len,         /* len of virtual space to modify state of */
    UINT          stateMask,   /* state mask */
    UINT          state        /* state */
    )
```

DESCRIPTION This routine changes the state of a block of virtual memory. Each page of virtual memory has at least three elements of state information: validity, writability, and cacheability. Specific architectures may define additional state information; see **vmLib.h** for additional architecture-specific states. Memory accesses to a page marked as invalid will result in an exception. Pages may be invalidated to prevent them from being corrupted by invalid references. Pages may be defined as read-only or writable, depending on the state of the writable bits. Memory accesses to pages marked as not-cacheable will always result in a memory cycle, bypassing the cache. This is useful for multiprocessing, multiple bus masters, and hardware control registers.

The following states are provided and may be or'ed together in the state parameter:

VM_STATE_VALID	**VM_STATE_VALID_NOT**	valid/invalid
VM_STATE_WRITABLE	**VM_STATE_WRITABLE_NOT**	writable/write-protected
VM_STATE_CACHEABLE	**VM_STATE_CACHEABLE_NOT**	cacheable/not-cacheable

Additionally, the following masks are provided so that only specific states may be set. These may be or'ed together in the **stateMask** parameter.

VM_STATE_MASK_VALID
VM_STATE_MASK_WRITABLE
VM_STATE_MASK_CACHEABLE

If *context* is specified as **NULL**, the current context is used.

This routine is callable from interrupt level.

RETURNS **OK**, or **ERROR** if the validation fails, *pVirtual* is not on a page boundary, *len* is not a multiple of the page size, or the architecture-dependent state set fails for the specified virtual address.

ERRNO **S_vmLib_NOT_PAGE_ALIGNED**, **S_vmLib_BAD_STATE_PARAM**, **S_vmLib_BAD_MASK_PARAM**

SEE ALSO **vmBaseLib**

---

vmContextCreate( )

NAME **vmContextCreate( )** – create a new virtual memory context (VxVMI Opt.)

SYNOPSIS

```
VM_CONTEXT_ID vmContextCreate (void)
```

DESCRIPTION This routine creates a new virtual memory context. The newly created context does not become the current context until explicitly installed by a call to **vmCurrentSet( )**. Modifications to the context state (mappings, state changes, etc.) may be performed on any virtual memory context, even if it is not the current context.

This routine should not be called from interrupt level.

AVAILABILITY This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS A pointer to a new virtual memory context, or **NULL** if the allocation or initialization fails.

SEE ALSO **vmLib**

vmContextDelete( )

NAME **vmContextDelete( )** – delete a virtual memory context (VxVMI Opt.)

SYNOPSIS

```
STATUS vmContextDelete
    (
    VM_CONTEXT_ID context
    )
```

DESCRIPTION This routine deallocates the underlying translation table associated with a virtual memory context. It does not free the physical memory already mapped into the virtual memory space.

This routine should not be called from interrupt level.

AVAILABILITY This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS **OK**, or **ERROR** if *context* is not a valid context descriptor or if an error occurs deleting the translation table.

SEE ALSO **vmLib**

vmContextShow( )

NAME **vmContextShow( )** – display the translation table for a context (VxVMI Opt.)

SYNOPSIS

```
STATUS vmContextShow
    (
    VM_CONTEXT_ID context     /* context - NULL == currentContext */
    )
```

DESCRIPTION This routine displays the translation table for a specified context. If *context* is specified as **NULL**, the current context is displayed. Output is formatted to show blocks of virtual memory with consecutive physical addresses and the same state. State information shows the writable and cacheable states. If the block is in global virtual memory, the word "global" is appended to the line. Only virtual memory that has its valid state bit set is displayed.

This routine should be used for debugging purposes only.

Note that this routine cannot report non-standard architecture-dependent states.

V

AVAILABILITY This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS **OK**, or **ERROR** if the virtual memory context is invalid.

SEE ALSO **vmShow**

vmCurrentGet( )

NAME **vmCurrentGet( )** – get the current virtual memory context (VxVMI Opt.)

SYNOPSIS

```
VM_CONTEXT_ID vmCurrentGet (void)
```

DESCRIPTION This routine returns the current virtual memory context.

This routine is callable from interrupt level.

AVAILABILITY This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS The current virtual memory context, or **NULL** if no virtual memory context is installed.

SEE ALSO **vmLib**

vmCurrentSet( )

NAME **vmCurrentSet( )** – set the current virtual memory context (VxVMI Opt.)

SYNOPSIS

```
STATUS vmCurrentSet
    (
    VM_CONTEXT_ID context     /* context to install */
    )
```

DESCRIPTION This routine installs a specified virtual memory context.

This routine is callable from interrupt level.

AVAILABILITY This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS **OK**, or **ERROR** if the validation or context switch fails.

SEE ALSO **vmLib**

vmEnable( )

NAME **vmEnable( )** – enable or disable virtual memory (VxVMI Opt.)

SYNOPSIS

```
STATUS vmEnable
    (
    BOOL enable               /* TRUE == enable MMU, FALSE == disable MMU */
    )
```

DESCRIPTION This routine turns virtual memory on and off. Memory management should not be turned off once it is turned on except in the case of system shutdown.

This routine is callable from interrupt level.

AVAILABILITY This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS **OK**, or **ERROR** if the validation or architecture-dependent code fails.

SEE ALSO **vmLib**

vmGlobalInfoGet( )

NAME **vmGlobalInfoGet( )** – get global virtual memory information (VxVMI Opt.)

SYNOPSIS

```
UINT8 *vmGlobalInfoGet (void)
```

DESCRIPTION This routine provides a description of those parts of the virtual memory space dedicated to global memory. The routine returns a pointer to an array of UINT8. Each element of the array corresponds to a block of virtual memory, the size of which is architecture-dependent and can be obtained with a call to **vmPageBlockSizeGet( )**. To determine if a particular address is in global virtual memory, use the following code:

```
UINT8 *globalPageBlockArray = vmGlobalInfoGet ();
int pageBlockSize = vmPageBlockSizeGet ();

if (globalPageBlockArray[addr/pageBlockSize])
    ...
```

The array pointed to by the returned pointer is guaranteed to be static as long as no calls are made to **vmGlobalMap( )** while the array is being examined. The information in the array can be used to determine what portions of the virtual memory space are available for use as private virtual memory within a virtual memory context.

This routine is callable from interrupt level.

AVAILABILITY This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS A pointer to an array of UINT8.

SEE ALSO **vmLib**, **vmPageBlockSizeGet( )**

---

vmGlobalMap( )

NAME **vmGlobalMap( )** – map physical pages to virtual space in shared global virtual memory (VxVMI Opt.)

SYNOPSIS

```
STATUS vmGlobalMap
    (
    void * virtualAddr,        /* virtual address */
    void * physicalAddr,       /* physical address */
    UINT   len                 /* len of virtual and physical spaces */
    )
```

DESCRIPTION This routine maps physical pages to virtual space that is shared by all virtual memory contexts. Calls to **vmGlobalMap( )** should be made before any virtual memory contexts are created to insure that the shared global mappings are included in all virtual memory contexts. Mappings created with **vmGlobalMap( )** after virtual memory contexts are created are not guaranteed to appear in all virtual memory contexts. After the call to **vmGlobalMap( )**, the state of all pages in the newly mapped virtual memory is unspecified and must be set with a call to **vmStateSet( )**, once the initial virtual memory context is created.

This routine should not be called from interrupt level.

AVAILABILITY This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS **OK**, or **ERROR** if *virtualAddr* or *physicalAddr* are not on page boundaries, *len* is not a multiple of the page size, or the mapping fails.

ERRNO **S_vmLib_NOT_PAGE_ALIGNED**

SEE ALSO **vmLib**

vmGlobalMapInit( )

NAME **vmGlobalMapInit( )** – initialize global mapping (VxVMI Opt.)

SYNOPSIS

```
VM_CONTEXT_ID vmGlobalMapInit
    (
    PHYS_MEM_DESC * pMemDescArray,     /* pointer to array of mem descs */
    int           numDescArrayElements, /* num of elements in pMemDescArray */
    BOOL          enable               /* enable virtual memory */
    )
```

DESCRIPTION This routine is a convenience routine that creates and installs a virtual memory context with global mappings defined for each contiguous memory segment defined in the physical memory descriptor array passed as an argument. The context ID returned becomes the current virtual memory context.

The physical memory descriptor also contains state information used to initialize the state information in the MMU's translation table for that memory segment. The following state bits may be or'ed together:

VM_STATE_VALID	**VM_STATE_VALID_NOT**	valid/invalid
VM_STATE_WRITABLE	**VM_STATE_WRITABLE_NOT**	writable/write-protected
VM_STATE_CACHEABLE	**VM_STATE_CACHEABLE_NOT**	cacheable/not-cacheable

Additionally, mask bits are or'ed together in the **initialStateMask** structure element to describe which state bits are being specified in the **initialState** structure element:

VM_STATE_MASK_VALID
VM_STATE_MASK_WRITABLE
VM_STATE_MASK_CACHEABLE

If the *enable* parameter is **TRUE**, the MMU is enabled upon return. The **vmGlobalMapInit( )** routine should be called only after **vmLibInit( )** has been called.

AVAILABILITY This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS A pointer to a newly created virtual memory context, or **NULL** if the memory cannot be mapped.

SEE ALSO **vmLib**

vmLibInit( )

NAME
vmLibInit( ) – initialize the virtual memory support module (VxVMI Opt.)

SYNOPSIS

```
STATUS vmLibInit
    (
    int pageSize              /* size of page */
    )
```

DESCRIPTION
This routine initializes the virtual memory context class. It is called only once during system initialization.

AVAILABILITY
This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS
OK.

SEE ALSO
vmLib

vmMap( )

NAME
vmMap( ) – map physical space into virtual space (VxVMI Opt.)

SYNOPSIS

```
STATUS vmMap
    (
    VM_CONTEXT_ID context,      /* context - NULL == currentContext */
    void *        virtualAddr,  /* virtual address */
    void *        physicalAddr, /* physical address */
    UINT          len           /* len of virtual and physical spaces */
    )
```

DESCRIPTION
This routine maps physical pages into a contiguous block of virtual memory. *virtualAddr* and *physicalAddr* must be on page boundaries, and *len* must be evenly divisible by the page size. After the call to **vmMap( )**, the state of all pages in the newly mapped virtual memory is valid, writable, and cacheable.

The **vmMap( )** routine can fail if the specified virtual address space conflicts with the translation tables of the global virtual memory space. The global virtual address space is architecture-dependent and is initialized at boot time with calls to **vmGlobalMap( )** by **vmGlobalMapInit( )**. If a conflict results, **errno** is set to

S_vmLib_ADDR_IN_GLOBAL_SPACE. To avoid this conflict, use **vmGlobalInfoGet( )** to ascertain which portions of the virtual address space are reserved for the global virtual address space. If *context* is specified as **NULL**, the current virtual memory context is used.

This routine should not be called from interrupt level.

AVAILABILITY This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS **OK**, or **ERROR** if *virtualAddr* or *physicalAddr* are not on page boundaries, *len* is not a multiple of the page size, the validation fails, or the mapping fails.

ERRNO **S_vmLib_NOT_PAGE_ALIGNED**, **S_vmLib_ADDR_IN_GLOBAL_SPACE**

SEE ALSO **vmLib**

vmPageBlockSizeGet( )

NAME **vmPageBlockSizeGet( )** – get the architecture-dependent page block size (VxVMI Opt.)

SYNOPSIS

```
int vmPageBlockSizeGet (void)
```

DESCRIPTION This routine returns the size of a page block for the current architecture. Each MMU architecture constructs translation tables such that a minimum number of pages are pre-defined when a new section of the translation table is built. This minimal group of pages is referred to as a "page block." This routine may be used in conjunction with **vmGlobalInfoGet( )** to examine the layout of global virtual memory.

This routine is callable from interrupt level.

AVAILABILITY This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS The page block size of the current architecture.

SEE ALSO **vmLib**, **vmGlobalInfoGet( )**

vmPageSizeGet( )

NAME **vmPageSizeGet( )** – return the page size (VxVMI Opt.)

SYNOPSIS

```
int vmPageSizeGet (void)
```

DESCRIPTION This routine returns the architecture-dependent page size.

This routine is callable from interrupt level.

AVAILABILITY This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS The page size of the current architecture.

SEE ALSO **vmLib**

vmShowInit( )

NAME **vmShowInit( )** – include virtual memory show facility (VxVMI Opt.)

SYNOPSIS

```
void vmShowInit (void)
```

DESCRIPTION This routine acts as a hook to include **vmContextShow( )**. It is called automatically when the virtual memory show facility is configured into VxWorks using either of the following methods:

– If you use the configuration header files, define both **INCLUDE_MMU_FULL**

and **INCLUDE_SHOW_ROUTINES** in **config.h**.

– If you use the Tornado project facility, select **INCLUDE_MMU_FULL_SHOW**.

AVAILABILITY * This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS N/A

SEE ALSO **vmShow**

vmStateGet( )

NAME

vmStateGet( ) – get the state of a page of virtual memory (VxVMI Opt.)

SYNOPSIS

```
STATUS vmStateGet
    (
    VM_CONTEXT_ID context,     /* context - NULL == currentContext */
    void *        pPageAddr,  /* virtual page addr */
    UINT *        pState      /* where to return state */
    )
```

DESCRIPTION

This routine extracts state bits with the following masks:

VM_STATE_MASK_VALID
VM_STATE_MASK_WRITABLE
VM_STATE_MASK_CACHEABLE

Individual states may be identified with the following constants:

VM_STATE_VALID	**VM_STATE_VALID_NOT**	valid/invalid
VM_STATE_WRITABLE	**VM_STATE_WRITABLE_NOT**	writable/write-protected
VM_STATE_CACHEABLE	**VM_STATE_CACHEABLE_NOT**	cacheable/not-cacheable

For example, to see if a page is writable, the following code would be used:

```
vmStateGet (vmContext, pageAddr, &state);
if ((state & VM_STATE_MASK_WRITABLE) & VM_STATE_WRITABLE)
   ...
```

If *context* is specified as **NULL**, the current virtual memory context is used.

This routine is callable from interrupt level.

AVAILABILITY

This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS

OK, or **ERROR** if *pageAddr* is not on a page boundary, the validity check fails, or the architecture-dependent state get fails for the specified virtual address.

ERRNO

S_vmLib_NOT_PAGE_ALIGNED

SEE ALSO

vmLib

vmStateSet( )

NAME

vmStateSet( ) – change the state of a block of virtual memory (VxVMI Opt.)

SYNOPSIS

```
STATUS vmStateSet
    (
    VM_CONTEXT_ID context,    /* context - NULL == currentContext */
    void *        pVirtual,   /* virtual address to modify state of */
    int           len,        /* len of virtual space to modify state of */
    UINT          stateMask,  /* state mask */
    UINT          state       /* state */
    )
```

DESCRIPTION

This routine changes the state of a block of virtual memory. Each page of virtual memory has at least three elements of state information: validity, writability, and cacheability. Specific architectures may define additional state information; see **vmLib.h** for additional architecture-specific states. Memory accesses to a page marked as invalid will result in an exception. Pages may be invalidated to prevent them from being corrupted by invalid references. Pages may be defined as read-only or writable, depending on the state of the writable bits. Memory accesses to pages marked as not-cacheable will always result in a memory cycle, bypassing the cache. This is useful for multiprocessing, multiple bus masters, and hardware control registers.

The following states are provided and may be or'ed together in the state parameter:

VM_STATE_VALID	**VM_STATE_VALID_NOT**	valid/invalid
VM_STATE_WRITABLE	**VM_STATE_WRITABLE_NOT**	writable/write-protected
VM_STATE_CACHEABLE	**VM_STATE_CACHEABLE_NOT**	cacheable/not-cacheable

Additionally, the following masks are provided so that only specific states may be set. These may be or'ed together in the **stateMask** parameter.

VM_STATE_MASK_VALID
VM_STATE_MASK_WRITABLE
VM_STATE_MASK_CACHEABLE

If *context* is specified as **NULL**, the current context is used.

This routine is callable from interrupt level.

AVAILABILITY

This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS

OK or, **ERROR** if the validation fails, *pVirtual* is not on a page boundary, *len* is not a multiple of page size, or the architecture-dependent state set fails for the specified virtual address.

ERRNO **S_vmLib_NOT_PAGE_ALIGNED, S_vmLib_BAD_STATE_PARAM, S_vmLib_BAD_MASK_PARAM**

SEE ALSO **vmLib**

vmTextProtect( )

NAME **vmTextProtect( )** – write-protect a text segment (VxVMI Opt.)

SYNOPSIS

```
STATUS vmTextProtect (void)
```

DESCRIPTION This routine write-protects the VxWorks text segment and sets a flag so that all text segments loaded by the incremental loader will be write-protected. The routine should be called after both **vmLibInit( )** and **vmGlobalMapInit( )** have been called.

AVAILABILITY This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS **OK**, or **ERROR** if the text segment cannot be write-protected.

ERRNO **S_vmLib_TEXT_PROTECTION_UNAVAILABLE**

SEE ALSO **vmLib**

vmTranslate( )

NAME **vmTranslate( )** – translate a virtual address to a physical address (VxVMI Opt.)

SYNOPSIS

```
STATUS vmTranslate
    (
    VM_CONTEXT_ID context,      /* context - NULL == currentContext */
    void *        virtualAddr,  /* virtual address */
    void *        *physicalAddr /* place to put result */
    )
```

DESCRIPTION This routine retrieves mapping information for a virtual address from the page translation tables. If the specified virtual address has never been mapped, the returned status can be either **OK** or **ERROR**; however, if it is **OK**, then the returned physical address will be -1. If *context* is specified as **NULL**, the current context is used.

This routine is callable from interrupt level.

AVAILABILITY This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS **OK**, or **ERROR** if the validation or translation fails.

SEE ALSO **vmLib**

vprintf( )

NAME **vprintf( )** – write a string formatted with a variable argument list to standard output (ANSI)

SYNOPSIS

```
int vprintf
    (
    const char * fmt,         /* format string to write */
    va_list      vaList       /* arguments to format */
    )
```

DESCRIPTION This routine prints a string formatted with a variable argument list to standard output. It is identical to **printf( )**, except that it takes the variable arguments to be formatted as a list *vaList* of type **va_list** rather than as in-line arguments.

RETURNS The number of characters output, or **ERROR** if there is an error during output.

SEE ALSO **fioLib**, **printf( )**, *American National Standard for Information Systems -Programming Language - C, ANSI X3.159-1989: Input/Output (**stdio.h**)*

vsprintf( )

NAME **vsprintf( )** – write a string formatted with a variable argument list to a buffer (ANSI)

SYNOPSIS

```
int vsprintf
    (
    char *       buffer,      /* buffer to write to */
    const char * fmt,         /* format string */
    va_list      vaList       /* optional arguments to format */
    )
```

DESCRIPTION This routine copies a string formatted with a variable argument list to a specified buffer. This routine is identical to **sprintf( )**, except that it takes the variable arguments to be formatted as a list *vaList* of type **va_list** rather than as in-line arguments.

RETURNS The number of characters copied to *buffer*, not including the **NULL** terminator.

SEE ALSO **fioLib**, **sprintf( )**, *American National Standard for Information Systems -Programming Language - C, ANSI X3.159-1989: Input/Output (**stdio.h**)*

vxCr2Get( )

NAME **vxCr2Get( )** – get a content of the Control Register 2 (x86)

SYNOPSIS

```
int vxCr2Get (void)
```

DESCRIPTION This routine gets a content of the Control Register 2.

RETURNS a value of the Control Register 2

SEE ALSO **vxLib**

vxCr2Set( )

NAME **vxCr2Set( )** – set a value to the Control Register 2 (x86)

SYNOPSIS

```
void vxCr2Set
    (
    int value                 /* CR2 value */
    )
```

DESCRIPTION This routine sets a value to the Control Register 2.

RETURNS N/A

SEE ALSO **vxLib**

V

vxCr3Get( )

NAME **vxCr3Get( )** – get a content of the Control Register 3 (x86)

SYNOPSIS

```
int vxCr3Get (void)
```

DESCRIPTION This routine gets a content of the Control Register 3.

RETURNS a value of the Control Register 3

SEE ALSO **vxLib**

vxCr3Set( )

NAME **vxCr3Set( )** – set a value to the Control Register 3 (x86)

SYNOPSIS

```
void vxCr3Set
    (
    int value                 /* CR3 value */
    )
```

DESCRIPTION This routine sets a value to the Control Register 3.

RETURNS N/A

SEE ALSO **vxLib**

vxCr4Get( )

NAME **vxCr4Get( )** – get a content of the Control Register 4 (x86)

SYNOPSIS

```
int vxCr4Get (void)
```

DESCRIPTION This routine gets a content of the Control Register 4.

RETURNS a value of the Control Register 4

SEE ALSO **vxLib**

vxCr4Set( )

NAME
vxCr4Set( ) – set a value to the Control Register 4 (x86)

SYNOPSIS
```
void vxCr4Set
    (
    int value                /* CR4 value */
    )
```

DESCRIPTION
This routine sets a value to the Control Register 4.

RETURNS
N/A

SEE ALSO
vxLib

vxCr0Get( )

NAME
vxCr0Get( ) – get a content of the Control Register 0 (x86)

SYNOPSIS
```
int vxCr0Get (void)
```

DESCRIPTION
This routine gets a content of the Control Register 0.

RETURNS
a value of the Control Register 0

SEE ALSO
vxLib

vxCr0Set( )

NAME **vxCr0Set( )** – set a value to the Control Register 0 (x86)

SYNOPSIS

```
void vxCr0Set
    (
    int value                 /* CR0 value */
    )
```

DESCRIPTION This routine sets a value to the Control Register 0.

RETURNS N/A

SEE ALSO **vxLib**

vxDrGet( )

NAME **vxDrGet( )** – get a content of the Debug Register 0 to 7 (x86)

SYNOPSIS

```
void vxDrGet
    (
    int * pDr0,               /* DR0 */
    int * pDr1,               /* DR1 */
    int * pDr2,               /* DR2 */
    int * pDr3,               /* DR3 */
    int * pDr4,               /* DR4 */
    int * pDr5,               /* DR5 */
    int * pDr6,               /* DR6 */
    int * pDr7                /* DR7 */
    )
```

DESCRIPTION This routine gets a content of the Debug Register 0 to 7.

RETURNS N/A

SEE ALSO **vxLib**

vxDrSet( )

NAME **vxDrSet( )** – set a value to the Debug Register 0 to 7 (x86)

SYNOPSIS

```
void vxDrSet
    (
    int dr0,                  /* DR0 */
    int dr1,                  /* DR1 */
    int dr2,                  /* DR2 */
    int dr3,                  /* DR3 */
    int dr4,                  /* DR4 */
    int dr5,                  /* DR5 */
    int dr6,                  /* DR6 */
    int dr7                   /* DR7 */
    )
```

DESCRIPTION This routine sets a value to the Debug Register 0 to 7.

RETURNS N/A

SEE ALSO **vxLib**

vxEflagsGet( )

NAME **vxEflagsGet( )** – get a content of the EFLAGS register (x86)

SYNOPSIS

```
int vxEflagsGet (void)
```

DESCRIPTION This routine gets a content of the EFLAGS register

RETURNS a value of the EFLAGS register

SEE ALSO **vxLib**

V

vxEflagsSet( )

NAME **vxEflagsSet( )** – set a value to the EFLAGS register (x86)

SYNOPSIS

```
void vxEflagsSet
    (
    int value                    /* EFLAGS value */
    )
```

DESCRIPTION This routine sets a value to the EFLAGS register

RETURNS N/A

SEE ALSO **vxLib**

vxGdtrGet( )

NAME **vxGdtrGet( )** – get a content of the Global Descriptor Table Register (x86)

SYNOPSIS

```
void vxGdtrGet
    (
    long long int * pGdtr        /* memory to store GDTR */
    )
```

DESCRIPTION This routine gets a content of the Global Descriptor Table Register

RETURNS N/A

SEE ALSO **vxLib**

vxIdtrGet( )

NAME
vxIdtrGet( ) – get a content of the Interrupt Descriptor Table Register (x86)

SYNOPSIS
```
void vxIdtrGet
    (
    long long int * pIdtr     /* memory to store IDTR */
    )
```

DESCRIPTION
This routine gets a content of the Interrupt Descriptor Table Register

RETURNS
N/A

SEE ALSO
vxLib

vxLdtrGet( )

NAME
vxLdtrGet( ) – get a content of the Local Descriptor Table Register (x86)

SYNOPSIS
```
void vxLdtrGet
    (
    long long int * pLdtr     /* memory to store LDTR */
    )
```

DESCRIPTION
This routine gets a content of the Local Descriptor Table Register

RETURNS
N/A

SEE ALSO
vxLib

vxMemArchProbe( )

NAME **vxMemArchProbe( )** – architecture-specific part of **vxMemProbe( )**

SYNOPSIS

```
STATUS vxMemArchProbe
    (
    char * adrs,              /* address to be probed */
    int    mode,              /* VX_READ or VX_WRITE */
    int    length,            /* 1, 2, 4, or 8 */
    char * pVal               /* where to return value, or ptr to value */
                              /* to be written */
    )
```

DESCRIPTION This is the routine implementing the architecture specific part of the **vxMemProbe( )** routine. It traps the relevant exceptions while accessing the specified address. If an exception occurs, it returns **ERROR**. If no exception occurs, it returns **OK**.

RETURNS **OK** or **ERROR** if an exception occurred during access.

SEE ALSO **vxLib**

vxMemProbe( )

NAME **vxMemProbe( )** – probe an address for a bus error

SYNOPSIS

```
STATUS vxMemProbe
    (
    char * adrs,              /* address to be probed */
    int    mode,              /* VX_READ or VX_WRITE */
    int    length,            /* 1, 2, 4, or 8 */
    char * pVal               /* where to return value, or ptr to value */
                              /* to be written */
    )
```

DESCRIPTION This routine probes a specified address to see if it is readable or writable, as specified by *mode*. The address is read or written as 1, 2, or 4 bytes, as specified by *length* (values other than 1, 2, or 4 yield unpredictable results). If the probe is a **VX_READ** (0), the value read is copied to the location pointed to by *pVal*. If the probe is a **VX_WRITE** (1), the value written is taken from the location pointed to by *pVal*. In either case, *pVal* should point to a value of 1, 2, or 4 bytes, as specified by *length*.

Note that only bus errors are trapped during the probe, and that the access must otherwise be valid (*i.e.*, it must not generate an address error).

EXAMPLE

```
testMem (adrs)
    char *adrs;
    {
    char testW = 1;
    char testR;
    if (vxMemProbe (adrs, VX_WRITE, 1, &testW) == OK)
        printf ("value %d written to adrs %x\n", testW, adrs);
    if (vxMemProbe (adrs, VX_READ, 1, &testR) == OK)
        printf ("value %d read from adrs %x\n", testR, adrs);
    }
```

MODIFICATION The BSP can modify the behavior of **vxMemProbe( )** by supplying an alternate routine and placing the address in the global variable **_func_vxMemProbeHook**. The BSP routine will be called instead of the architecture specific routine **vxMemArchProbe( )**.

RETURNS **OK**, or **ERROR** if the probe caused a bus error or was misaligned.

SEE ALSO **vxLib**, **vxMemArchProbe( )**

vxPowerDown( )

NAME **vxPowerDown( )** – place the processor in reduced-power mode (PowerPC, SH)

SYNOPSIS

```
UINT32 vxPowerDown (void)
```

DESCRIPTION This routine activates the reduced-power mode if power management is enabled. It is called by the scheduler when the kernel enters the idle loop. The power management mode is selected by **vxPowerModeSet( )**.

RETURNS **OK**, or **ERROR** if power management is not supported or if external interrupts are disabled.

SEE ALSO **vxLib**, **vxPowerModeSet( )**, **vxPowerModeGet( )**

V

vxPowerModeGet( )

NAME
vxPowerModeGet( ) – get the power management mode (PowerPC, SH, x86)

SYNOPSIS
```
UINT32 vxPowerModeGet (void)
```

DESCRIPTION
This routine returns the power management mode set by **vxPowerModeSet( )**.

RETURNS
The power management mode, or **ERROR** if no mode has been selected or if power management is not supported.

SEE ALSO
vxLib, **vxPowerModeSet( )**, **vxPowerDown( )**

vxPowerModeSet( )

NAME
vxPowerModeSet( ) – set the power management mode (PowerPC, SH, x86)

SYNOPSIS
```
STATUS vxPowerModeSet
    (
    UINT32 mode                /* power management mode to select */
    )
```

DESCRIPTION
This routine selects the power management mode to be activated when **vxPowerDown( )** is called. **vxPowerModeSet( )** is normally called in the BSP initialization routine **sysHwInit( )**.

USAGE PPC
Power management modes include the following:

VX_POWER_MODE_DISABLE (0x1)
Power management is disabled; this prevents the MSR(POW) bit from being set (all PPC).

VX_POWER_MODE_FULL (0x2)
All CPU units are active while the kernel is idle (PPC603, PPCEC603 and PPC860 only).

VX_POWER_MODE_DOZE (0x4)
Only the decrementer, data cache, and bus snooping are active while the kernel is idle (PPC603, PPCEC603 and PPC860).

VX_POWER_MODE_NAP (0x8)
Only the decrementer is active while the kernel is idle (PPC603, PPCEC603 and PPC604).

VX_POWER_MODE_SLEEP (0x10)
All CPU units are inactive while the kernel is idle (PPC603, PPCEC603 and PPC860 - not recommended for the PPC603 and PPCEC603 architecture).

VX_POWER_MODE_DEEP_SLEEP (0x20)
All CPU units are inactive while the kernel is idle (PPC860 only - not recommended).

VX_POWER_MODE_DPM (0x40)
Dynamic Power Management Mode (PPC603 and PPCEC603 only).

VX_POWER_MODE_DOWN (0x80)
Only a hard reset causes an exit from power-down low power mode (PPC860 only - not recommended).

USAGE SH

Power management modes include the following:

VX_POWER_MODE_DISABLE (0x0)
Power management is disabled.

VX_POWER_MODE_SLEEP (0x1)
The core CPU is halted, on-chip peripherals operating, external memory refreshing.

VX_POWER_MODE_DEEP_SLEEP (0x2)
The core CPU is halted, on-chip peripherals operating, external memory self-refreshing (SH-4 only).

VX_POWER_MODE_USER (0xff)
Set up to three 8-bit standby registers with user-specified values:

```
vxPowerModeSet (VX_POWER_MODE_USER | sbr1<<8 | sbr2<<16 | sbr3<<24);
```

The sbr1 value is written to the STBCR or SBYCR1, sbr2 is written to the STBCR2 or SBYCR2, and sbr3 is written to the STBCR3 register (when available), depending on the SH processor type.

USAGE x86: **vxPowerModeSet( )** is called in the BSP initialization routine **sysHwInit( )**. Power management modes include the following:

VX_POWER_MODE_DISABLE (0x1)
Power management is disable: this prevents halting the CPU.

VX_POWER_MODE_AUTOHALT (0x4)
Power management is enable: this allows halting the CPU.

RETURNS

OK, or **ERROR** if *mode* is incorrect or not supported by the processor.

SEE ALSO

vxLib, **vxPowerModeGet( )**, **vxPowerDown( )**

V

vxSSDisable( )

NAME **vxSSDisable( )** – disable the superscalar dispatch (MC68060)

SYNOPSIS

```
void vxSSDisable (void)
```

DESCRIPTION This function resets the ESS bit of the Processor Configuration Register (PCR) to disable the superscalar dispatch.

RETURNS N/A

SEE ALSO **vxLib**

vxSSEnable( )

NAME **vxSSEnable( )** – enable the superscalar dispatch (MC68060)

SYNOPSIS

```
void vxSSEnable (void)
```

DESCRIPTION This function sets the ESS bit of the Processor Configuration Register (PCR) to enable the superscalar dispatch.

RETURNS N/A

SEE ALSO **vxLib**

vxTas( )

NAME

vxTas( ) – C-callable atomic test-and-set primitive

SYNOPSIS

```
BOOL vxTas
    (
    void * address            /* address to test and set */
    )
```

DESCRIPTION

This routine provides a C-callable interface to a test-and-set instruction. The test-and-set instruction is executed on the specified address. The architecture test-and-set instruction is:

68K: **tas**
x86: **lock bts**
SH: **tas.b**
ARM **swpb**

This routine is equivalent to **sysBusTas( )** in **sysLib**.

NOTE MIPS

Because VxWorks does not support the MIPS MMU, only kseg0 and kseg1 addresses are accepted; other addresses return **FALSE**.

NOTE x86

BTS "Bit Test and Set" instruction is executed with LOCK instruction prefix to lock the Bus during the execution. The bit position 0 is toggled.

NOTE SH

The SH version of **vxTas( )** simply executes the **tas.b** instruction, and the test-and-set (atomic read-modify-write) operation may require an external bus locking mechanism on some hardware. In this case, wrap the **vxTas( )** with a bus locking and unlocking code in the **sysBusTas( )**.

RETURNS

TRUE if the value had not been set (but is now), or **FALSE** if the value was set already.

SEE ALSO

vxLib, **sysBusTas( )**

vxTssGet( )

NAME
vxTssGet( ) – get a content of the TASK register (x86)

SYNOPSIS
```
int vxTssGet (void)
```

DESCRIPTION
This routine gets a content of the TASK register

RETURNS
a value of the TASK register

SEE ALSO
vxLib

vxTssSet( )

NAME
vxTssSet( ) – set a value to the TASK register (x86)

SYNOPSIS
```
void vxTssSet
    (
    int value                  /* TASK register value */
    )
```

DESCRIPTION
This routine sets a value to the TASK register

RETURNS
N/A

SEE ALSO
vxLib

wcstombs( )

NAME **wcstombs( )** – convert a series of wide char's to multibyte char's (Unimplemented) (ANSI)

SYNOPSIS

```
size_t wcstombs
    (
    char *          s,
    const wchar_t * pwcs,
    size_t          n
    )
```

DESCRIPTION This multibyte character function is unimplemented in VxWorks.

INCLUDE FILES **stdlib.h**

RETURNS **OK**, or **ERROR** if the parameters are invalid.

SEE ALSO **ansiStdlib**

wctomb( )

NAME **wctomb( )** – convert a wide character to a multibyte character (Unimplemented) (ANSI)

SYNOPSIS

```
int wctomb
    (
    char *  s,
    wchar_t wchar
    )
```

DESCRIPTION This multibyte character function is unimplemented in VxWorks.

INCLUDE FILES **stdlib.h**

RETURNS **OK**, or **ERROR** if the parameters are invalid.

SEE ALSO **ansiStdlib**

wdbSystemSuspend( )

NAME **wdbSystemSuspend( )** – suspend the system.

SYNOPSIS

```
STATUS wdbSystemSuspend (void)
```

DESCRIPTION This routine transfers control from the run time system to the WDB agent running in external mode. In order to give back the control to the system it must be resumed by the the external WDB agent.

EXAMPLE The code below, called in a vxWorks application, suspends the system:

```
if (wdbSystemSuspend != OK)
    printf ("External mode is not supported by the WDB agent.\n");
```

From a host tool, we can detect that the system is suspended.

First, attach to the target server:

```
wtxtcl> wtxToolAttach EP960CX
EP960CX_ps@sevre
```

Then, you can get the agent mode:

```
wtxtcl> wtxAgentModeGet
AGENT_MODE_EXTERN
```

To get the status of the system context, execute:

```
wtxtcl> wtxContextStatusGet CONTEXT_SYSTEM 0
CONTEXT_SUSPENDED
```

In order to resume the system, simply execute:

```
wtxtcl>  wtxContextResume CONTEXT_SYSTEM 0
0
```

You will see that the system is now running:

```
wtxtcl> wtxContextStatusGet CONTEXT_SYSTEM 0
CONTEXT_RUNNING
```

RETURNS **OK** upon successful completion, **ERROR** if external mode is not supported by the WDB agent.

SEE ALSO **wdbLib**

wdbUserEvtLibInit( )

NAME

wdbUserEvtLibInit( ) – include the WDB user event library

SYNOPSIS

```
void wdbUserEvtLibInit (void)
```

DESCRIPTION

This null routine is provided so that **wdbUserEvtLib** can be linked into the system. If **INCLUDE_WDB_USER_EVENT** is defined in **configAll.h**, **wdbUserEvtLibInit( )** is called by the WDB config routine, **wdbConfig( )**, in **usrWdb.c**.

RETURNS

N/A

SEE ALSO

wdbUserEvtLib

wdbUserEvtPost( )

NAME

wdbUserEvtPost( ) – post a user event string to host tools.

SYNOPSIS

```
STATUS wdbUserEvtPost
    (
    char * event              /* event string to send */
    )
```

DESCRIPTION

This routine posts the string *event* to host tools that have registered for it. Host tools will receive a USER WTX event string. The maximum size of the event is **WDB_MAX_USER_EVT_SIZE** (defined in $**WIND_BASE/target/h/wdb/wdbLib.h**).

EXAMPLE

The code below sends a WDB user event to host tools:

```
char * message = "Alarm: reactor overheating !!!";
if (wdbUserEvtPost (message) != OK)
    printf ("Can't send alarm message to host tools");
```

This event will be received by host tools that have registered for it. For example a WTX TCL based tool would do:

```
wtxtcl> wtxToolAttach EP960CX
EP960CX_ps@sevre
wtxtcl> wtxRegisterForEvent "USER.*"
0
wtxtcl> wtxEventGet
```

W

```
USER Alarm: reactor overheating !!!
```

Host tools can register for more specific user events:

```
wtxtcl> wtxToolAttach EP960CX
EP960CX_ps@sevre
wtxtcl> wtxRegisterForEvent "USER Alarm.*"
0
wtxtcl> wtxEventGet
USER Alarm: reactor overheating !!!
```

In this piece of code, only the USER events beginning with "Alarm" will be received.

RETURNS
OK upon successful completion, a WDB error code if unable to send the event to the host or **ERROR** if the size of the event is greater than **WDB_MAX_USER_EVT_SIZE**.

SEE ALSO
wdbUserEvtLib

wdCancel( )

NAME
wdCancel( ) – cancel a currently counting watchdog

SYNOPSIS

```
STATUS wdCancel
    (
    WDOG_ID wdId              /* ID of watchdog to cancel */
    )
```

DESCRIPTION
This routine cancels a currently running watchdog timer by zeroing its delay count. Watchdog timers may be canceled from interrupt level.

RETURNS
OK, or **ERROR** if the watchdog timer cannot be canceled.

SEE ALSO
wdLib, **wdStart( )**

wdCreate( )

NAME **wdCreate( )** – create a watchdog timer

SYNOPSIS

```
WDOG_ID wdCreate (void)
```

DESCRIPTION This routine creates a watchdog timer by allocating a WDOG structure in memory.

RETURNS The ID for the watchdog created, or **NULL** if memory is insufficient.

SEE ALSO **wdLib**, **wdDelete( )**

wdDelete( )

NAME **wdDelete( )** – delete a watchdog timer

SYNOPSIS

```
STATUS wdDelete
    (
    WDOG_ID wdId              /* ID of watchdog to delete */
    )
```

DESCRIPTION This routine de-allocates a watchdog timer. The watchdog will be removed from the timer queue if it has been started. This routine complements **wdCreate( )**.

RETURNS **OK**, or **ERROR** if the watchdog timer cannot be de-allocated.

SEE ALSO **wdLib**, **wdCreate( )**

wdShow( )

NAME **wdShow( )** – show information about a watchdog

SYNOPSIS

```
STATUS wdShow
    (
    WDOG_ID wdId              /* watchdog to display */
    )
```

DESCRIPTION This routine displays the state of a watchdog.

EXAMPLE A summary of the state of a watchdog is displayed as follows:

```
-> wdShow myWdId
Watchdog Id          : 0x3dd46c
State                : OUT_OF_Q
Ticks Remaining      : 0
Routine              : 0
Parameter            : 0
```

RETURNS **OK** or **ERROR**.

SEE ALSO **wdShow**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

wdShowInit( )

NAME **wdShowInit( )** – initialize the watchdog show facility

SYNOPSIS

```
void wdShowInit (void)
```

DESCRIPTION This routine links the watchdog show facility into the VxWorks system. It is called automatically when the watchdog show facility is configured into VxWorks using either of the following methods:

– If you use the configuration header files, define **INCLUDE_SHOW_ROUTINES** in **config.h**.

– If you use the Tornado project facility, select **INCLUDE_WATCHDOGS_SHOW**.

RETURNS N/A

SEE ALSO **wdShow**

wdStart( )

NAME

wdStart( ) – start a watchdog timer

SYNOPSIS

```
STATUS wdStart
    (
    WDOG_ID wdId,           /* watchdog ID */
    int     delay,          /* delay count, in ticks */
    FUNCPTR pRoutine,       /* routine to call on time-out */
    int     parameter       /* parameter with which to call routine */
    )
```

DESCRIPTION

This routine adds a watchdog timer to the system tick queue. The specified watchdog routine will be called from interrupt level after the specified number of ticks has elapsed. Watchdog timers may be started from interrupt level.

To replace either the timeout *delay* or the routine to be executed, call **wdStart( )** again with the same *wdId*; only the most recent **wdStart( )** on a given watchdog ID has any effect. (If your application requires multiple watchdog routines, use **wdCreate( )** to generate separate a watchdog ID for each.) To cancel a watchdog timer before the specified tick count is reached, call **wdCancel( )**.

Watchdog timers execute only once, but some applications require periodically executing timers. To achieve this effect, the timer routine itself must call **wdStart( )** to restart the timer on each invocation.

WARNING: The watchdog routine runs in the context of the system-clock ISR; thus, it is subject to all ISR restrictions.

RETURNS

OK, or **ERROR** if the watchdog timer cannot be started.

SEE ALSO

wdLib, **wdCancel( )**

whoami( )

NAME **whoami( )** – display the current remote identity

SYNOPSIS

```
void whoami (void)
```

DESCRIPTION This routine displays the user name currently used for remote machine access. The user name is set with **iam( )** or **remCurIdSet( )**.

RETURNS N/A

SEE ALSO **remLib**, **iam( )**, **remCurIdGet( )**, **remCurIdSet( )**

write( )

NAME **write( )** – write bytes to a file

SYNOPSIS

```
int write
    (
    int    fd,                /* file descriptor on which to write */
    char * buffer,            /* buffer containing bytes to be written */
    size_t nbytes             /* number of bytes to write */
    )
```

DESCRIPTION This routine writes *nbytes* bytes from *buffer* to a specified file descriptor *fd*. It calls the device driver to do the work.

RETURNS The number of bytes written (if not equal to *nbytes*, an error has occurred), or **ERROR** if the file descriptor does not exist, the driver does not have a write routine, or the driver returns **ERROR**. If the driver does not have a write routine, **errno** is set to **ENOTSUP**.

SEE ALSO **ioLib**

wvEvent( )

NAME **wvEvent( )** – log a user-defined event (WindView)

SYNOPSIS

```
STATUS wvEvent
    (
    event_t usrEventId,       /* event */
    char *  buffer,           /* buffer */
    size_t  bufSize           /* buffer size */
    )
```

DESCRIPTION This routine logs a user event. Event logging must have been started with **wvEvtLogEnable( )** or from the WindView GUI to use this routine. The *usrEventId* should be in the range 0-25535. A buffer of data can be associated with the event; *buffer* is a pointer to the start of the data block, and *bufSize* is its length in bytes. The size of the event buffer configured with **wvInstInit( )** should be adjusted when logging large user events.

RETURNS **OK**, or **ERROR** if the event can not be logged.

SEE ALSO **wvLib**, **dbgLib**, **e( )**

wvEventInst( )

NAME **wvEventInst( )** – instrument VxWorks Events (WindView)

SYNOPSIS

```
STATUS wvEventInst
    (
    int mode                  /* instrumentation mode */
    )
```

DESCRIPTION This routine instruments VxWorks Event activity.

If *mode* is **INSTRUMENT_ON**, instrumentation for events is turned on; if it is any other value (including **INSTRUMENT_OFF**), instrumentation for VxWorks Events is turned off.

This routine has effect only if **INCLUDE_WINDVIEW** is defined in **configAll.h** and event logging has been enabled for system objects.

RETURNS **OK** or **ERROR**.

SEE ALSO **wvLib**

wvEvtBufferGet( )

NAME **wvEvtBufferGet( )** – return the ID of the WindView event buffer (WindView)

SYNOPSIS

```
BUFFER_ID wvEvtBufferGet (void)
```

RETURNS The event buffer ID if one exists, otherwise **NULL**.

SEE ALSO **wvLib**

wvEvtClassClear( )

NAME **wvEvtClassClear( )** – clear a class of events from those being logged (WindView)

SYNOPSIS

```
void wvEvtClassClear
    (
    UINT32 classDescription   /* description of evt classes to clear */
    )
```

DESCRIPTION This routine clears the class or classes described by *classDescription* from the set of classes currently being logged.

RETURNS N/A

SEE ALSO **wvLib**

wvEvtClassClearAll( )

NAME **wvEvtClassClearAll( )** – clear all classes of events from those logged (WindView)

SYNOPSIS

```
void wvEvtClassClearAll (void)
```

DESCRIPTION This routine clears all classes of events so that no classes are logged if event logging is started.

RETURNS N/A

SEE ALSO **wvLib**

wvEvtClassGet( )

NAME **wvEvtClassGet( )** – get the current set of classes being logged (WindView)

SYNOPSIS

```
UINT32 wvEvtClassGet (void)
```

DESCRIPTION This routine returns the set of classes currently being logged.

RETURNS The class description.

SEE ALSO **wvLib**

wvEvtClassSet( )

NAME **wvEvtClassSet( )** – set the class of events to log (WindView)

SYNOPSIS

```
void wvEvtClassSet
    (
    UINT32 classDescription   /* description of evt classes to set */
    )
```

DESCRIPTION This routine sets the class of events which are logged when event logging is started. *classDescription* can take the following values:

```
WV_CLASS_1      /* Events causing context switches */
WV_CLASS_2      /* Events causing task-state transitions */
WV_CLASS_3      /* Events from object and system libraries */
```

See **wvLib** for more information about these classes, particularly Class 3.

RETURNS N/A

SEE ALSO **wvLib**, **wvObjInst( )**, **wvObjInstModeSet( )**, **wvSigInst( )**, **wvEventInst( )**

wvEvtLogInit( )

NAME **wvEvtLogInit( )** – initialize an event log (WindView)

SYNOPSIS

```
void wvEvtLogInit
    (
    BUFFER_ID evtBufId        /* event-buffer id */
    )
```

DESCRIPTION This routine initializes event logging by associating a particular event buffer with the logging functions. It must be called before event logging is turned on.

RETURNS N/A

SEE ALSO **wvLib**

wvEvtLogStart( )

NAME **wvEvtLogStart( )** – start logging events to the buffer (WindView)

SYNOPSIS

```
void wvEvtLogStart (void)
```

DESCRIPTION This routine starts event logging. It also resets the timestamp mechanism so that it can be called more than once without stopping event logging.

RETURNS N/A

SEE ALSO **wvLib**

wvEvtLogStop( )

NAME **wvEvtLogStop( )** – stop logging events to the buffer (WindView)

SYNOPSIS

```
void wvEvtLogStop (void)
```

DESCRIPTION This routine turns off all event logging, including event-logging of objects and signals specifically requested by the user. In addition, it disables the timestamp facility.

RETURNS N/A

SEE ALSO **wvLib**

wvLibInit( )

NAME **wvLibInit( )** – initialize **wvLib** - first step (WindView)

SYNOPSIS

```
void wvLibInit (void)
```

DESCRIPTION This routine starts initializing **wvLib**. Its actions should be performed before object creation, so it is called from **usrKernelInit( )** in **usrKernel.c**.

RETURNS N/A

SEE ALSO **wvLib**

wvLibInit2( )

NAME **wvLibInit2( )** – initialize **wvLib** - final step (WindView)

SYNOPSIS

```
void wvLibInit2 (void)
```

DESCRIPTION This routine is called after **wvLibInit( )** to complete the initialization of **wvLib**. It should be called before starting any event logging.

RETURNS N/A

SEE ALSO **wvLib**

wvLogHeaderCreate( )

NAME **wvLogHeaderCreate( )** – create the event-log header (WindView)

SYNOPSIS

```
WV_LOG_HEADER_ID wvLogHeaderCreate
    (
    PART_ID memPart            /* partition where header should be stored */
    )
```

DESCRIPTION This routine creates the header of **EVENT_CONFIG**, **EVENT_BUFFER**, and **EVENT_BEGIN** events that is required at the beginning of every event log. These events are stored in a packed array allocated from the specified memory partition. In addition to this separate header, this routine also logs all tasks active in the system to the event buffer for uploading along with the other events.

This routine should be called after **wvEvtLogInit( )** is called. If uploading events continuously to the host, this routine should be called after the upload task is started. This ensures that the upload task is included in the snapshot of active tasks. If upload will occur after event logging has stopped (deferred upload), this routine can be called any time before event logging is turned on.

RETURNS A valid **WV_LOG_HEADER_ID**, or **NULL** if memory can not be allocated.

SEE ALSO **wvLib**

wvLogHeaderUpload( )

NAME **wvLogHeaderUpload( )** – transfer the log header to the host (WindView)

SYNOPSIS

```
STATUS wvLogHeaderUpload
    (
    WV_LOG_HEADER_ID pHeader, /* pointer to the header */
    UPLOAD_ID        pathId   /* path by which to upload to host */
    )
```

DESCRIPTION This functions transfers the log header events (**EVENT_BEGIN**, **EVENT_CONFIG**, **EVENT_BUFFER**) to the host. These events were saved to a local buffer with the call to **wvLogHeaderCreate( )**. This routine should be called before any events or task names are uploaded to the host. The events in the header buffer must be the first things the parser sees.

If continuously uploading events, it is best to start the uploader, and then call this routine. If deferring upload until after event logging is stopped, this should be called before the uploader is started.

RETURNS **OK**, or **ERROR** if there is trouble with the upload path.

SEE ALSO **wvLib**

wvNetAddressFilterClear( )

NAME **wvNetAddressFilterClear( )** – remove the address filter for events

SYNOPSIS

```
void wvNetAddressFilterClear
    (
    int type,                 /* 0 for source, 1 for destination */
    int direction             /* 0 for input, 1 for output */
    )
```

DESCRIPTION This routine removes any active address filter test indicated by the *type* and *direction* parameters used to enable it. Affected events will be reported unconditionally.

RETURNS N/A

SEE ALSO **wvNetLib**

wvNetAddressFilterSet( )

NAME **wvNetAddressFilterSet( )** – specify an address filter for events

SYNOPSIS

```
STATUS wvNetAddressFilterSet
    (
    char * pAddress,          /* target address for event comparisons */
    char * pMask,             /* mask value applied to data fields */
    int    type,              /* 0 for source, 1 for destination */
    int    direction          /* 0 for input, 1 for output */
    )
```

W

DESCRIPTION This routine activates an additional test that disables certain events that do not match the specified IP address. The *pAddress* parameter provides the test value in dotted-decimal format. The *type* parameter indicates whether that address is compared against the source or destination values, and the *direction* value identifies whether the *type* is interpreted from the perspective of incoming or outgoing traffic. The *pMask* parameter provides a network mask to support testing for a group of events.

RETURNS **OK** if filter set, or **ERROR** otherwise.

ERRNO N/A

SEE ALSO **wvNetLib**

wvNetDisable( )

NAME **wvNetDisable( )** – end reporting of network events to WindView

SYNOPSIS

```
void wvNetDisable (void)
```

DESCRIPTION This routine stops WindView event reporting for all network components.

RETURNS N/A

ERRNO N/A

SEE ALSO **wvNetLib**

wvNetEnable( )

NAME **wvNetEnable( )** – begin reporting network events to WindView

SYNOPSIS

```
void wvNetEnable
    (
    int priority              /* minimum priority, or 0 for default of */
                              /* WV_NET_VERBOSE */
    )
```

DESCRIPTION

This routine activates WindView event reporting for network components, after disabling all events with a priority less than *level*. The default value (or a *level* of **WV_NET_VERBOSE**) will not disable any additional events. The available priority values are:

WV_NET_EMERGENCY (1)
WV_NET_ALERT (2)
WV_NET_CRITICAL (3)
WV_NET_ERROR (4)
WV_NET_WARNING (5)
WV_NET_NOTICE (6)
WV_NET_INFO (7)
WV_NET_VERBOSE (8)

If an event is not explicitly disabled by the priority level, it uses the current event selection map and class settings. The initial values enable all events of both classes.

RETURNS

N/A

ERRNO

N/A

SEE ALSO

wvNetLib

wvNetEventDisable( )

NAME

wvNetEventDisable( ) – deactivate specific network events

SYNOPSIS

```
STATUS wvNetEventDisable
    (
    int priority,             /* priority level of event */
    int offset                /* identifier within priority level */
    )
```

DESCRIPTION

This routine prevents reporting of a single event within the priority equal to *level*. The activation is overridden if the setting for the entire priority level changes. The available priority values are:

WV_NET_EMERGENCY (1)
WV_NET_ALERT (2)
WV_NET_CRITICAL (3)
WV_NET_ERROR (4)
WV_NET_WARNING (5)
WV_NET_NOTICE (6)

WV_NET_INFO (7)
WV_NET_VERBOSE (8)

Offset values for individual events are listed in the documentation.

RETURNS **OK**, or **ERROR** for unknown event.

ERRNO N/A

SEE ALSO **wvNetLib**

wvNetEventEnable( )

NAME **wvNetEventEnable( )** – activate specific network events

SYNOPSIS

```
STATUS wvNetEventEnable
    (
    int priority,             /* priority level of event */
    int offset                /* identifier within priority level */
    )
```

DESCRIPTION This routine allows reporting of a single event within the priority equal to *level*. The activation is overridden if the setting for the entire priority level changes. The available priority values are:

WV_NET_EMERGENCY (1)
WV_NET_ALERT (2)
WV_NET_CRITICAL (3)
WV_NET_ERROR (4)
WV_NET_WARNING (5)
WV_NET_NOTICE (6)
WV_NET_INFO (7)
WV_NET_VERBOSE (8)

Offset values for individual events are listed in the documentation.

RETURNS **OK**, or **ERROR** for unknown event.

ERRNO N/A

SEE ALSO **wvNetLib**

wvNetLevelAdd( )

NAME **wvNetLevelAdd( )** – enable network events with specific priority level

SYNOPSIS

```
STATUS wvNetLevelAdd
    (
    int priority              /* priority level to enable */
    )
```

DESCRIPTION This routine changes the event selection map to allow reporting of any events with priority equal to *level*. It will override current event selections for the given priority, but has no effect on settings for events with higher or lower priorities. The available priority values are:

WV_NET_EMERGENCY (1)
WV_NET_ALERT (2)
WV_NET_CRITICAL (3)
WV_NET_ERROR (4)
WV_NET_WARNING (5)
WV_NET_NOTICE (6)
WV_NET_INFO (7)
WV_NET_VERBOSE (8)

Events are only reported based on the current WindView class setting. The initial (default) setting includes networking events from both classes.

RETURNS **OK**, or **ERROR** for unknown event level.

ERRNO N/A

SEE ALSO **wvNetLib**

wvNetLevelRemove( )

NAME **wvNetLevelRemove( )** – disable network events with specific priority level

SYNOPSIS

```
STATUS wvNetLevelRemove
    (
    int priority              /* priority level to disable */
    )
```

DESCRIPTION

This routine changes the event selection map to prevent reporting of any events with priority equal to *level*. It will override the current event selection for the given priority, but has no effect on settings for events with higher or lower priorities. The available priority values are:

WV_NET_EMERGENCY (1)
WV_NET_ALERT (2)
WV_NET_CRITICAL (3)
WV_NET_ERROR (4)
WV_NET_WARNING (5)
WV_NET_NOTICE (6)
WV_NET_INFO (7)
WV_NET_VERBOSE (8)

Events are only reported based on the current WindView class setting. The initial (default) setting includes networking events from both classes.

RETURNS

OK, or **ERROR** for unknown event level.

ERRNO

N/A

SEE ALSO

wvNetLib

wvNetPortFilterClear( )

NAME

wvNetPortFilterClear( ) – remove the port number filter for events

SYNOPSIS

```
void wvNetPortFilterClear
    (
    int type,                /* 0 for source, 1 for destination */
    int direction            /* 0 for input, 1 for output */
    )
```

DESCRIPTION

This routine removes any active port filter test indicated by the *type* and *direction* parameters used to enable it. Affected events will be reported unconditionally.

RETURNS

N/A

ERRNO

N/A

SEE ALSO

wvNetLib

wvNetPortFilterSet( )

NAME

wvNetPortFilterSet( ) – specify an address filter for events

SYNOPSIS

```
STATUS wvNetPortFilterSet
    (
    int port,                 /* target port for event comparisons */
    int type,                 /* 0 for source, 1 for destination */
    int direction             /* 0 for input, 1 for output */
    )
```

DESCRIPTION

This routine activates an additional filter, which disables certain events that do not match the specified port value. The *port* parameter provides the test value and the *type* parameter indicates whether that value is compared against the source or destination fields. The *direction* setting identifies whether the *type* is interpreted from the perspective of incoming or outgoing traffic.

RETURNS

OK if filter set, or **ERROR** otherwise.

ERRNO

N/A

SEE ALSO

wvNetLib

wvObjInst( )

NAME

wvObjInst( ) – instrument objects (WindView)

SYNOPSIS

```
STATUS wvObjInst
    (
    int    objType,           /* object type */
    void * objId,             /* object ID or NULL for all objects */
    int    mode               /* instrumentation mode */
    )
```

DESCRIPTION

This routine instruments a specified object or set of objects and has effect when system objects have been enabled for event logging.

objType can be set to one of the following: **OBJ_TASK** (tasks), **OBJ_SEM** (semaphores), **OBJ_MSG** (message queues), or **OBJ_WD** (watchdogs). *objId* specifies the identifier of the

W

particular object to be instrumented. If *objId* is **NULL**, then all objects of *objType* have instrumentation turned on or off depending on the value of *mode*.

If *mode* is **INSTRUMENT_ON**, instrumentation is turned on; if it is any other value (including **INSTRUMENT_OFF**) then instrumentation is turned off for *objId*.

Call **wvObjInstModeSet( )** with **INSTRUMENT_ON** if you want to enable instrumentation for all objects created after a certain place in your code. Use **wvSigInst( )** if you want to enable instrumentation for all signal activity.

This routine has effect only if **INCLUDE_WINDVIEW** is defined in **configAll.h**.

RETURNS **OK** or **ERROR**.

SEE ALSO **wvLib**, **wvSigInst( )**, **wvEventInst( )**, **wvObjInstModeSet( )**

wvObjInstModeSet( )

NAME **wvObjInstModeSet( )** – set object instrumentation on/off (WindView)

SYNOPSIS

```
STATUS wvObjInstModeSet
    (
    int mode                   /* object instrumentation on/off */
    )
```

DESCRIPTION This routine causes objects to be created either instrumented or not depending on the value of *mode*, which can be **INSTRUMENT_ON** or **INSTRUMENT_OFF**. All objects created after **wvObjInstModeSet( )** is called with **INSTRUMENT_ON** and before it is called with **INSTRUMENT_OFF** are created as instrumented objects.

Use **wvObjInst( )** if you want to enable instrumentation for a specific object or set of objects. Use **wvSigInst( )** if you want to enable instrumentation for all signal activity, and **wvEventInst( )** to enable instrumentation for VxWorks Event activity.

This routine has effect only if **INCLUDE_WINDVIEW** is defined in **configAll.h**.

RETURNS The previous value of *mode* or **ERROR**.

SEE ALSO **wvLib**, **wvObjInst( )**, **wvSigInst( )**, **wvEventInst( )**

wvRBuffMgrPrioritySet( )

NAME **wvRBuffMgrPrioritySet( )** – set the priority of the WindView rBuff manager (WindView)

SYNOPSIS

```
STATUS wvRBuffMgrPrioritySet
    (
    int priority              /* new priority */
    )
```

DESCRIPTION This routine changes the priority of the **tWvRBuffMgr** task to the value of *priority*. Priorities range from 0, the highest priority, to 255, the lowest priority. If the task is not yet running, this priority is used when it is spawned.

RETURNS **OK**, or **ERROR** if the priority can not be set.

SEE ALSO **rBuffLib**, **taskPrioritySet( )**, *VxWorks Programmer's Guide: Basic OS*

wvSigInst( )

NAME **wvSigInst( )** – instrument signals (WindView)

SYNOPSIS

```
STATUS wvSigInst
    (
    int mode                  /* instrumentation mode */
    )
```

DESCRIPTION This routine instruments all signal activity.

If *mode* is **INSTRUMENT_ON**, instrumentation for signals is turned on; if it is any other value (including **INSTRUMENT_OFF**), instrumentation for signals is turned off.

This routine has effect only if **INCLUDE_WINDVIEW** is defined in **configAll.h** and event logging has been enabled for system objects.

RETURNS **OK** or **ERROR**.

SEE ALSO **wvLib**

W

wvTaskNamesPreserve( )

NAME

wvTaskNamesPreserve( ) – preserve an extra copy of task name events (WindView)

SYNOPSIS

```
TASKBUF_ID wvTaskNamesPreserve
    (
    PART_ID memPart,          /* memory where preserved names are stored */
    int     size              /* must be a power of 2 */
    )
```

DESCRIPTION

This routine initializes the data structures and instrumentation necessary to allow WindView to store an extra copy of each **EVENT_TASKNAME** event, which is necessary for post-mortem analysis. This routine should be called after **wvEvtLogInit( )** has been called, and before event logging is started.

If this routine is called before event logging is started, all **EVENT_TASKNAME** events that are produced by VxWorks are logged into the standard event buffer, and a copy of each is logged automatically to the task name buffer created by this routine. All tasks running when this routine is called are also added to the buffer. The events in this buffer can be uploaded after the other events have been uploaded, to provide the task names for any events in the log which no longer have a corresponding task name event due to wrapping of data in the buffers. Because there may be two copies of some of the task name events after the buffer data wraps around, the resultant log may have two task name events for the same task. This is not a problem for the parser.

Occasionally the task ID of a task is reused, and in this case, only the last instance of the task name event with a particular task ID is maintained.

The buffer size must be a power of two.

This routine sets the event class **WV_CLASS_TASKNAMES_PRESERVE**, which can be turned off by calling **wvEvtClassClear( )** or **wvEvtClassSet( )**.

RETURNS

A valid **TASKBUF_ID** to be used for later uploading, or **NULL** if not enough memory exists to create the task buffer.

SEE ALSO

wvLib

wvTaskNamesUpload( )

NAME **wvTaskNamesUpload( )** – upload preserved task name events (WindView)

SYNOPSIS

```
STATUS wvTaskNamesUpload
    (
    TASKBUF_ID taskBufId,      /* taskname event buffer to upload */
    UPLOAD_ID  pathId          /* upload path id */
    )
```

DESCRIPTION This routine uploads task name events, saved after calling **wvTaskNamesPreserve( )**, to the host by the specified upload path. There is no particular order to the events uploaded. All the events contained in the buffer are uploaded in one pass. After all have been uploaded, the buffer used to store the events is destroyed.

RETURNS **OK**, or **ERROR** if the upload path or task name buffer is invalid.

SEE ALSO **wvLib**

wvTmrRegister( )

NAME **wvTmrRegister( )** – register a timestamp timer (WindView)

SYNOPSIS

```
void wvTmrRegister
    (
    UINTFUNCPTR wvTmrRtn,      /* timestamp routine */
    UINTFUNCPTR wvTmrLockRtn,  /* locked timestamp routine */
    FUNCPTR     wvTmrEnable,   /* enable timer routine */
    FUNCPTR     wvTmrDisable,  /* disable timer routine */
    FUNCPTR     wvTmrConnect,  /* connect to timer routine */
    UINTFUNCPTR wvTmrPeriod,   /* period of timer routine */
    UINTFUNCPTR wvTmrFreq      /* frequency of timer routine */
    )
```

DESCRIPTION This routine registers a timestamp routine for each of the following:

wvTmrRtn
a timestamp routine, which returns a timestamp when called (must be called with interrupts locked).

W

wvTmrLockRtn
a timestamp routine, which returns a timestamp when called (locks interrupts).

wvTmrEnable
an enable-timer routine, which enables the timestamp timer.

wvTmrDisable
a disable-timer routine, which disables the timestamp timer.

wvTmrConnect
a connect-to-timer routine, which connects a handler to be run when the timer rolls over; this routine should return **NULL** if the system clock tick is to be used.

wvTmrPeriod
a period-of-timer routine, which returns the period of the timer.

wvTmrFreq
a frequency-of-timer routine, which returns the frequency of the timer.

If any of these routines is set to **NULL**, the behavior of instrumented code is undefined.

RETURNS N/A

SEE ALSO **wvTmrLib**

wvUploadStart( )

NAME **wvUploadStart( )** – start upload of events to the host (WindView)

SYNOPSIS

```
WV_UPLOADTASK_ID wvUploadStart
    (
    BUFFER_ID bufId,              /* event data buffer ID */
    UPLOAD_ID pathId,             /* upload path to host */
    BOOL      uploadContinuously /* upload continuously if true */
    )
```

DESCRIPTION This routine starts uploading events from the event buffer to the host. Events can be uploaded either continuously or in one pass until the buffer is emptied. If *uploadContinuously* is set to **TRUE**, the task uploading events pends until more data arrives in the buffer. If **FALSE**, the buffer is flushed without waiting, but this routine returns immediately with an ID that can be used to kill the upload task. Upload is done by spawning the task **tWVUpload**. The buffer to upload is identified by *bufId*, and the upload path to use is identified by *pathId*.

This routine blocks if no event data is in the buffer, so it should be called before event logging is started to ensure the buffer does not overflow.

RETURNS A valid **WV_UPLOADTASK_ID** if started for continuous upload, a non-**NULL** value if started for one-pass upload, and **NULL** if the task can not be spawned or memory for the descriptor can not be allocated.

SEE ALSO **wvLib**

---

wvUploadStop( )

NAME **wvUploadStop( )** – stop upload of events to host (WindView)

SYNOPSIS

```
STATUS wvUploadStop
    (
    WV_UPLOADTASK_ID upTaskId
    )
```

DESCRIPTION This routine stops continuous upload of events to the host. It does this by making a request to the upload task to terminate after it has emptied the buffer. For this reason it is important to make sure data is no longer being logged to the buffer before calling this routine.

This task blocks until the buffer is emptied, and then frees memory associated with *upTaskId*.

RETURNS **OK** if the upload task terminates successfully, or **ERROR** either if *upTaskId* is invalid or if the upload task terminates with an **ERROR**.

SEE ALSO **wvLib**

W

wvUploadTaskConfig( )

NAME

wvUploadTaskConfig( ) – set priority and stack size of **tWVUpload** task (WindView)

SYNOPSIS

```
void wvUploadTaskConfig
    (
    int stackSize,            /* the new stack size for tWVUpload */
    int priority              /* the new priority for tWVUpload */
    )
```

DESCRIPTION

This routine sets the stack size and priority of future instances of the event-data upload task, created by calling **wvUploadStart( )**. The default stack size for this task is 5000 bytes, and the default priority is 150.

RETURNS

N/A

SEE ALSO

wvLib

xattrib( )

NAME

xattrib( ) – modify MS-DOS file attributes of many files

SYNOPSIS

```
STATUS xattrib
    (
    const char * source,      /* file or directory 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")
```

WARNING: This function may call itself in accordance with the depth of the source directory, and occupies approximately 520 bytes per stack frame, meaning that to accommodate the maximum depth of subdirectories which is 20, at least 10 Kbytes of stack space should be available to avoid stack overflow.

RETURNS

OK, or **ERROR** if the file can not be opened.

SEE ALSO

usrFsLib

X

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.

WARNING: This function may call itself in accordance with the depth of the source directory, and occupies approximately 800 bytes per stack frame, meaning that to accommodate the maximum depth of subdirectories which is 20, at least 16 Kbytes of stack space should be available to avoid stack overflow.

RETURNS **OK** or **ERROR** if any operation has failed.

SEE ALSO **usrFsLib**, **tarLib**, **checkStack( )**, **cp( )**

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.

WARNING: This function may call itself in accordance with the depth of the source directory, and occupies approximately 520 bytes per stack frame, meaning that to accommodate the maximum depth of subdirectories which is 20, at least 10 Kbytes of stack space should be available to avoid stack overflow.

RETURNS **OK** or **ERROR** if any operation has failed.

SEE ALSO **usrFsLib**, **checkStack( )**, **cp( )**, **copy( )**, **xcopy( )**, **tarLib**

zbufCreate( )

NAME

zbufCreate( ) – create an empty zbuf

SYNOPSIS

```
ZBUF_ID zbufCreate (void)
```

DESCRIPTION

This routine creates a zbuf, which remains empty (that is, it contains no data) until segments are added by the zbuf insertion routines. Operations performed on zbufs require a zbuf ID, which is returned by this routine.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, the returned ID is valid within the kernel protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

A zbuf ID, or **NULL** if a zbuf cannot be created.

SEE ALSO

zbufLib, **zbufDelete( )**

zbufCut( )

NAME

zbufCut( ) – delete bytes from a zbuf

SYNOPSIS

```
ZBUF_SEG zbufCut
    (
    ZBUF_ID  zbufId,          /* zbuf from which bytes are cut */
    ZBUF_SEG zbufSeg,         /* zbuf segment base for offset */
    int      offset,          /* relative byte offset */
    int      len              /* number of bytes to cut */
    )
```

DESCRIPTION

This routine deletes *len* bytes from *zbufId* starting at the specified byte location.

The starting location of deletion is specified by *zbufSeg* and *offset*. See the **zbufLib** manual page for more information on specifying a byte location within a zbuf. In particular, the first byte deleted is the exact byte specified by *zbufSeg* and *offset*.

The number of bytes to delete is given by *len*. If this parameter is negative, or is larger than the number of bytes in the zbuf after the specified byte location, the rest of the zbuf is deleted. The bytes deleted may span more than one segment.

If all the bytes in any one segment are deleted, then the segment is deleted, and the data buffer that it referenced will be freed if no other zbuf segments reference it. No segment may survive with zero bytes referenced.

Deleting bytes out of the middle of a segment splits the segment into two. The first segment contains the portion of the data buffer before the deleted bytes, while the other segment contains the end portion that remains after deleting *len* bytes.

This routine returns the zbuf segment ID of the segment just after the deleted bytes. In the case where bytes are cut off the end of a zbuf, a value of **ZBUF_NONE** is returned.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. Likewise, the returned value is valid in the protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS The zbuf segment ID of the segment following the deleted bytes, or **NULL** if the operation fails.

SEE ALSO **zbufLib**

---

zbufDelete( )

NAME **zbufDelete( )** – delete a zbuf

SYNOPSIS

```
STATUS zbufDelete
    (
    ZBUF_ID zbufId            /* zbuf to be deleted */
    )
```

DESCRIPTION This routine deletes any zbuf segments in the specified zbuf, then deletes the zbuf ID itself. *zbufId* must not be used after this routine executes successfully.

For any data buffers that were not in use by any other zbuf, **zbufDelete( )** calls the associated free routine (callback).

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS **OK**, or **ERROR** if the zbuf cannot be deleted.

SEE ALSO **zbufLib**, **zbufCreate( )**, **zbufInsertBuf( )**

zbufDup( )

NAME **zbufDup( )** – duplicate a zbuf

SYNOPSIS

```
ZBUF_ID zbufDup
    (
    ZBUF_ID  zbufId,          /* zbuf to duplicate */
    ZBUF_SEG zbufSeg,         /* zbuf segment base for offset */
    int      offset,          /* relative byte offset */
    int      len              /* number of bytes to duplicate */
    )
```

DESCRIPTION This routine duplicates *len* bytes of *zbufId* starting at the specified byte location, and returns the zbuf ID of the newly created duplicate zbuf.

The starting location of duplication is specified by *zbufSeg* and *offset*. See the **zbufLib** manual page for more information on specifying a byte location within a zbuf. In particular, the first byte duplicated is the exact byte specified by *zbufSeg* and *offset*.

The number of bytes to duplicate is given by *len*. If this parameter is negative, or is larger than the number of bytes in the zbuf after the specified byte location, the rest of the zbuf is duplicated.

Duplication of zbuf data does not usually involve copying of the data. Instead, the zbuf segment pointer information is duplicated, while the data is not, which means that the data is shared among all zbuf segments that reference the data. See the **zbufLib** manual page for more information on copying and sharing zbuf data.

RETURNS The zbuf ID of a newly created duplicate zbuf, or **NULL** if the operation fails.

SEE ALSO **zbufLib**

zbufExtractCopy( )

NAME

zbufExtractCopy( ) – copy data from a zbuf to a buffer

SYNOPSIS

```
int zbufExtractCopy
    (
    ZBUF_ID  zbufId,          /* zbuf from which data is copied */
    ZBUF_SEG zbufSeg,         /* zbuf segment base for offset */
    int      offset,          /* relative byte offset */
    caddr_t  buf,             /* buffer into which data is copied */
    int      len              /* number of bytes to copy */
    )
```

DESCRIPTION

This routine copies *len* bytes of data from *zbufId* to the application buffer *buf*.

The starting location of the copy is specified by *zbufSeg* and *offset*. See the **zbufLib** manual page for more information on specifying a byte location within a zbuf. In particular, the first byte copied is the exact byte specified by *zbufSeg* and *offset*.

The number of bytes to copy is given by *len*. If this parameter is negative, or is larger than the number of bytes in the zbuf after the specified byte location, the rest of the zbuf is copied. The bytes copied may span more than one segment.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

The number of bytes copied from the zbuf to the buffer, or **ERROR** if the operation fails.

SEE ALSO

zbufLib

zbufInsert( )

NAME

zbufInsert( ) – insert a zbuf into another zbuf

SYNOPSIS

```
ZBUF_SEG zbufInsert
    (
    ZBUF_ID  zbufId1,         /* zbuf to insert zbufId2 into */
    ZBUF_SEG zbufSeg,         /* zbuf segment base for offset */
    int      offset,          /* relative byte offset */
    ZBUF_ID  zbufId2          /* zbuf to insert into zbufId1 */
    )
```

DESCRIPTION

This routine inserts all *zbufId2* zbuf segments into *zbufId1* at the specified byte location.

The location of insertion is specified by *zbufSeg* and *offset*. See the **zbufLib** manual page for more information on specifying a byte location within a zbuf. In particular, insertion within a zbuf occurs before the byte location specified by *zbufSeg* and *offset*. Additionally, *zbufSeg* and *offset* must be **NULL** and 0, respectively, when inserting into an empty zbuf.

After all the *zbufId2* segments are inserted into *zbufId1*, the zbuf ID *zbufId2* is deleted. *zbufId2* must not be used after this routine executes successfully.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. Likewise, the returned **ZBUF_SEG** is valid within the kernel protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

The zbuf segment ID for the first inserted segment, or **NULL** if the operation fails.

SEE ALSO

zbufLib

zbufInsertBuf( )

NAME **zbufInsertBuf( )** – create a zbuf segment from a buffer and insert into a zbuf

SYNOPSIS

```
ZBUF_SEG zbufInsertBuf
    (
    ZBUF_ID     zbufId,       /* zbuf in which buffer is inserted */
    ZBUF_SEG    zbufSeg,      /* zbuf segment base for offset */
    int         offset,       /* relative byte offset */
    caddr_t     buf,          /* application buffer for segment */
    int         len,          /* number of bytes to insert */
    VOIDFUNCPTR freeRtn,      /* free-routine callback */
    int         freeArg       /* argument to free routine */
    )
```

DESCRIPTION This routine creates a zbuf segment from the application buffer *buf* and inserts it at the specified byte location in *zbufId*.

The location of insertion is specified by *zbufSeg* and *offset*. See the **zbufLib** manual page for more information on specifying a byte location within a zbuf. In particular, insertion within a zbuf occurs before the byte location specified by *zbufSeg* and *offset*. Additionally, *zbufSeg* and *offset* must be **NULL** and 0, respectively, when inserting into an empty zbuf.

The parameter *freeRtn* specifies a free-routine callback that runs when the data buffer *buf* is no longer referenced by any zbuf segments. If *freeRtn* is **NULL**, the zbuf functions normally, except that the application is not notified when no more zbufs segments reference *buf*. The free-routine callback runs from the context of the task that last deletes reference to the buffer. Declare the *freeRtn* callback as follows (using whatever routine name suits your application):

```
void freeCallback
    (
    caddr_t     buf,    /* pointer to application buffer */
    int         freeArg /* argument to free routine */
    )
```

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS The zbuf segment ID of the inserted segment, or **NULL** if the operation fails.

SEE ALSO **zbufLib**

zbufInsertCopy( )

NAME **zbufInsertCopy( )** – copy buffer data into a zbuf

SYNOPSIS

```
ZBUF_SEG zbufInsertCopy
    (
    ZBUF_ID  zbufId,          /* zbuf into which data is copied */
    ZBUF_SEG zbufSeg,         /* zbuf segment base for offset */
    int      offset,          /* relative byte offset */
    caddr_t  buf,             /* buffer from which data is copied */
    int      len              /* number of bytes to copy */
    )
```

DESCRIPTION This routine copies *len* bytes of data from the application buffer *buf* and inserts it at the specified byte location in *zbufId*. The application buffer is in no way tied to the zbuf after this operation; a separate copy of the data is made.

The location of insertion is specified by *zbufSeg* and *offset*. See the **zbufLib** manual page for more information on specifying a byte location within a zbuf. In particular, insertion within a zbuf occurs before the byte location specified by *zbufSeg* and *offset*. Additionally, *zbufSeg* and *offset* must be **NULL** and 0, respectively, when inserting into an empty zbuf.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. Likewise, the returned value is valid in the protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS The zbuf segment ID of the first inserted segment, or **NULL** if the operation fails.

SEE ALSO **zbufLib**

zbufLength( )

NAME **zbufLength( )** – determine the length in bytes of a zbuf

SYNOPSIS

```
int zbufLength
    (
    ZBUF_ID zbufId            /* zbuf to determine length */
    )
```

DESCRIPTION This routine returns the number of bytes in the zbuf *zbufId*.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS The number of bytes in the zbuf, or **ERROR** if the operation fails.

SEE ALSO **zbufLib**

zbufSegData( )

NAME **zbufSegData( )** – determine the location of data in a zbuf segment

SYNOPSIS

```
caddr_t zbufSegData
    (
    ZBUF_ID  zbufId,          /* zbuf to examine */
    ZBUF_SEG zbufSeg          /* segment to get pointer to data */
    )
```

DESCRIPTION This routine returns the location of the first byte of data in the zbuf segment *zbufSeg*. If *zbufSeg* is **NULL**, the location of data in the first segment in *zbufId* is returned.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. Likewise, the returned value is valid in the protection domain only. This restriction does not apply under non-AE versions of VxWorks.

Z

RETURNS A pointer to the first byte of data in the specified zbuf segment, or **NULL** if the operation fails.

SEE ALSO **zbufLib**

zbufSegFind( )

NAME **zbufSegFind( )** – find the zbuf segment containing a specified byte location

SYNOPSIS

```
ZBUF_SEG zbufSegFind
    (
    ZBUF_ID  zbufId,           /* zbuf to examine */
    ZBUF_SEG zbufSeg,          /* zbuf segment base for pOffset */
    int *    pOffset           /* relative byte offset */
    )
```

DESCRIPTION This routine translates an address within a zbuf to its most local formulation. **zbufSegFind( )** locates the zbuf segment in *zbufId* that contains the byte location specified by *zbufSeg* and **pOffset*, then returns that zbuf segment, and writes in **pOffset* the new offset relative to the returned segment.

If the *zbufSeg*, **pOffset* pair specify a byte location past the end of the zbuf, or before the first byte in the zbuf, **zbufSegFind( )** returns **NULL**.

See the **zbufLib** manual page for a full discussion of addressing zbufs by segment and offset.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. Likewise, the returned value is valid in the protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS The zbuf segment ID of the segment containing the specified byte, or **NULL** if the operation fails.

SEE ALSO **zbufLib**

zbufSegLength( )

NAME **zbufSegLength( )** – determine the length of a zbuf segment

SYNOPSIS

```
int zbufSegLength
    (
    ZBUF_ID  zbufId,          /* zbuf to examine */
    ZBUF_SEG zbufSeg          /* segment to determine length of */
    )
```

DESCRIPTION This routine returns the number of bytes in the zbuf segment *zbufSeg*. If *zbufSeg* is **NULL**, the length of the first segment in *zbufId* is returned.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS The number of bytes in the specified zbuf segment, or **ERROR** if the operation fails.

SEE ALSO **zbufLib**

zbufSegNext( )

NAME **zbufSegNext( )** – get the next segment in a zbuf

SYNOPSIS

```
ZBUF_SEG zbufSegNext
    (
    ZBUF_ID  zbufId,          /* zbuf to examine */
    ZBUF_SEG zbufSeg          /* segment to get next segment */
    )
```

DESCRIPTION This routine finds the zbuf segment in *zbufId* that is just after the zbuf segment *zbufSeg*. If *zbufSeg* is **NULL**, the segment after the first segment in *zbufId* is returned. If *zbufSeg* is the last segment in *zbufId*, **NULL** is returned.

RETURNS The zbuf segment ID of the segment after *zbufSeg*, or **NULL** if the operation fails.

SEE ALSO **zbufLib**

Z

zbufSegPrev( )

NAME

zbufSegPrev( ) – get the previous segment in a zbuf

SYNOPSIS

```
ZBUF_SEG zbufSegPrev
    (
    ZBUF_ID  zbufId,           /* zbuf to examine */
    ZBUF_SEG zbufSeg           /* segment to get previous segment */
    )
```

DESCRIPTION

This routine finds the zbuf segment in *zbufId* that is just before the zbuf segment *zbufSeg*. If *zbufSeg* is **NULL**, or is the first segment in *zbufId*, **NULL** is returned.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. Likewise, the returned value is valid in the protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

The zbuf segment ID of the segment before *zbufSeg*, or **NULL** if the operation fails.

SEE ALSO

zbufLib

zbufSockBufSend( )

NAME

zbufSockBufSend( ) – create a zbuf from user data and send it to a TCP socket

SYNOPSIS

```
int zbufSockBufSend
    (
    int         s,             /* socket to send to */
    char *      buf,           /* pointer to data buffer */
    int         bufLen,        /* number of bytes to send */
    VOIDFUNCPTR freeRtn,       /* free routine callback */
    int         freeArg,       /* argument to free routine */
    int         flags          /* flags to underlying protocols */
    )
```

DESCRIPTION

This routine creates a zbuf from the user buffer *buf*, and transmits it to a previously established connection-based (stream) socket.

The user-provided free routine callback at *freeRtn* is called when *buf* is no longer in use by the TCP/IP network stack. Applications can exploit this callback to receive notification that *buf* is free. If *freeRtn* is **NULL**, the routine functions normally, except that the application has no way of being notified when *buf* is released by the network stack. The free routine runs in the context of the task that last references the buffer. This is typically either the context of **tNetTask**, or the context of the caller's task. Declare *freeRtn* as follows (using whatever name is convenient):

```
void freeCallback
    (
    caddr_t     buf,    /* pointer to user buffer */
    int         freeArg /* user-provided argument to free routine */
    )
```

You may OR the following values into the *flags* parameter with this operation:

MSG_OOB (0x1)
Out-of-band data.

MSG_DONTROUTE (0x4)
Send without using routing tables.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS The number of bytes sent, or **ERROR** if the call fails.

SEE ALSO **zbufSockLib**, **zbufSockSend( )**, **send( )**

---

zbufSockBufSendto( )

NAME **zbufSockBufSendto( )** – create a zbuf from a user message and send it to a UDP socket

SYNOPSIS

```
int zbufSockBufSendto
    (
    int                 s,       /* socket to send to */
    char *              buf,     /* pointer to data buffer */
    int                 bufLen,  /* number of bytes to send */
    VOIDFUNCPTR         freeRtn, /* free routine callback */
    int                 freeArg, /* argument to free routine */
    int                 flags,   /* flags to underlying protocols */
```

Z

```
    struct sockaddr * to,      /* recipient's address */
    int               tolen    /* length of to socket addr */
    )
```

DESCRIPTION

This routine creates a zbuf from the user buffer *buf*, and sends it to the datagram socket named by *to*. The socket *s* is the sending socket.

The user-provided free routine callback at *freeRtn* is called when *buf* is no longer in use by the UDP/IP network stack. Applications can exploit this callback to receive notification that *buf* is free. If *freeRtn* is **NULL**, the routine functions normally, except that the application has no way of being notified when *buf* is released by the network stack. The free routine runs in the context of the task that last references the buffer. This is typically either **tNetTask** context, or the caller's task context. Declare *freeRtn* as follows (using whatever name is convenient):

```
void freeCallback
    (
    caddr_t     buf,    /* pointer to user buffer */
    int         freeArg /* user-provided argument to free routine */
    )
```

You may OR the following values into the *flags* parameter with this operation:

MSG_OOB (0x1)
Out-of-band data.

MSG_DONTROUTE (0x4)
Send without using routing tables.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

The number of bytes sent, or **ERROR** if the call fails.

SEE ALSO

zbufSockLib, **zbufSockSendto( )**, **sendto( )**

zbufSockLibInit( )

NAME **zbufSockLibInit( )** – initialize the zbuf socket interface library

SYNOPSIS

```
STATUS zbufSockLibInit (void)
```

DESCRIPTION This routine initializes the zbuf socket interface library. It must be called before any zbuf socket routines are used. It is called automatically when **INCLUDE_ZBUF_SOCK** is defined.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS **OK**, or **ERROR** if the zbuf socket interface could not be initialized.

SEE ALSO **zbufSockLib**

zbufSockRecv( )

NAME **zbufSockRecv( )** – receive data in a zbuf from a TCP socket

SYNOPSIS

```
ZBUF_ID zbufSockRecv
    (
    int   s,                  /* socket to receive data from */
    int   flags,              /* flags to underlying protocols */
    int * pLen                /* number of bytes requested/returned */
    )
```

DESCRIPTION This routine receives data from a connection-based (stream) socket, and returns the data to the user in a newly created zbuf.

The *pLen* parameter indicates the number of bytes requested by the caller. If the operation is successful, the number of bytes received is copied to *pLen*.

You may OR the following values into the *flags* parameter with this operation:

MSG_OOB (0x1)
Out-of-band data.

Z

MSG_PEEK (0x2)
Return data without removing it from socket.

Once the user application is finished with the zbuf, **zbufDelete( )** should be called to return the zbuf memory buffer to the VxWorks network stack.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

The zbuf ID of a newly created zbuf containing the received data, or **NULL** if the operation fails.

SEE ALSO

zbufSockLib, **recv( )**

zbufSockRecvfrom( )

NAME

zbufSockRecvfrom( ) – receive a message in a zbuf from a UDP socket

SYNOPSIS

```
ZBUF_ID zbufSockRecvfrom
    (
    int                s,        /* socket to receive from */
    int                flags,    /* flags to underlying protocols */
    int *              pLen,     /* number of bytes requested/returned */
    struct sockaddr * from,     /* where to copy sender's addr */
    int *              pFromLen /* value/result length of from */
    )
```

DESCRIPTION

This routine receives a message from a datagram socket, and returns the message to the user in a newly created zbuf.

The message is received regardless of whether the socket is connected. If *from* is nonzero, the address of the sender's socket is copied to it. Initialize the value-result parameter *pFromLen* to the size of the *from* buffer. On return, *pFromLen* contains the actual size of the address stored in *from*.

The *pLen* parameter indicates the number of bytes requested by the caller. If the operation is successful, the number of bytes received is copied to *pLen*.

You may OR the following values into the *flags* parameter with this operation:

MSG_OOB (0x1)
Out-of-band data.

MSG_PEEK (0x2)
Return data without removing it from socket.

Once the user application is finished with the zbuf, **zbufDelete( )** should be called to return the zbuf memory buffer to the VxWorks network stack.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

The zbuf ID of a newly created zbuf containing the received message, or **NULL** if the operation fails.

SEE ALSO

zbufSockLib

---

zbufSockSend( )

NAME

zbufSockSend( ) – send zbuf data to a TCP socket

SYNOPSIS

```
int zbufSockSend
    (
    int     s,                /* socket to send to */
    ZBUF_ID zbufId,           /* zbuf to transmit */
    int     zbufLen,          /* length of entire zbuf */
    int     flags             /* flags to underlying protocols */
    )
```

DESCRIPTION

This routine transmits all of the data in *zbufId* to a previously established connection-based (stream) socket.

The *zbufLen* parameter is used only for determining the amount of space needed from the socket write buffer. *zbufLen* has no effect on how many bytes are sent; the entire zbuf is always transmitted. If the length of *zbufId* is not known, the caller must first determine it by calling **zbufLength( )**.

This routine transfers ownership of the zbuf from the user application to the VxWorks network stack. The zbuf ID, *zbufId*, is deleted by this routine, and should not be used after the routine is called, even if an **ERROR** status is returned. (Exceptions: when the routine fails because the zbuf socket interface library was not initialized or an invalid zbuf ID was

Z

passed in, in which case there is no zbuf to delete. Moreover, if the call fails during a non-blocking I/O socket write with an **errno** of **EWOULDBLOCK**, then *zbufId* is not deleted; thus the caller may send it again at a later time.)

You may OR the following values into the *flags* parameter with this operation:

MSG_OOB (0x1)
Out-of-band data.

MSG_DONTROUTE (0x4)
Send without using routing tables.

VXWORKS AE PROTECTION DOMAINS
Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS The number of bytes sent, or **ERROR** if the call fails.

SEE ALSO **zbufSockLib**, **zbufLength( )**, **zbufSockBufSend( )**, **send( )**

zbufSockSendto( )

NAME **zbufSockSendto( )** – send a zbuf message to a UDP socket

SYNOPSIS

```
int zbufSockSendto
    (
    int               s,       /* socket to send to */
    ZBUF_ID           zbufId,  /* zbuf to transmit */
    int               zbufLen, /* length of entire zbuf */
    int               flags,   /* flags to underlying protocols */
    struct sockaddr * to,      /* recipient's address */
    int               tolen    /* length of to socket addr */
    )
```

DESCRIPTION This routine sends the entire message in *zbufId* to the datagram socket named by *to*. The socket *s* is the sending socket.

The *zbufLen* parameter is used only for determining the amount of space needed from the socket write buffer. *zbufLen* has no effect on how many bytes are sent; the entire zbuf is always transmitted. If the length of *zbufId* is not known, the caller must first determine it by calling **zbufLength( )**.

This routine transfers ownership of the zbuf from the user application to the VxWorks network stack. The zbuf ID *zbufId* is deleted by this routine, and should not be used after the routine is called, even if an **ERROR** status is returned. (Exceptions: when the routine fails because the zbuf socket interface library was not initialized or an invalid zbuf ID was passed in, in which case there is no zbuf to delete. Moreover, if the call fails during a non-blocking I/O socket write with an **errno** of **EWOULDBLOCK**, then *zbufId* is not deleted; thus the caller may send it again at a later time.)

You may OR the following values into the *flags* parameter with this operation:

MSG_OOB (0x1)
Out-of-band data.

MSG_DONTROUTE (0x4)
Send without using routing tables.

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

The number of bytes sent, or **ERROR** if the call fails.

SEE ALSO

zbufSockLib, **zbufLength( )**, **zbufSockBufSendto( )**, **sendto( )**

---

zbufSplit( )

NAME

zbufSplit( ) – split a zbuf into two separate zbufs

SYNOPSIS

```
ZBUF_ID zbufSplit
    (
    ZBUF_ID  zbufId,          /* zbuf to split into two */
    ZBUF_SEG zbufSeg,         /* zbuf segment base for offset */
    int      offset           /* relative byte offset */
    )
```

DESCRIPTION

This routine splits *zbufId* into two separate zbufs at the specified byte location. The first portion remains in *zbufId*, while the end portion is returned in a newly created zbuf.

The location of the split is specified by *zbufSeg* and *offset*. See the **zbufLib** manual page for more information on specifying a byte location within a zbuf. In particular, after the split operation, the first byte of the returned zbuf is the exact byte specified by *zbufSeg* and *offset*.

Z

VXWORKS AE PROTECTION DOMAINS

Under VxWorks AE, you can call this function from within the kernel protection domain only. In addition, all arguments to this function can reference only that data which is valid in the kernel protection domain. Likewise, the returned value is valid in the protection domain only. This restriction does not apply under non-AE versions of VxWorks.

RETURNS

The zbuf ID of a newly created zbuf containing the end portion of *zbufId*, or **NULL** if the operation fails.

SEE ALSO

zbufLib

Keyword Index

IX

IX

IX

IX

IX

IX

IX

IX

IX

IX

IX

IX

IX