VxWorks®

Reference Manual

5.3.1

Edition 1

WindRiver
SYSTEMS

Copyright © 1984 – 1998 Wind River Systems, Inc.

ALL RIGHTS RESERVED. No part of this publication may be copied in any form, by photocopy, microfilm, retrieval system, or by any other means now known or hereafter invented without the prior written permission of Wind River Systems, Inc.

VxWorks, Wind River Systems, the Wind River Systems logo, and *wind* are registered trademarks of Wind River Systems, Inc. CrossWind, IxWorks, Tornado, VxMP, VxSim, VxVMI, WindC++, WindConfig, Wind Foundation Classes, WindNet, WindPower, WindSh, and WindView are trademarks of Wind River Systems, Inc.

All other trademarks used in this document are the property of their respective owners.

Corporate Headquarters
Wind River Systems, Inc.
1010 Atlantic Avenue
Alameda, CA 94501-1153
USA

toll free (US): 800/545-WIND
telephone: 510/748-4100
facsimile: 510/814-2010

Europe
Wind River Systems, S.A.R.L.
19, Avenue de Norvège
Immeuble B4, Bâtiment 3
Z.A. de Courtaboeuf 1
91953 Les Ulis Cédex
FRANCE

telephone: 33-1-60-92-63-00
facsimile: 33-1-60-92-63-15

Japan
Wind River Systems Japan
Pola Ebisu Bldg. 11F
3-9-19 Higashi
Shibuya-ku
Tokyo 150
JAPAN

telephone: 81-3-5467-5900
facsimile: 81-3-5467-5877

CUSTOMER SUPPORT

	Telephone	**E-mail**	**Fax**
Corporate:	800/872-4977 toll free, U.S. & Canada 510/748-4100 direct	support@wrs.com	510/814-2164
Europe:	33-1-69-07-78-78	support@wrsec.fr	33-1-69-07-08-26
Japan:	011-81-3-5467-5900	support@kk.wrs.com	011-81-3-5467-5877

If you purchased your Wind River Systems product from a distributor, please contact your distributor to determine how to reach your technical support organization.

Please provide your license number when contacting Customer Support.

VxWorks Reference Manual, 5.3.1
Edition 1
19 Apr 98
Part #: DOC-12068-ZD-00

Contents

1 Libraries

aioPxLib

NAME

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

SYNOPSIS

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_cancel( ) – cancel an asynchronous I/O request (POSIX)
aio_fsync( ) – asynchronous file synchronization (POSIX)
aio_error( ) – retrieve error status of asynchronous I/O operation (POSIX)
aio_return( ) – retrieve return status of asynchronous I/O operation (POSIX)

```
STATUS aioPxLibInit
    (int lioMax)

int aio_read
    (struct aiocb * pAiocb)

int aio_write
    (struct aiocb * pAiocb)

int lio_listio
    (int mode, struct aiocb * list[], int nEnt, struct sigevent * pSig)

int aio_suspend
    (const struct aiocb * list[], int nEnt, const struct timespec * timeout)

int aio_cancel
    (int fildes, struct aiocb * pAiocb)

int aio_fsync
    (int op, struct aiocb * pAiocb)

int aio_error
    (const struct aiocb * pAiocb)

size_t aio_return
    (struct aiocb * pAiocb)
```

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, *aioShow***( )** (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);
    }
```

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

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

```
STATUS aioShow
    (int drvNum)
```

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

aioSysDrv

NAME **aioSysDrv** – AIO system driver

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

```
STATUS aioSysInit
    (int numTasks, int taskPrio, int taskStackSize)
```

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

SYNOPSIS *assert***( )** – put diagnostics into programs (ANSI)

```
void assert
    (int a)
```

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

SYNOPSIS

*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)

```
int isalnum
     (int c)

int isalpha
     (int c)

int iscntrl
     (int c)

int isdigit
     (int c)

int isgraph
     (int c)

int islower
     (int c)

int isprint
     (int c)

int ispunct
     (int c)

int isspace
     (int c)

int isupper
     (int c)
```

```
int isxdigit
    (int c)

int tolower
    (int c)

int toupper
    (int c)
```

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.

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

SYNOPSIS

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

```
struct lconv *localeconv
    (void)

char *setlocale
    (int category, const char *localeName)
```

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

SYNOPSIS

*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)

```
double asin
     (double x)

double acos
     (double x)

double atan
     (double x)

double atan2
     (double y, double x)

double ceil
     (double v)

double cosh
     (double x)

double exp
     (double x)
```

```
double fabs
    (double v)

double floor
    (double v)

double fmod
    (double x, double y)

double frexp
    (double value, int *pexp)

double ldexp
    (double v, int xexp)

double log
    (double x)

double log10
    (double x)

double modf
    (double value, double *pIntPart)

double pow
    (double x, double y)

double sin
    (double x)

double cos
    (double x)

double sinh
    (double x)

double sqrt
    (double x)

double tan
    (double x)

double tanh
    (double x)
```

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

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

```
int setjmp
     (jmp_buf env)

void longjmp
     (jmp_buf env, int val)
```

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

SYNOPSIS

*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

```
void va_start
     (ap, parmN)

void va_arg
     (ap, type)

void va_end
     (ap)
```

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

SYNOPSIS

*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)

```
void clearerr
    (FILE * fp)

int fclose
    (FILE * fp)

FILE * fdopen
    (int fd, const char * mode)

int feof
    (FILE * fp)

int ferror
    (FILE * fp)

int fflush
    (FILE * fp)

int fgetc
    (FILE * fp)

int fgetpos
    (FILE * fp, fpos_t * pos)

char * fgets
    (char * buf, size_t n, FILE * fp)

int fileno
    (FILE * fp)

FILE * fopen
    (const char * file, const char * mode)

int fprintf
    (FILE * fp, const char * fmt, ...)

int fputc
    (int c, FILE * fp)

int fputs
    (const char * s, FILE * fp)

int fread
    (void * buf, size_t size, size_t count, FILE * fp)

FILE * freopen
    (const char * file, const char * mode, FILE * fp)
```

```
int fscanf
     (FILE * fp, char const * fmt, ...)

int fseek
     (FILE * fp, long offset, int whence)

int fsetpos
     (FILE * iop, const fpos_t * pos)

long ftell
     (FILE * fp)

int fwrite
     (const void * buf, size_t size, size_t count, FILE * fp)

int getc
     (FILE * fp)

int getchar
     (void)

char * gets
     (char * buf)

int getw
     (FILE * fp)

void perror
     (const char * __s)

int putc
     (int c, FILE * fp)

int putchar
     (int c)

int puts
     (char const * s)

int putw
     (int w, FILE * fp)

void rewind
     (FILE * fp)

int scanf
     (char const * fmt, ...)

void setbuf
     (FILE * fp, char * buf)

void setbuffer
     (FILE * fp, char * buf, int size)
```

```
int setlinebuf
    (FILE * fp)

int setvbuf
    (FILE * fp, char * buf, int mode, size_t size)

STATUS stdioInit
    (void)

FILE * stdioFp
    (int stdFd)

STATUS stdioShowInit
    (void)

STATUS stdioShow
    (FILE * fp, int level)

FILE * tmpfile
    (void)

char * tmpnam
    (char * s)

int ungetc
    (int c, FILE * fp)

int vfprintf
    (FILE * fp, const char * fmt, va_list vaList)
```

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 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 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

SYNOPSIS

*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)

```
void abort
    (void)

int abs
    (int i)

int atexit
    (void (*__func)(void))

double atof
    (const char * s)

int atoi
    (const char * s)

long atol
    (const register char * s)

void * bsearch
    (const void * key, const void * base0, size_t nmemb, size_t size,
    int (*compar) (const void *, const void *))

div_t div
    (int numer, int denom)

void div_r
    (int numer, int denom, div_t * divStructPtr)

long labs
    (long i)

ldiv_t ldiv
    (long numer, long denom)

void ldiv_r
    (long numer, long denom, ldiv_t * divStructPtr)

int mblen
    (const char * s, size_t n)

int mbtowc
    (wchar_t * pwc, const char * s, size_t n)

int wctomb
    (char * s, wchar_t wchar)

size_t mbstowcs
    (wchar_t * pwcs, const char * s, size_t n)
```

```
size_t wcstombs
     (char * s, const wchar_t * pwcs, size_t n)

void qsort
     (void * bot, size_t nmemb, size_t size, int (*compar) (const void *,
     const void *))

int rand
     (void)

void * srand
     (uint_t seed)

double strtod
     (const char * s, char ** endptr)

long strtol
     (const char * nptr, char ** endptr, int base)

ulong_t strtoul
     (const char * nptr, char ** endptr, int base)

int system
     (const char * string)
```

DESCRIPTION

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

SYNOPSIS

*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)

```
void * memchr
    (const void * m, int c, size_t n)

int memcmp
    (const void * s1, const void * s2, size_t n)

void * memcpy
    (void * destination, const void * source, size_t size)

void * memmove
    (void * destination, const void * source, size_t size)

void * memset
    (void * m, int c, size_t size)

char * strcat
    (char * destination, const char * append)
```

```
char * strchr
     (const char * s, int c)

int strcmp
     (const char * s1, const char * s2)

int strcoll
     (const char * s1, const char * s2)

char * strcpy
     (char * s1, const char * s2)

size_t strcspn
     (const char * s1, const char * s2)

STATUS strerror_r
     (int errcode, char * buffer)

char * strerror
     (int errcode)

size_t strlen
     (const char * s)

char * strncat
     (char * dst, const char * src, size_t n)

int strncmp
     (const char * s1, const char * s2, size_t n)

char *strncpy
     (char * s1, const char *s2, size_t n)

char * strpbrk
     (const char * s1, const char * s2)

char * strrchr
     (const char * s, int c)

size_t strspn
     (const char * s, const char * sep)

char * strstr
     (const char * s, const char * find)

char * strtok
     (char * string, const char * separator)

char * strtok_r
     (char * string, const char * separators, char ** ppLast)

size_t strxfrm
     (char * s1, const char * s2, size_t n)
```

DESCRIPTION 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

ansiTime

NAME **ansiTime** – ANSI **time** documentation

SYNOPSIS

*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)

```
char * asctime
     (const struct tm *timeptr)

int asctime_r
     (const struct tm *timeptr, char * asctimeBuf, size_t * buflen)

clock_t clock
     (void)

char * ctime
     (const time_t *timer)

char * ctime_r
     (const time_t * timer, char * asctimeBuf, size_t * buflen)

double difftime
     (time_t time1, time_t time0)
```

```
struct tm *gmtime
     (const time_t *timer)

int gmtime_r
     (const time_t *timer, struct tm * timeBuffer)

struct tm *localtime
     (const time_t * timer)

int localtime_r
     (const time_t * timer, struct tm * timeBuffer)

time_t mktime
     (struct tm * timeptr)

size_t strftime
     (char * s, size_t n, const char * format, const struct tm * tptr)

time_t time
     (time_t *timer)
```

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, 61]
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
```

INCLUDE FILES **time.h**

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

arpLib

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

SYNOPSIS *arpAdd***( )** – add an entry to the system ARP table
*arpDelete***( )** – delete an entry from the system ARP table
*arpFlush***( )** – flush all entries in the system ARP table

```
STATUS arpAdd
    (char * host, char * eaddr, int flags)

STATUS arpDelete
    (char * host)

void arpFlush
    (void)
```

DESCRIPTION This library provides functionality for manipulating the system Address Resolution Protocol (ARP) table (cache). ARP is used by the networking modules to map dynamically between Internet Protocol (IP) addresses and physical hardware (Ethernet) addresses. Once these addresses get resolved, they are stored in the system ARP table.

Two routines allow the caller to modify this ARP table manually: *arpAdd***( )** and *arpDelete***( )**. Use *arpAdd***( )** to add new or modify existing entries in the ARP table. Call

*arpDelete***( )** to delete entries from the ARP table. Call *arpShow***( )** to show current entries in the ARP table.

INCLUDE FILES **arpLib.h**

SEE ALSO **inetLib**, **routeLib**, **etherLib**, **netShow**, *VxWorks Programmer's Guide: Network*

ataDrv

NAME **ataDrv** – ATA/IDE (LOCAL and PCMCIA) disk device driver

SYNOPSIS *ataDrv***( )** – initialize the ATA driver
*ataDevCreate***( )** – create a device for a ATA/IDE disk
*ataRawio***( )** – do raw I/O access

```
STATUS ataDrv
    (int ctrl, int drives, int vector, int level, BOOL configType,
    int semTimeout, int wdgTimeout)

BLK_DEV *ataDevCreate
    (int ctrl, int drive, int nBlocks, int blkOffset)

STATUS ataRawio
    (int ctrl, int drive, ATA_RAW *pAtaRaw)
```

DESCRIPTION This is a driver for the ATA/IDE (LOCAL and PCMCIA) device used on the IBM PC.

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: *ataDrv***( )** to initialize the driver and *ataDevCreate***( )** to create devices.

Before the driver can be used, it must be initialized by calling *ataDrv***( )**. This routine must be called exactly once, before any reads, writes, or calls to *ataDevCreate***( )**. Normally, it is called from *usrRoot***( )** in **usrConfig.c**.

The routine *ataRawio***( )** supports physical I/O access. The first argument is a drive number, 0 or 1; the second argument is a pointer to an **ATA_RAW** structure.

NOTE Format is not supported, because ATA/IDE disks are already formatted, and bad sectors are mapped.

PARAMETERS

The *ataDrv*( ) function requires a configuration flag as a parameter. The configuration flag is one of the following:

Transfer mode

ATA_PIO_DEF_0	PIO default mode
ATA_PIO_DEF_1	PIO default mode, no IORDY
ATA_PIO_0	PIO mode 0
ATA_PIO_1	PIO mode 1
ATA_PIO_2	PIO mode 2
ATA_PIO_3	PIO mode 3
ATA_PIO_4	PIO mode 4
ATA_PIO_AUTO	PIO max supported mode
ATA_DMA_0	DMA mode 0
ATA_DMA_1	DMA mode 1
ATA_DMA_2	DMA mode 2
ATA_DMA_AUTO	DMA max supported mode

Transfer bits

ATA_BITS_16	RW bits size, 16 bits
ATA_BITS_32	RW bits size, 32 bits

Transfer unit

ATA_PIO_SINGLE	RW PIO single sector
ATA_PIO_MULTI	RW PIO multi sector
ATA_DMA_SINGLE	RW DMA single word
ATA_DMA_MULTI	RW DMA multi word

Geometry parameters

ATA_GEO_FORCE	set geometry in the table
ATA_GEO_PHYSICAL	set physical geometry
ATA_GEO_CURRENT	set current geometry

DMA transfer is not supported in this release. If **ATA_PIO_AUTO** or **ATA_DMA_AUTO** is specified, the driver automatically chooses the maximum mode supported by the device. If **ATA_PIO_MULTI** or **ATA_DMA_MULTI** is specified, and the device does not support it, the driver automatically chooses single sector or word mode. If **ATA_BITS_32** is specified, the driver uses 32-bit transfer mode regardless of the capability of the drive.

If **ATA_GEO_PHYSICAL** is specified, the driver uses the physical geometry parameters stored in the drive. If **ATA_GEO_CURRENT** is specified, the driver uses current geometry parameters initialized by BIOS. If **ATA_GEO_FORCE** is specified, the driver uses geometry parameters stored in **sysLib.c**.

The geometry parameters are stored in the structure table **ataTypes[]** in **sysLib.c**. That table has two entries, the first for drive 0, the second for drive 1. The members of the structure are:

```
int cylinders;              /* number of cylinders */
int heads;                  /* number of heads */
int sectors;                /* number of sectors per track */
int bytes;                  /* number of bytes per sector */
int precomp;                /* precompensation cylinder */
```

This driver does not access the PCI-chip-set IDE interface, but rather takes advantage of BIOS initialization. Thus, the BIOS setting should match the modes specified by the configuration flag.

SEE ALSO *VxWorks Programmer's Guide: I/O System*

ataShow

NAME **ataShow** – ATA / IDE (LOCAL and PCMCIA) disk device driver show routine

SYNOPSIS *ataShowInit***( )** – initialize the ATA / IDE disk driver show routine
*ataShow***( )** – show the ATA / IDE disk parameters

```
void ataShowInit
    (void)

STATUS ataShow
    (int ctrl, int drive)
```

DESCRIPTION This library contains a driver show routine for the ATA / IDE (PCMCIA and LOCAL) devices supported on the IBM PC.

autopushLib

NAME **autopushLib** – WindNet STREAMS autopush facility (STREAMS Opt.)

SYNOPSIS *autopushAdd***( )** – add a list of automatically pushed STREAMS modules
*autopushDelete***( )** – delete autopush information for a device
*autopushGet***( )** – get autopush information for a device

```
void autopushAdd
    (char * arg)
```

```
void autopushDelete
    (char *deviceName)

void autopushGet
    (char *deviceName)
```

DESCRIPTION This library consists of routines to support the autopush facility in VxWorks. Autopush is an SVR4 STREAMS feature that allows users to specify module names which are to be pushed onto a device when the device is opened.

bALib

NAME **bALib** – buffer manipulation library SPARC assembly language routines

SYNOPSIS *bzeroDoubles***( )** – zero out a buffer eight bytes at a time (SPARC)
*bfillDoubles***( )** – fill a buffer with a specified eight-byte pattern (SPARC)
*bcopyDoubles***( )** – copy one buffer to another eight bytes at a time (SPARC)

```
STATUS bzeroDoubles
    (void * buffer, int nbytes)

STATUS bfillDoubles
    (void * buffer, int nbytes, ULONG bits_63to32, ULONG bits_31to0)

STATUS bcopyDoubles
    (void * source, void * destination, int ndoubles)
```

DESCRIPTION This library contains routines to manipulate buffers, which are simply variable length byte arrays. These routines are highly optimized loops.

All address pointers must be properly aligned for 8-byte moves. Note that buffer lengths are specified in terms of bytes or doubles. Since this is meant to be a high-performance operation, the minimum number of bytes is 256.

NOTE None of the buffer routines have been hand-coded in assembly. These are additional routines that exploit the SPARC's LDD and STD instructions.

SEE ALSO **bLib**, **ansiString**

bLib

NAME

bLib – buffer manipulation library

SYNOPSIS

*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

```
int bcmp
     (char *buf1, char *buf2, int nbytes)

void binvert
     (char *buf, int nbytes)

void bswap
     (char *buf1, char *buf2, int nbytes)

void swab
     (char *source, char *destination, int nbytes)

void uswab
     (char *source, char *destination, int nbytes)

void bzero
     (char *buffer, int nbytes)

void bcopy
     (const char *source, char *destination, int nbytes)

void bcopyBytes
     (char *source, char *destination, int nbytes)

void bcopyWords
     (char *source, char *destination, int nwords)

void bcopyLongs
     (char *source, char *destination, int nlongs)
```

```
void bfill
    (char *buf, int nbytes, int ch)

void bfillBytes
    (char *buf, int nbytes, int ch)

char *index
    (const char *s, int c)

char *rindex
    (const char *s, int c)
```

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

SYNOPSIS 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

SYNOPSIS

*romStart***( )** – generic ROM initialization

```
void romStart
     (int startType)
```

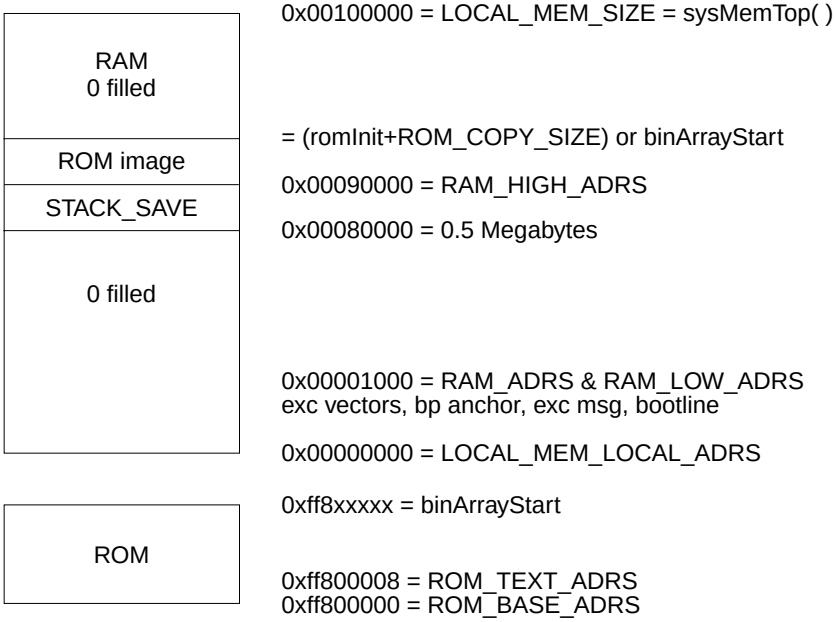
DESCRIPTION

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.

bootLib

NAME **bootLib** – boot ROM subroutine library

SYNOPSIS
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
bootNetmaskExtract( ) – extract the net mask field from an Internet address
bootBpAnchorExtract( ) – extract a backplane address from a device field

```
char *bootStringToStruct
    (char *bootString, BOOT_PARAMS *pBootParams)

STATUS bootStructToString
    (char *paramString, BOOT_PARAMS *pBootParams)

void bootParamsShow
    (char *paramString)

void bootParamsPrompt
    (char *string)

STATUS bootNetmaskExtract
    (char *string, int *pNetmask)

STATUS bootBpAnchorExtract
    (char *string, char **pAnchorAdrs)
```

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

where:

bootdev
: the boot device (e.g., “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***( )**).

procnum
: the processor number on the backplane (0..n).

hostname
: the name of the boot host.

filename
: the file to be booted.

e the Internet address of the Ethernet interface. This field can have an optional subnet mask of the form *inet_adrs*:*subnet_mask* (see *bootNetmaskExtract***( )**).

b the Internet address of the backplane interface. This field can have an optional subnet mask 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 – BOOTP client library

SYNOPSIS

*bootpParamsGet***( )** – retrieve boot parameters via BOOTP
*bootpMsgSend***( )** – send a BOOTP request message

```
STATUS bootpParamsGet
    (char * ifName, int port, char * pInetAddr, char * pHostAddr,
    char * pBootFile, int * pSizeFile, int * pSubnet, char * pGateway,
    u_int timeOut)

STATUS bootpMsgSend
    (char * ifName, struct in_addr * pIpDest, int port,
    BOOTP_MSG * pBootpMsg, u_int timeOut)
```

DESCRIPTION

This library implements the client side of the Bootstrap Protocol (BOOTP). BOOTP allows a booting target to configure itself dynamically by obtaining its IP address, the boot-file name, and the boot host's IP address over the network, instead of the more traditional way of using the information encoded in system non-volatile RAM or ROM. BOOTP simply retrieves the boot parameters. The actual transfer of the boot image is done by a file transfer program, such as TFTP, FTP, or RSH.

HIGH-LEVEL INTERFACE

A VxWorks target can use one of two interface levels in **bootpLib** to obtain its boot parameters. The *bootpParamsGet***( )** routine provides the highest level. It obtains the boot file, the Internet address and the host Internet address. It also obtains the subnet mask, if the BOOTP server supports the extensions described in RFC 1048, and the mask is specified in the database.

LOW-LEVEL INTERFACE

The *bootpMsgSend***( )** routine provides a lower-level interface, accepting and returning a BOOTP message as a parameter. This interface is more flexiblile because it gives the caller direct access to the fields in the BOOTP request/reply messages. For example, if it is desirable to obtain vendor-specific options, such as those described in RFC 1048, the caller can retrieve them from the vendor-specific field in the BOOTP message after using *bootpMsgSend***( )**.

EXAMPLE

The following code is an example of how to use the **bootpLib** high-level interface:

```
char        clntAddr [INET_ADDR_LEN];
char        bootServer [INET_ADDR_LEN];
char        bootFile [SIZE_FILE];
int         fileSize;
int         subnetMask;
```

```
char        gateway [INET_ADDR_LEN];
fileSize    = SIZE_FILE;
subnetMask  = 0;
bzero (clntAddr, INET_ADDR_LEN);
bzero (bootServer, INET_ADDR_LEN);
bzero (bootFile, SIZE_FILE);
if (bootpParamsGet ("ln0", 0, clntAddr, bootServer, bootFile,
                    &fileSize, &subnetMask, &gateway, 0) == ERROR)
    return (ERROR);
```

NOTE

Certain targets (typically those with no NV-RAM) concoct their Ethernet address based on the target's IP address. These targets must know their IP address at boot time in order to boot over the network.

BOOTP is not supported currently over the following network interfaces: **if_sl** (SLIP) and **if_ie** (Sun IE driver).

INCLUDE FILES **bootpLib.h**

SEE ALSO **bootLib**, RFC 951, RFC 1048, *VxWorks Programmer's Guide: Network*

cacheArchLib

NAME **cacheArchLib** – 68K cache management library

SYNOPSIS

*cacheArchLibInit***( )** – initialize the 68K cache library
*cacheArchClearEntry***( )** – clear an entry from a 68K cache
*cacheStoreBufEnable***( )** – enable the store buffer (MC68060 only)
*cacheStoreBufDisable***( )** – disable the store buffer (MC68060 only)

```
STATUS cacheArchLibInit
    (CACHE_MODE instMode, CACHE_MODE dataMode)

STATUS cacheArchClearEntry
    (CACHE_TYPE cache, void * address)

void cacheStoreBufEnable
    (void)

void cacheStoreBufDisable
    (void)
```

DESCRIPTION This library contains architecture-specific cache library functions for the Motorola 68K family instruction and data caches. The various members of the 68K family of processors

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. There is no data cache on the 68020; however, data cache operations return OK.

INCLUDE FILES **cacheLib.h**

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

cacheCy604Lib

NAME **cacheCy604Lib** – Cypress CY7C604/605 SPARC cache management library

SYNOPSIS *cacheCy604LibInit***( )** – initialize the Cypress CY7C604 cache library
*cacheCy604ClearLine***( )** – clear a line from a CY7C604 cache
*cacheCy604ClearPage***( )** – clear a page from a CY7C604 cache
*cacheCy604ClearSegment***( )** – clear a segment from a CY7C604 cache
*cacheCy604ClearRegion***( )** – clear a region from a CY7C604 cache

```
STATUS cacheCy604LibInit
    (CACHE_MODE instMode, CACHE_MODE dataMode)

STATUS cacheCy604ClearLine
    (CACHE_TYPE cache, void * address)

STATUS cacheCy604ClearPage
    (CACHE_TYPE cache, void * address)

STATUS cacheCy604ClearSegment
    (CACHE_TYPE cache, void * address)

STATUS cacheCy604ClearRegion
    (CACHE_TYPE cache, void * address)
```

DESCRIPTION This library contains architecture-specific cache library functions for the Cypress CY7C604 architecture. There is a 64-Kbyte mixed instruction and data cache that operates in write-through or copyback mode. Each cache line contains 32 bytes. Cache tag operations are performed with “line,” “page,” “segment,” or “region” 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**

cacheI960CxALib

NAME **cacheI960CxALib** – I960Cx cache management assembly routines

SYNOPSIS
*cacheI960CxICDisable***( )** – disable the I960Cx instruction cache (i960)
*cacheI960CxICEnable***( )** – enable the I960Cx instruction cache (i960)
*cacheI960CxICInvalidate***( )** – invalidate the I960Cx instruction cache (i960)
*cacheI960CxICLoadNLock***( )** – load and lock I960Cx 512-byte instruction cache (i960)
*cacheI960CxIC1kLoadNLock***( )** – load and lock I960Cx 1KB instruction cache (i960)

```
void cacheI960CxICDisable
     (void)

void cacheI960CxICEnable
     (void)

void cacheI960CxICInvalidate
     (void)

void cacheI960CxICLoadNLock
     (void*address)

void cacheI960CxIC1kLoadNLock
     (void*address)
```

DESCRIPTION This library contains Intel I960Cx cache management routines written in assembly language. The I960CX utilize a 1KB instruction cache and no data cache.

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

INCLUDE FILES **cacheLib.h**

SEE ALSO **cacheI960CxLib**, **cacheLib**, *I960Cx Processors User's Manual*

cacheI960CxLib

NAME **cacheI960CxLib** – I960Cx cache management library

SYNOPSIS *cacheI960CxLibInit***( )** – initialize the I960Cx cache library (i960)

```
STATUS cacheI960CxLibInit
    (CACHE_MODE instMode, CACHE_MODE dataMode)
```

DESCRIPTION This library contains architecture-specific cache library functions for the Intel I960Cx architecture. The I960Cx utilizes a 1KB instruction cache and no data cache. Cache line size is fixed at 16 bytes.

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

INCLUDE FILES **cacheLib.h**

SEE ALSO **cacheLib**, *Intel i960Cx User's Manual*

cacheI960JxALib

NAME **cacheI960JxALib** – I960Jx cache management assembly routines

SYNOPSIS *cacheI960JxICDisable***( )** – disable the I960Jx instruction cache (i960)
*cacheI960JxICEnable***( )** – enable the I960Jx instruction cache (i960)
*cacheI960JxICInvalidate***( )** – invalidate the I960Jx instruction cache (i960)
*cacheI960JxICLoadNLock***( )** – load and lock the I960Jx instruction cache (i960)
*cacheI960JxICStatusGet***( )** – get the I960Jx instruction cache status (i960)
*cacheI960JxICLockingStatusGet***( )** – get the I960Jx I-cache locking status (i960)
*cacheI960JxICFlush***( )** – flush the I960Jx instruction cache (i960)
*cacheI960JxDCDisable***( )** – disable the I960Jx data cache (i960)
*cacheI960JxDCEnable***( )** – enable the I960Jx data cache (i960)
*cacheI960JxDCInvalidate***( )** – invalidate the I960Jx data cache (i960)
*cacheI960JxDCCoherent***( )** – ensure data cache coherency (i960)
*cacheI960JxDCStatusGet***( )** – get the I960Jx data cache status (i960)
*cacheI960JxDCFlush***( )** – flush the I960Jx data cache (i960)

```
void cacheI960JxICDisable
    (void)

void cacheI960JxICEnable
    (void)
```

```
void cacheI960JxICInvalidate
     (void)

void cacheI960JxICLoadNLock
     (sysICCodeStart sysICNoBlocks)

void cacheI960JxICStatusGet
     (sysICStatusBuf)

void cacheI960JxICLockingStatusGet
     (sysICStatusBuf)

void cacheI960JxICFlush
     (sysICDestAddr sysICSetsToStore)

void cacheI960JxDCDisable
     (void)

void cacheI960JxDCEnable
     (void)

void cacheI960JxDCInvalidate
     (void)

void cacheI960JxDCCoherent
     (void)

void cacheI960JxDCStatusGet
     (sysDCStatusBuf)

void cacheI960JxDCFlush
     (sysDCDestAddr sysICSetsToStore)
```

DESCRIPTION

This library contains Intel I960Jx cache-management routines written in assembly language. The I960JF and JD utilize a 4KB instruction cache and a 2KB data cache while the I960JA has a 2KB instruction cache and a 1KB 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. See also the manual entry for **cacheI960JxLib**.

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

INCLUDE FILES **arch/i960/cacheI960JxLib.h**, **cacheLib.h**

SEE ALSO **cacheI960JxLib**, **cacheLib**, *I960Jx Processors User's Manual*

cacheI960JxLib

NAME **cacheI960JxLib** – I960Jx cache management library

SYNOPSIS *cacheI960JxLibInit***( )** – initialize the I960Jx cache library (i960)

```
STATUS cacheI960JxLibInit
    (CACHE_MODE instMode, CACHE_MODE dataMode)
```

DESCRIPTION This library contains architecture-specific cache library functions for the Intel I960Jx architecture. The I960JF utilizes a 4KB instruction cache and a 2KB data cache that operate in write-through mode. The I960JA utilizes a 2KB instruction cache and a 1KB data cache that operate in write-through mode. Cache line size is fixed at 16 bytes.

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

INCLUDE FILES **arch/i960/cacheI960JxLib.h**, **cacheLib.h**

SEE ALSO **cacheLib**, *Intel i960Jx User's Manual*

cacheLib

NAME **cacheLib** – cache management library

SYNOPSIS
*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

```
STATUS cacheLibInit
    (CACHE_MODE instMode, CACHE_MODE dataMode)

STATUS cacheEnable
    (CACHE_TYPE cache)

STATUS cacheDisable
    (CACHE_TYPE cache)

STATUS cacheLock
    (CACHE_TYPE cache, void * address, size_t bytes)

STATUS cacheUnlock
    (CACHE_TYPE cache, void * address, size_t bytes)

STATUS cacheFlush
    (CACHE_TYPE cache, void * address, size_t bytes)

STATUS cacheInvalidate
    (CACHE_TYPE cache, void * address, size_t bytes)

STATUS cacheClear
    (CACHE_TYPE cache, void * address, size_t bytes)

STATUS cachePipeFlush
    (void)

STATUS cacheTextUpdate
    (void * address, size_t bytes)

void * cacheDmaMalloc
    (size_t bytes)

STATUS cacheDmaFree
    (void * pBuf)

STATUS cacheDrvFlush
    (CACHE_FUNCS * pFuncs, void * address, size_t bytes)

STATUS cacheDrvInvalidate
    (CACHE_FUNCS * pFuncs, void * address, size_t bytes)

void * cacheDrvVirtToPhys
    (CACHE_FUNCS * pFuncs, void * address)

void * cacheDrvPhysToVirt
    (CACHE_FUNCS * pFuncs, void * address)
```

DESCRIPTION

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

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

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

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

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

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

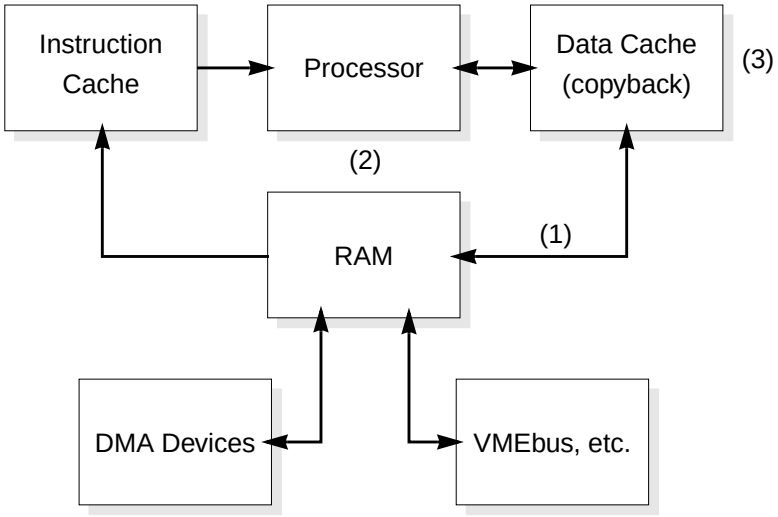
Several routines accept two additional arguments: an address and the number of bytes. Some cache operations can be applied to the entire cache (bytes = **ENTIRE_CACHE**) or to a portion of the cache. This range specification allows the cache to be selectively locked, unlocked, flushed, invalidated, and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

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 pre-empts the access and writes the valid data to RAM. The interrupted access then restarts and retrieves the now-valid data in RAM. Note that this configuration allows only one valid entry at any time. At this time, only a few boards provide the snooping capability; therefore, cache support software must be designed to handle incoherency hazards without degrading performance.

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

USING THE CACHE LIBRARY

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

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

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

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

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

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

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

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

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

Example 1

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

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

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

acquired from the memory allocator. The driver will return OK (line 24) to distinguish a successful from an unsuccessful operation.

```
STATUS drvExample1 ()           /* simple driver - good performance */
    {
3:  void *      pInBuf;         /* input buffer */
4:  void *      pOutBuf;        /* output buffer */
6:  pOutBuf = memalign (_CACHE_ALIGN_SIZE, BUF_SIZE);
8:  if (pOutBuf == NULL)
9:      return (ERROR);         /* memory allocator failed */
11: /* other driver initialization and buffer filling */
13: cacheFlush (DATA_CACHE, pOutBuf, BUF_SIZE);
14: drvWrite (pOutBuf);         /* output data to device */
16: /* more driver code */
18: pInBuf = drvRead ();        /* wait for device data */
19: cacheInvalidate (DATA_CACHE, pInBuf, BUF_SIZE);
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 NULLified to enhance system and driver performance in *sysHwInit***( )**.

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

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

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

INCLUDE FILES

cacheLib.h

SEE ALSO

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

cacheMb930Lib

NAME **cacheMb930Lib** – Fujitsu MB86930 (SPARClite) cache management library

SYNOPSIS *cacheMb930LibInit*( ) – initialize the Fujitsu MB86930 cache library
cacheMb930LockAuto( ) – enable MB86930 automatic locking of kernel instructions / data
cacheMb930ClearLine( ) – clear a line from an MB86930 cache

```
STATUS cacheMb930LibInit
    (CACHE_MODE instMode, CACHE_MODE dataMode)

void cacheMb930LockAuto
    (void)

STATUS cacheMb930ClearLine
    (CACHE_TYPE cache, void * address)
```

DESCRIPTION This library contains architecture-specific cache library functions for the Fujitsu MB86930 (SPARClite) architecture. There are separate small instruction and data caches on chip, both of which operate 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. Locked data cache tags are not invalidated since the data resides only in the cache and not in RAM. The global and local cache locking features are beneficial for real-time systems. Note that there is no MMU (Memory Management Unit) support.

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

INCLUDE FILES **arch/sparc/sparclite.h**, **cacheLib.h**

SEE ALSO **cacheLib**

cacheMicroSparcLib

NAME **cacheMicroSparcLib** – microSPARC cache management library

SYNOPSIS *cacheMicroSparcLibInit*( ) – initialize the microSPARC cache library

```
STATUS cacheMicroSparcLibInit
    (CACHE_MODE instMode, CACHE_MODE dataMode)
```

DESCRIPTION This library contains architecture-specific cache library functions for the microSPARC architecture. Currently two microSPARC CPU are supported: the Texas Instrument TMS3900S10 (also known as Tsunami) and the FUJITSU MB86904 (also know as Swift). The TMS390S10 implements a 4-Kbyte Instruction and a 2-Kbyte Data cache, the MB86904 a 16-Kbyte Instruction and a 8-Kbyte Data cache. Both operate in write-through mode. The Instruction Cache Line size is 32 bytes while the Data Cache Line size is 16 bytes, but for memory allocation purposes, a cache line alignment size of 32 bytes will be assumed. The TMS390S10 either cache only supports invalidation of all entries and no cache locking is available, the MB86904 supports a per cache line invalidation, with specific alternate stores, but no cache locking

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**

---

cacheR33kLib

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

SYNOPSIS *cacheR33kLibInit***( )** – initialize the R33000 cache library

```
STATUS cacheR33kLibInit
    (CACHE_MODE instMode, CACHE_MODE dataMode)
```

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*

cacheR3kALib

NAME **cacheR3kALib** – MIPS R3000 cache management assembly routines

SYNOPSIS *cacheR3kDsize*( ) – return the size of the R3000 data cache
cacheR3kIsize( ) – return the size of the R3000 instruction cache

```
ULONG cacheR3kDsize
    (void)

ULONG cacheR3kIsize
    (void)
```

DESCRIPTION This library contains MIPS R3000 cache set-up and invalidation routines written in assembly language. 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 **cacheR3kLib**.

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

INCLUDE FILES **cacheLib.h**

SEE ALSO **cacheR3kLib**, **cacheLib**, Gerry Kane: *MIPS R3000 RISC Architecture*

cacheR3kLib

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

SYNOPSIS *cacheR3kLibInit*( ) – initialize the R3000 cache library

```
STATUS cacheR3kLibInit
    (CACHE_MODE instMode, CACHE_MODE dataMode)
```

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

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

```
STATUS cacheR4kLibInit
    (CACHE_MODE instMode, CACHE_MODE dataMode)
```

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**

cacheSun4Lib

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

SYNOPSIS *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

```
STATUS cacheSun4LibInit
    (CACHE_MODE instMode, CACHE_MODE dataMode)
```

```
STATUS cacheSun4ClearLine
    (CACHE_TYPE cache, void * address)
```

```
STATUS cacheSun4ClearPage
    (CACHE_TYPE cache, void * address)

STATUS cacheSun4ClearSegment
    (CACHE_TYPE cache, void * address)

STATUS cacheSun4ClearContext
    (CACHE_TYPE cache, void * address)
```

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**

cacheTiTms390Lib

NAME **cacheTiTms390Lib** – TI TMS390 SuperSPARC cache management library

SYNOPSIS

*cacheTiTms390LibInit***( )** – initialize the TI TMS390 cache library
*cacheTiTms390VirtToPhys***( )** – translate a virtual address for **cacheLib**
*cacheTiTms390PhysToVirt***( )** – translate a physical address for drivers
*cleanUpStoreBuffer***( )** – clean up store buffer after a data store error interrupt

```
STATUS cacheTiTms390LibInit
    (CACHE_MODE instMode, CACHE_MODE dataMode)

void * cacheTiTms390VirtToPhys
    (void * address)

void * cacheTiTms390PhysToVirt
    (void * address)

void cleanUpStoreBuffer
    (UINT mcntl, BOOL exception)
```

DESCRIPTION

This library contains architecture-specific cache library functions for the TI TMS390 SuperSPARC architecture. The on-chip cache architecture is explained in the first table below. Note, the data cache mode depends on whether there is an external Multicache Controller (MCC). Both on-chip caches support cache coherency via snooping and line locking. For memory allocation purposes, a cache line alignment size of 64 bytes is assumed. The MCC supports cache coherency via snooping, but does not support line locking.

Cache Type	Size	Lines	Sets	Ways	Line Size (Bytes)	Mode
Instr	20K	320	64	5	2*32	never written back
Data	16K	512	128	4	32	with MCC: Write-through without MCC: Copy-back with write allocation

The cache operations provided are explained in the table below. Operations marked “Hardware” and “Software” are implemented as marked, and are fast and slow, respectively. Operations marked “NOP” return OK without doing anyting. Operations with another operation name perform that operation rather than their own. Partial operations marked “Entire”actually perform an “Entire” operation. When the MCC is installed, operations upon the data cache are performed upon both the data cache and the MCC. Lines “Data-Data” and “Data-MCC” desribe the data cache and MCC, respectively, portions of a data cache operation.

MCC:		No	No	Yes	Yes	Yes
Cache Type:		**Instr**	**Data**	**Instr**	**Data-Data**	**Data-MCC**
*cacheInvalidate***( )**	entire	H/W	H/W	H/W	H/W	S/W
	partial	Entire	S/W	Entire	S/W	S/W
*cacheFlush***( )**	entire	NOP	Clear	NOP	NOP	S/W
	partial	NOP	Clear	NOP	NOP	Clear
*cacheClear***( )**	entire	H/W	S/W	H/W	H/W	S/W
	partial	Entire	S/W	Entire	S/W	S/W
*cacheLock***( )** and	entire	S/W	S/W	S/W	S/W	NOP
*cacheUnlock***( )**	partial	S/W	S/W	S/W	S/W	NOP

The architecture of the optional Multicache Controller (MCC) is explained in the table below. The MCC supports cache coherency via snooping, and does not support line locking.

The MCC does not have a **CACHE_TYPE** value for *cacheEnable***( )** or *cacheDisable***( )**. For enable and disable operations, the MCC is treated as an extension of both the on-chip data and instruction caches. If either the data or instruction caches are enabled, the MCC is enabled. If both the data and the instruction caches are disabled, the MCC is disabled. For invalidate, flush, and clear operations the MCC is treated as an extension of only the on-chip data cache. The *cacheInvalidate***( )**, *cacheFlush***( )**, and *cacheClear***( )** operations for the

instruction cache operate only on the on-chip instruction cache. However these operations for the data cache operate on both the on-chip data cache and the MCC.

Cache Type	Size	Blocks	Ways	Block Size (bytes)	Mode
MCC on MBus	0, 1M	0, 8K	1	4*32	Copy-back
MCC on XBus	512K, 1M, 2M	2K, 4K, 8K	1	4*64	Copy-back

Any input peripheral that does not support cache coherency cay be accessed through either a cached buffer with a partial *cacheTiTms390Invalidate***( )** operation, or an uncached buffer without it. (*cacheInvalidate***( )** cannot be used; it is a NOP since it assumes cache coherency.) Choose whichever is faster for the application.

Any output peripheral that does not support cache coherency may be accessed through either a cached buffer with a partial *cacheTiTms390Flush***( )** operation, or an uncached buffer without it. (*cacheFlush***( )** cannot be used; it is a NOP since it assumes cache coherency.) Choose whichever is faster for the application.

Any peripheral that supports cache coherency should be accessed through a cached buffer without using any of the above operations. Using either an uncached buffer or any of the above operations will just slow the system down.

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**

cd2400Sio

NAME **cd2400Sio** – CL-CD2400 MPCC serial driver

SYNOPSIS *cd2400HrdInit***( )** – initialize the chip
*cd2400IntRx***( )** – handle receiver interrupts
*cd2400IntTx***( )** – handle transmitter interrupts
*cd2400Int***( )** – handle special status interrupts

```
void cd2400HrdInit
    (CD2400_QUSART * pQusart)

void cd2400IntRx
    (CD2400_CHAN * pChan)
```

```
void cd2400IntTx
    (CD2400_CHAN * pChan)

void cd2400Int
    (CD2400_CHAN * pChan)
```

DESCRIPTION This is the driver for the Cirus Logic CD2400 MPCC. It uses the SCC's in asynchronous mode.

USAGE A **CD2400_QUSART** structure is used to describe the chip. This data structure contains four **CD2400_CHAN** structure which describe the chip's four serial channels. The BSP's *sysHwInit***( )** routine typically calls *sysSerialHwInit***( )** which initializes all the values in the **CD2400_QUSART** structure (except the **SIO_DRV_FUNCS**) before calling *cd2400HrdInit***( )**. The BSP's *sysHwInit2***( )** routine typically calls *sysSerialHwInit2***( )** which connects the chips interrupts (cd2400Int, cd2400IntRx, and cd2400IntTx) via *intConnect***( )**.

IOCTL FUNCTIONS This driver responds to the same *ioctl***( )** codes as a normal serial driver; for more information, see the comments in **sioLib.h**. The available baud rates are: 50, 110, 150, 300, 600, 1200, 2400, 3600, 4800, 7200, 9600, 19200, and 38400.

INCLUDE FILES **drv/sio/cd2400Sio.h**

cisLib

NAME **cisLib** – PCMCIA CIS library

SYNOPSIS *cisGet***( )** – get information from a PC card's CIS
*cisFree***( )** – free tuples from the linked list
*cisConfigregGet***( )** – get the PCMCIA configuration register
*cisConfigregSet***( )** – set the PCMCIA configuration register

```
STATUS cisGet
    (int sock)

void cisFree
    (int sock)

STATUS cisConfigregGet
    (int sock, int reg, int *pValue)

STATUS cisConfigregSet
    (int sock, int reg, int value)
```

DESCRIPTION This library contains routines to manipulate the CIS (Configuration Information Structure) tuples and the card configuration registers. The library uses a memory window which is defined in **pcmciaMemwin** to access the CIS of a PC card. All CIS tuples in a PC card are read and stored in a linked list, **cisTupleList**. If there are configuration tuples, they are interpreted and stored in another link list, **cisConifigList**. After the CIS is read, the PC card's enabler routine allocates resources and initializes a device driver for the PC card.

If a PC card is inserted, the CSC (Card Status Change) interrupt handler gets a CSC event from the PCMCIA chip and adds a *cisGet***( )** job to the PCMCIA daemon. The PCMCIA daemon initiates the *cisGet***( )** work. The CIS library reads the CIS from the PC card and makes a linked list of CIS tuples. It then enables the card.

If the PC card is removed, the CSC interrupt handler gets a CSC event from the PCMCIA chip and adds a *cisFree***( )** job to the PCMCIA daemon. The PCMCIA daemon initiates the *cisFree***( )** work. The CIS library frees allocated memory for the linked list of CIS tuples.

cisShow

NAME **cisShow** – PCMCIA CIS show library

SYNOPSIS *cisShow***( )** – show CIS information

```
void cisShow
    (int sock)
```

DESCRIPTION This library provides a show routine for CIS tuples.

clockLib

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

SYNOPSIS *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)

```
int clock_getres
    (clockid_t clock_id, struct timespec * res)
```

```
int clock_setres
    (clockid_t clock_id, struct timespec * res)

int clock_gettime
    (clockid_t clock_id, struct timespec * tp)

int clock_settime
    (clockid_t clock_id, const struct timespec * tp)
```

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***( )** is provided (temporarily) so that **clockLib** is informed if there are changes in the system clock rate (e.g., after a call to *sysClkRateSet***( )**).

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

---

connLib

NAME **connLib** – target-host connection library (WindView)

SYNOPSIS *connRtnSet***( )** – set up connection routines for target-host communication (WindView)

```
void connRtnSet
    (FUNCPTR initRtn, VOIDFUNCPTR closeRtn, VOIDFUNCPTR errorRtn,
    VOIDFUNCPTR dataXferRtn)
```

DESCRIPTION This library provides routines for configuring the WindView target-host connection.

By default, the routines provide target-host communication over TCP sockets. Users can replace these routines with their own, without modifying the kernel or the WindView architecture. For example, the user may want to write the WindView event buffer to shared memory, diskette, or hard disks.

Four routines are required: An initialization routine, a close connection routine, an error handler, and a routine that transfers the data from the event buffer to another location.

The data transfer routine must complete its job before the next data transfer cycle. If it fails to do so, a bandwidth exceeded condition occurs and event logging stops.

INCLUDE FILES **connLib.h**

SEE ALSO **wvLib**, *WindView User's Guide*

cplusLib

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

SYNOPSIS

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

```
extern "C" STATUS cplusLibMinInit
    (void)

extern void operator delete
    (void *pMem)

extern "C" void cplusCtorsLink()

extern "C" void cplusDtorsLink()

extern "C" STATUS cplusLibInit
    (void)

extern void * operator new
    (size_t n)

extern void cplusCallNewHandler()

extern void
    (*set_new_handler (void(*pNewNewHandler)()))()
```

```
extern void * operator new
    (size_t, void *pMem)

extern "C" void cplusDemanglerSet
    (int mode)

extern "C" void cplusXtorSet
    (int strategy)

extern "C" void cplusCtors
    (const char * moduleName)

extern "C" void cplusDtors
    (const char * moduleName)
```

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 arrays of C++ objects.
- 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 *Tornado User's Guide: Cross-Development*.

SEE ALSO

Tornado User's Guide: Cross-Development

dbgArchLib

NAME

dbgArchLib – architecture-dependent debugger library

SYNOPSIS

*g0***( )** – return the contents of register **g0**, also **g1** – **g7** (SPARC) and **g1** – **g14** (i960)
*a0***( )** – return the contents of register **a0** (also **a1** – **a7**) (MC680x0)
*d0***( )** – return the contents of register **d0** (also **d1** – **d7**) (MC680x0)
*sr***( )** – return the contents of the status register (MC680x0)
*psrShow***( )** – display the meaning of a specified **psr** value, symbolically (SPARC)
*fsrShow***( )** – display the meaning of a specified fsr value, symbolically (SPARC)

o0( ) – return the contents of register **o0** (also **o1** – **o7**) (SPARC)
l0( ) – return the contents of register **l0** (also **l1** – **l7**) (SPARC)
i0( ) – return the contents of register **i0** (also **i1** – **i7**) (SPARC)
npc( ) – return the contents of the next program counter (SPARC)
psr( ) – return the contents of the processor status register (SPARC)
wim( ) – return the contents of the window invalid mask register (SPARC)
y( ) – return the contents of the **y** register (SPARC)
pfp( ) – return the contents of register **pfp** (i960)
tsp( ) – return the contents of register **sp** (i960)
rip( ) – return the contents of register **rip** (i960)
r3( ) – return the contents of register **r3** (also **r4** – **r15**) (i960)
fp( ) – return the contents of register **fp** (i960)
fp0( ) – return the contents of register **fp0** (also **fp1** – **fp3**) (i960KB, i960SB)
pcw( ) – return the contents of the **pcw** register (i960)
tcw( ) – return the contents of the **tcw** register (i960)
acw( ) – return the contents of the **acw** register (i960)
dbgBpTypeBind( ) – bind a breakpoint handler to a breakpoint type (MIPS R3000, R4000)
edi( ) – return the contents of register **edi** (also **esi** – **eax**) (i386/i486)
eflags( ) – return the contents of the status register (i386/i486)

```
int g0
    (int taskId)

int a0
    (int taskId)

int d0
    (int taskId)

int sr
    (int taskId)

void psrShow
    (ULONG psrValue)

void fsrShow
    (UINT fsrValue)

int o0
    (int taskId)

int l0
    (int taskId)

int i0
    (int taskId)

int npc
    (int taskId)
```

```
int psr
    (int taskId)

int wim
    (int taskId)

int y
    (int taskId)

int pfp
    (int taskId)

int tsp
    (int taskId)

int rip
    (int taskId)

int r3
    (int taskId)

int fp
    (int taskId)

double fp0
    (volatile int taskId)

int pcw
    (int taskId)

int tcw
    (int taskId)

int acw
    (int taskId)

STATUS dbgBpTypeBind
    (int bpType, FUNCPTR routine)

int edi
    (int taskId)

int eflags
    (int taskId)
```

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**)

SPARC:	*psrShow***( )**	– **psr** value, symbolically
	*fsrShow***( )**	– **fsr** value, symbolically
	*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
i960:	*g0***( )** – *g14***( )**	– global registers
	*r3***( )** – *r15***( )**	– local registers
	*tsp***( )**	– stack pointer
	*rip***( )**	– return instruction pointer
	*pfp***( )**	– previous frame pointer
	*fp***( )**	– frame pointer
	*fp0***( )** – *fp3***( )**	– floating-point registers (i960 KB and SB only)
	*pcw***( )**	– processor control word
	*tcw***( )**	– trace control word
	*acw***( )**	– arithmetic control word
MIPS:	*dbgBpTypeBind***( )**	– bind a breakpoint handler to a breakpoint type
i386/i486:	*edi***( )** – *eax***( )**	– named register values
	*eflags***( )**	– status register value

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

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

dbgLib

NAME **dbgLib** – debugging facilities

SYNOPSIS
*dbgHelp***( )** – display the 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( ) – print a stack trace of a task

```
void dbgHelp
    (void)

STATUS dbgInit
    (void)

STATUS b
    (INSTR * addr, int task, int count, BOOL quiet)

STATUS e
    (INSTR * addr, event_t eventId, int taskNameOrId, FUNCPTR evtRtn,
    int arg)

STATUS bh
    (INSTR * addr, int access, int task, int count, BOOL quiet)

STATUS bd
    (INSTR * addr, int task)

STATUS bdall
    (int task)

STATUS c
    (int task, INSTR * addr, INSTR * addr1)

STATUS cret
    (int task)

STATUS s
    (int taskNameOrId, INSTR * addr, INSTR * addr1)

STATUS so
    (int task)

void l
    (INSTR * addr, int count)

STATUS tt
    (int 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 in a VxWorks system by defining **INCLUDE_DEBUG** in **configAll.h**. This will enable 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.

CAVEAT

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*

---

dirLib

NAME

dirLib – directory handling library (POSIX)

SYNOPSIS

*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

```
DIR *opendir
    (char *dirName)

struct dirent *readdir
    (DIR *pDir)

void rewinddir
    (DIR *pDir)

STATUS closedir
    (DIR *pDir)

STATUS fstat
    (int fd, struct stat *pStat)

STATUS stat
    (char *name, struct stat *pStat)

STATUS fstatfs
    (int fd, struct statfs *pStat)

STATUS statfs
    (char *name, struct statfs *pStat)

int utime
    (char * file, struct utimbuf * newTimes)
```

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**

dlpiLib

NAME **dlpiLib** – Data Link Provider Interface (DLPI) Library (STREAMS Opt.)

SYNOPSIS *dlpiInit***( )** – initialize the DLPI driver

```
STATUS dlpiInit
    (void)
```

DESCRIPTION This library implements the generic Data Link Provider Interface (DLPI) driver which is common for all network drivers. This is a STREAMS-based interface between the data link layer (the Data Link Service provider) and the network layer. This library enables a Data Link Service (DLS) user to access the DLPI-conformant driver (the DLS provider). It also provides an interface to the Wind River-specific network drivers.

USER-CALLABLE ROUTINES

The DLPI interface is initialized by the *dlpiInit***( )** routine which installs the DLPI STREAMS driver in the VxWorks I/O subsystem.

IMPLEMENTATION This library supports up to eight SAPs (Service Access Points). The driver open calls are treated as clone opens, thereby assigning a new stream for each open. Each opened stream is bound to a SAP; there is a one-to-one correspondence between open stream and SAP.

The DLPI driver serves as the generic driver under which operates a network driver. The network driver hands over the received packets to the DLPI driver using the network driver's **etherInputHook** function pointer. This pointer is installed at the time the stream is bound to the SAP, that is, when the **DL_BIND_REQ** primitive is called by the DLS user. The network driver must support Ethernet input hooks. For more information on Ethernet input hooks, see the manual entry for **etherLib**. This DLPI driver is a style 2 DLS provider.

The **DL_ATTACH_REQ** (attach request) primitive generated by the user should concatenate the name of the network device to be attached to the attach-request message. The attach-request primitive implemented in this driver gets the name of the appropriate network device from the attach-request message. It then gets the pointer to the appropriate network-controller data structure from the name obtained. The network device-control structure obtained is a pointer to an **arpcom** structure. The attach-request primitive calls *ifunit***( )** to obtain the pointer to the device-control structure.

The packet **type** field in the Ethernet frame is used to multiplex between various SAPs. This DLPI driver supports only Ethernet frame formats. It does not support IEEE 802.3 frame formats.

DLPI SERVICES. This library supports a subset of DLPI services. The services provided by this library are:

DL_ATTACH_REQ
DL_DETACH_REQ
DL_BIND_REQ

DL_BIND_ACK
DL_INFO_REQ
DL_INFO_ACK
DL_UNBIND_REQ
DL_ERROR_ACK
DL_UNITDATA_REQ
DL_UNITDATA_IND
DL_OK_ACK

INCLUDE FILES **dlpi.h**, **stream.h**, **mikernel.h**

SEE ALSO **strmLib**, *Data Link Provider Interface Specification, Revision 2.0.0, UNIX SVR4.2 STREAMS-based Data Link Provider Interface.*

dosFsLib

NAME **dosFsLib** – MS-DOS® media-compatible file system library

SYNOPSIS
*dosFsConfigGet***( )** – obtain dosFs volume configuration values
*dosFsConfigInit***( )** – initialize dosFs volume configuration structure
*dosFsConfigShow***( )** – display dosFs volume configuration data
*dosFsDateSet***( )** – set the dosFs file system date
*dosFsDateTimeInstall***( )** – install a user-supplied date/time function
*dosFsDevInit***( )** – associate a block device with dosFs file system functions
*dosFsDevInitOptionsSet***( )** – specify volume options for *dosFsDevInit***( )**
*dosFsInit***( )** – prepare to use the dosFs library
*dosFsMkfs***( )** – initialize a device and create a dosFs file system
*dosFsMkfsOptionsSet***( )** – specify volume options for *dosFsMkfs***( )**
*dosFsModeChange***( )** – modify the mode of a dosFs volume
*dosFsReadyChange***( )** – notify dosFs of a change in ready status
*dosFsTimeSet***( )** – set the dosFs file system time
*dosFsVolOptionsGet***( )** – get current dosFs volume options
*dosFsVolOptionsSet***( )** – set dosFs volume options
*dosFsVolUnmount***( )** – unmount a dosFs volume

```
STATUS dosFsConfigGet
    (DOS_VOL_DESC *vdptr, DOS_VOL_CONFIG *pConfig)

STATUS dosFsConfigInit
    (DOS_VOL_CONFIG *pConfig, char mediaByte, UINT8 secPerClust,
    short nResrvd, char nFats, UINT16 secPerFat, short maxRootEnts,
    UINT nHidden, UINT options)
```

```
STATUS dosFsConfigShow
    (char *devName)

STATUS dosFsDateSet
    (int year, int month, int day)

void dosFsDateTimeInstall
    (FUNCPTR pDateTimeFunc)

DOS_VOL_DESC *dosFsDevInit
    (char *devName, BLK_DEV *pBlkDev, DOS_VOL_CONFIG *pConfig)

STATUS dosFsDevInitOptionsSet
    (UINT options)

STATUS dosFsInit
    (int maxFiles)

DOS_VOL_DESC *dosFsMkfs
    (char *volName, BLK_DEV *pBlkDev)

STATUS dosFsMkfsOptionsSet
    (UINT options)

void dosFsModeChange
    (DOS_VOL_DESC *vdptr, int newMode)

void dosFsReadyChange
    (DOS_VOL_DESC *vdptr)

STATUS dosFsTimeSet
    (int hour, int minute, int second)

STATUS dosFsVolOptionsGet
    (DOS_VOL_DESC * vdptr, UINT * pOptions)

STATUS dosFsVolOptionsSet
    (DOS_VOL_DESC * vdptr, UINT options)

STATUS dosFsVolUnmount
    (DOS_VOL_DESC *vdptr)
```

DESCRIPTION

This library provides services for file-oriented device drivers to use the MS-DOS® file standard. This module takes care of all necessary buffering, directory maintenance, and file system details.

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 *dosFsInit***( )** routine is the principal initialization function; it need only be called once, regardless of how many dosFs devices are to be used. In addition, *dosFsDateTimeInstall***( )** (if used) will typically be called only once, prior to performing any actual file operations, to install a user-supplied routine which provides the current date and time.

Other dosFs functions are used for device initialization. For each dosFs device, either *dosFsDevInit***( )** or *dosFsMkfs***( )** must be called to install the device and define its configuration. The *dosFsConfigInit***( )** routine is provided to easily initialize the data structure used during device initialization; however, its use is optional.

Several routines are provided to inform the file system of changes in the system environment. The *dosFsDateSet***( )** and *dosFsTimeSet***( )** routines are used to set the current date and time; these are normally used only if no user routine has been installed via *dosFsDateTimeInstall***( )**. The *dosFsModeChange***( )** call may be used to modify the readability or writability of a particular device. The *dosFsReadyChange***( )** 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. Finally, *dosFsVolUnmount***( )** informs the file system that a particular device should be synchronized and unmounted, generally in preparation for a disk change.

More detailed information on all of these routines is discussed in the following sections.

INITIALIZING DOSFSLIB

Before any other routines in **dosFsLib** can be used, the routine *dosFsInit***( )** must be called to initialize this library. This call specifies the maximum number of dosFs files that can be open simultaneously. Attempts to open more dosFs files than the specified maximum will result in errors from *open***( )** and *creat***( )**.

To enable this initialization, define **INCLUDE_DOSFS** in **configAll.h**; *dosFsInit***( )** will then be called from the root task, *usrRoot***( )**, in **usrConfig.c**.

DEFINING A DOSFS DEVICE

To use this library for a particular device, the device descriptor structure used by the device driver must contain, as the very first item, a block device description structure (**BLK_DEV**). This must be initialized before calling *dosFsDevInit***( )**. 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. These routines are described below. The **BLK_DEV** structure also contains fields which describe the physical configuration of the device. For more information about defining block devices, see the *VxWorks Programmer's Guide: I/O System*.

The *dosFsDevInit***( )** 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. 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) A pointer to the **BLK_DEV** structure which describes the device and contains the addresses of the five required functions. The fields in this structure must have been initialized before the call to *dosFsDevInit*( ).

(3) A pointer to a volume configuration structure (**DOS_VOL_CONFIG**). This structure contains configuration data for the volume which are specific to the dosFs file system. (See "Changes in Volume Configuration", below, for more information.) The fields in this structure must have been initialized before the call to *dosFsDevInit*( ). The **DOS_VOL_CONFIG** structure may be initialized by using the *dosFsConfigInit*( ) routine.

As an example:

```
dosFsDevInit
    (
    char            *volName,     /* name to be used for volume   */
    BLK_DEV         *pBlkDev,     /* pointer to device descriptor */
    DOS_VOL_CONFIG  *pVolConfig   /* pointer to vol config data   */
    )
```

Once *dosFsDevInit*( ) has been called, when **dosFsLib** receives a request from the I/O system, it calls the device driver routines (whose addresses were passed in the **BLK_DEV** structure) to access the device.

The *dosFsMkfs*( ) routine is an alternative to using *dosFsDevInit*( ). The *dosFsMkfs*( ) routine always initializes a new dosFs file system on the disk; thus, it is unsuitable for disks containing data that should be preserved. Default configuration parameters are supplied by *dosFsMkfs*( ), since no **DOS_VOL_CONFIG** structure is used.

See "Network File System (NFS) Support", below, for additional NFS-related parameters you can set before calling *dosFsDevInit*( ).

MULTIPLE LOGICAL DEVICES

The sector number passed to the driver's 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 can be done by keeping an offset parameter in the driver device structure, and adding the offset to the sector number passed by the file system's read and write routines.

ACCESSING THE RAW DISK

As a special case in *open*( ) and *creat*( ) calls, the dosFs file system recognizes a null filename as indicating access to the entire "raw" disk rather than to an individual file on the disk. (To open a device in raw mode, specify only the device name — no filename — during the *open*( ) or *creat*( ) call.)

Raw mode is the only means of accessing a disk that has no file system. For example, to initialize a new file system on the disk, first the raw disk is opened and the returned file descriptor is used for an *ioctl***( )** call with **FIODISKINIT**. Opening the disk in raw mode is also a common operation when doing other *ioctl***( )** functions which do not involve a particular file (e.g., **FIONFREE**, **FIOLABELGET**).

To read the root directory of a disk on which no file names are known, specify the device name when calling *opendir***( )**. Subsequent *readdir***( )** calls will return the names of files and subdirectories in the root directory.

Data written to the disk in raw mode uses the same area on the disk as normal dosFs files and subdirectories. Raw I/O does not use the disk sectors used for the boot sector, root directory, or File Allocation Table (FAT). For more information about raw disk I/O using the entire disk, see the manual entry for **rawFsLib**.

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 *dosFsDevInit***( )** or *dosFsMkfs***( )** call.

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

When using the VxWorks shell to make calls specifying dosFs pathnames, you must allow for the C-style interpretation which is performed. In cases where the file name is enclosed in quote marks, any backslashes must be "escaped" by a second, preceding backslash. For example:

```
-> copy ("DOS1:\\subdir\\file1", "file2")
```

However, shell commands which use pathnames without enclosing quotes do not require the second backslash. For example:

```
-> copy < DOS1:\subdir\file1
```

Forward slashes do not present these inconsistencies, and may therefore be preferable for use within the shell.

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 FILE NAMES

The MS-DOS standard only allows for file names which fit the restrictions of eight upper-case characters optionally followed by a three-character extension. This may not be

convenient if you are transferring files to or from a remote system, or if your application requires particular file naming conventions.

To provide additional flexibility, the dosFs file system provides an option to use longer, less restricted file names. When this option is enabled, file names may consist of any sequence of up to 40 ASCII characters. No case conversion is performed and no characters have any special significance.

NOTE: Because special directory entries are used on the disk, disks which use the extended names are *not* compatible with true MS-DOS systems and cannot be read on MS-DOS machines. Disks which use the extended name option must be initialized by the VxWorks dosFs file system (using FIODISKINIT); disks which have been initialized (software-formatted) on MS-DOS systems cannot be used.

To enable the extended file names, set the **DOS_OPT_LONGNAMES** bit in the **dosvc_options** field in the **DOS_VOL_CONFIG** structure when calling *dosFsDevInit***( )**. (The *dosFsMkfs***( )** routine may also be used to enable extended file names; however, the **DOS_OPT_LONGNAMES** option must already have been specified in a previous call to *dosFsMkfsOptionsSet***( )**.)

NETWORK FILE SYSTEM (NFS) SUPPORT

To enable the export of a file system, the **DOS_OPT_EXPORT** option must be set when initializing the device via *dosFsDevInit***( )** or *dosFsMkfs***( )**. This option may also be made the default for use with disks when no explicit configuration is given. See the manual entry for *dosFsDevInitOptionsSet***( )**.

If the remote client that will be mounting the dosFs volume is a PC-based client, you may also need to specify the **DOS_OPT_LOWERCASE** option. This option causes filenames to be mapped to lowercase (when not using the **DOS_OPT_LONGNAMES** option). This lowercase mapping is expected by many PC-based NFS implementations.

When the **DOS_OPT_EXPORT** option is enabled, the VxWorks NFS file system uses the reserved fields of a dosFs directory entry to store information needed to uniquely identify a dosFs file.

Every time a file is created in a directory, the directory timestamp is incremented. This is necessary to avoid cache inconsistencies in clients, because some UNIX clients use the directory timestamp to determine if their local cache needs to be updated.

You can also specify integers for a user ID, group ID, and file access permissions byte when you initialize a dosFs file system for NFS export. The values you specify will apply to all files in the file system.

Set **dosFsUserId** to specify the numeric user ID. The default is 65534.

Set **dosFsGroupId** to specify the numeric group ID. The default is 65534.

Set **dosFsFileMode** to specify the numeric file access mode. The default is 777.

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 a time and date for each file or subdirectory. This time is set when the file is created, and it is updated when a file is closed, if it has been modified. Directory time and date fields are set only when the directory is created, not when it is modified.

The dosFs file system library maintains the date and time in an internal structure. While there is currently no mechanism for automatically advancing the date or time, two different methods for setting the date and time are provided.

The first method involves using two routines, *dosFsDateSet*( ) and *dosFsTimeSet*( ), which are provided to set the current date and time.

Examples of setting the date and time would be:

```
dosFsDateSet (1990, 12, 25);  /* set date to Dec-25-1990 */
dosFsTimeSet (14, 30, 22);    /* set time to 14:30:22    */
```

The second method requires a user-provided hook routine. If a time and date hook routine is installed using *dosFsDateTimeInstall*( ), the routine will be called whenever **dosFsLib** requires the current date. This facility is provided to take advantage of hardware time-of-day clocks which may be read to obtain the current time.

The date/time hook routine should be defined as follows:

```
void dateTimeHook
    (
    DOS_DATE_TIME   *pDateTime   /* ptr to dosFs date/time struct */
    )
```

On entry to the hook routine, the **DOS_DATE_TIME** structure will contain the last time and date which was set in **dosFsLib**. The structure should then be filled by the hook routine with the correct values for the current time and date. Unchanged fields in the structure will retain their previous values.

The MS-DOS specification only provides for 2-second granularity for file time stamps. If the number of seconds in the time specified during *dosFsTimeSet*( ) or the date/time hook routine is odd, it will be rounded down to the next even number.

The date and time used by **dosFsLib** is initially Jan-01-1980, 00:00:00.

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
DOS_ATTR_HIDDEN
DOS_ATTR_SYSTEM
DOS_ATTR_VOL_LABEL
DOS_ATTR_DIRECTORY
DOS_ATTR_ARCHIVE

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         */
```

CONTIGUOUS FILE SUPPORT

The VxWorks dosFs file system provides efficient handling of 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 (or directory) and optimized access to such a file when it is used.

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** function. The other parameter to the **FIOCONTIG** function is the size of the requested contiguous area in bytes. It is also possible to request that the largest contiguous free area on the disk be obtained. In this case, the special value **CONTIG_MAX** (-1) is used instead of an actual size.

The FAT is searched for a suitable section of the disk, and if found, it is assigned to the file. (If there is no contiguous area on the volume large enough to satisfy the request, an **S_dosFsLib_NO_CONTIG_SPACE** error is returned.) The file may then be closed or used for further I/O operations. For example, the following will create a file and allocate 0x10000 contiguous bytes:

```
fd = creat ("file", O_RDWR, 0);            /* open file            */
status = ioctl (fd, FIOCONTIG, 0x10000);   /* 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            */
```

It is important that the file descriptor used for the *ioctl***( )** call be the only descriptor open to the file. Furthermore, since a file may be assigned a different area of the disk than was originally allocated, the **FIOCONTIG** operation should take place before any data is written to the file.

To determine the actual amount of contiguous space obtained when **CONTIG_MAX** is specified as the size, use *fstat***( )** to examine the file size. For more information, see **dirLib**.

Space which has been allocated to a file may later be freed by using *ioctl***( )** with the **FIOTRUNC** function.

Directories may also be allocated a contiguous disk area. A file descriptor to the directory is used to call **FIOCONTIG**, just as for a regular file. A directory should be empty (except for the "." and ".." entries) before it has contiguous space allocated to it. The root directory allocation may not be changed. Space allocated to a directory is not reclaimed until the directory is deleted; directories may not be truncated using the **FIOTRUNC** function.

When any file is opened, it is checked for contiguity. If a file is recognized as contiguous, more efficient techniques for locating specific sections of the file are used, rather than following cluster chains in the FAT as must be done for fragmented files. This enhanced handling of contiguous files takes place regardless of whether the space was actually allocated using **FIOCONTIG**.

CHANGING, UNMOUNTING, AND SYNCHRONIZING DISKS

Copies of directory entries and the FAT for each volume are kept in memory. This greatly speeds up access to files, but it requires that **dosFsLib** be notified when disks are changed (i.e., floppies are swapped). Two different notification mechanisms are provided.

Unmounting Volumes

The first, and preferred, method of announcing a disk change is for *dosFsVolUnmount***( )** to be called prior to removal of the disk. This call flushes all modified data structures to disk, if possible (see the description of disk synchronization below), and also marks any open file descriptors as obsolete. During the next I/O operation, the disk is remounted. The *ioctl***( )** call may also be used to initiate *dosFsVolUnmount***( )** by specifying the function code **FIOUNMOUNT**. (Any open file descriptor to the device may be used in the *ioctl***( )** call.)

There may be open files or directories on a dosFs 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_dosFsLib_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_dosFsLib_FD_OBSOLETE**. File descriptors acquired when opening the entire volume (raw mode) will not be marked as obsolete during *dosFsVolUnmount***( )** and may still be used.

Interrupt handlers must not call *dosFsVolUnmount***( )** directly, because it is possible for the *dosFsVolUnmount***( )** call to block while the device becomes available. The interrupt handler may instead give a semaphore which readies a task to unmount the volume. (Note that *dosFsReadyChange***( )** may be called directly from interrupt handlers.)

When *dosFsVolUnmount***( )** is called, it attempts to write buffered data out to the disk. It is therefore inappropriate for situations where the disk change notification does not occur until a new disk has been inserted. (The old buffered data would be written to the new disk.) In these circumstances, *dosFsReadyChange***( )** should be used.

If *dosFsVolUnmount***( )** is called after the disk is physically removed (i.e., there is no disk in the drive), the data-flushing operation will fail. However, the file descriptors will still be marked as obsolete, and the disk will be marked as requiring remounting. An error will not be returned by *dosFsVolUnmount***( )** in this situation. To avoid lost data in such a situation, the disk should be explicitly synchronized before it is removed.

Announcing Disk Changes with Ready-Change

The second method of informing **dosFsLib** that a disk change is taking place is via the "ready-change" mechanism. A change in the disk's ready status is interpreted by **dosFsLib** to indicate that the disk should be remounted during the next I/O operation.

There are three ways to announce a ready-change. First, the *dosFsReadyChange***( )** routine may be called directly. Second, the *ioctl***( )** call may be used, with the **FIODISKCHANGE** function code. Finally, the device driver may set the "bd_readyChanged" field in the **BLK_DEV** structure to TRUE. This has the same effect as notifying **dosFsLib** directly.

The ready-change mechanism does not provide the ability to flush data structures to the disk. It merely marks the volume as needing remounting. As a result, buffered data (data written to files, directory entries, or FAT changes) may be lost. This may be avoided by synchronizing the disk before asserting ready-change. (The combination of synchronizing and asserting ready-change provides all the functionality of *dosFsVolUnmount***( )**, except for marking file descriptors as obsolete.)

Since it does not attempt to flush data or to perform other operations that could cause a delay, ready-change may be used in interrupt handlers.

Disks with No Change Notification

If it is not possible for *dosFsVolUnmount***( )** or *dosFsReadyChange***( )** to be called each time the disk is changed, the device must be specially identified when it is initialized with the file system. One of the parameters of *dosFsDevInit***( )** is the address of a

DOS_VOL_CONFIG structure, which specifies various configuration parameters. **DOS_OPT_CHANGENOWARN** must be set in the **dosvc_options** field of the **DOS_VOL_CONFIG** structure, if the driver and/or application is unable to issue a *dosFsVolUnmount***( )** call or assert a ready-change when a disk is changed.

This configuration option results in a significant performance disadvantage, because the disk configuration data must be regularly read in from the physical disk, in case the disk has been changed. In addition, setting **DOS_OPT_CHANGENOWARN** also enables auto-sync mode (see below).

Note that for disk change notification, all that is required is that *dosFsVolUnmount***( )** or *dosFsReadyChange***( )** be called each time the disk is changed. It is not necessary that either routine be called from the device driver or an interrupt handler. For example, if your application provided a user interface through which an operator could enter a command which would result in a *dosFsVolUnmount***( )** call before removing the disk, that would be sufficient, and **DOS_OPT_CHANGENOWARN** should not be set. It is important, however, that such a procedure be followed strictly.

Synchronizing Volumes

A disk should be "synchronized" before is is unmounted. To synchronize a disk means to write out all buffered data (files, directories, and the FAT table) that have been modified, so that the disk is "up-to-date." It may or may not be necessary to explicitly synchronize a disk, depending on when (or if) the *dosFsVolUnmount***( )** call is issued.

When *dosFsVolUnmount***( )** is called, an attempt will be made to synchronize the device before unmounting. If the disk is still present and writable at the time *dosFsVolUnmount***( )** is called, the synchronization will take place; there is no need to independently synchronize the disk.

However, if *dosFsVolUnmount***( )** is called after a disk has been removed, it is obviously too late to synchronize. (In this situation, *dosFsVolUnmount***( )** discards the buffered data.) Therefore, a separate *ioctl***( )** call with the **FIOFLUSH** or **FIOSYNC** function should be made before the disk is removed. (This could be done in response to an operator command.)

Auto-Sync Mode

The dosFs file system provides a modified mode of behavior called "auto-sync."This mode is enabled by setting **DOS_OPT_AUTOSYNC** in the **dosvc_options** field of the **DOS_VOL_CONFIG** structure when calling *dosFsDevInit***( )**. When this option is enabled, modified directory and FAT data is written to the physical device as soon as these structures are altered. (Normally, such changes may not be written out until the involved file is closed.) This results in a performance penalty, but it provides the highest level of data security, since it minimizes the amount of time when directory and FAT data on the disk are not up-to-date.

Auto-sync mode is automatically enabled if the volume does not have disk change notification, i.e., if **DOS_OPT_CHANGENOWARN** is set in the **dosvc_options** field of the **DOS_VOL_CONFIG** structure when *dosFsDevInit***( )** is called. It may also be desirable for applications where data integrity—in case of a system crash—is a larger concern than simple disk I/O performance.

CHANGES IN VOLUME CONFIGURATION

Various disk configuration parameters are specified when the dosFs device is first initialized using *dosFsDevInit***( )**. This data is kept in the volume descriptor (**DOS_VOL_DESC**) for the device. However, it is possible for a disk with different parameters than those defined to be placed in a drive after the device has already been initialized. For such a disk to be usable, the configuration data in the volume descriptor must be modified when a new disk is present.

When a disk is mounted, the boot sector information is read from the disk. This data is used to update the configuration data in the volume descriptor. This happens the first time the disk is accessed after the volume is unmounted (using *dosFsVolUnmount***( )**).

This automatic re-initialization of the configuration data has two important implications:

(1) Since the values in the volume descriptor are reset when a new volume is mounted, it is possible to omit the dosFs configuration data (by specifying a NULL pointer instead of the address of a **DOS_VOL_CONFIG** structure during *dosFsDevInit***( )**). The first use of the volume must be with a properly formatted and initialized disk. (Attempting to initialize a disk, using **FIODISKINIT**, before a valid disk has been mounted is fruitless.)

(2) The volume descriptor data is used when initializing a disk (with **FIODISKINIT**). The **FIODISKINIT** function initializes a disk with the configuration of the most recently mounted disk, regardless of the original specification during *dosFsDevInit***( )**. Therefore, it is recommended that **FIODISKINIT** be used immediately after *dosFsDevInit***( )**, before any disk has been mounted. (The device should be opened in raw mode; the **FIODISKINIT** function is then performed; and the device is then closed.)

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.

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 a DOS 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

*dosFsReadyChange***( )**. 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*:

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

FIORENAME

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

```
status = ioctl (fd, FIORENAME, "newname");
```

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);
```

FIOFLUSH

Flushes the file output buffer. It guarantees that any output that has been requested is actually written to the device. If the specified file descriptor was obtained by opening the entire volume (raw mode), this function will flush all buffered file buffers, directories, and the FAT table to the physical device:

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

FIOSYNC

Same as **FIOFLUSH**, but additionally re-reads buffered file data from the disk. This allows file changes made via a different file descriptor to be seen.

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 returned, 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 or the entire volume 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);
```

FIONREAD

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

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

FIONFREE

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

```
status = ioctl (fd, FIONFREE, &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 (in root directory) and copies the string to *labelBuffer*:

```
status = ioctl (fd, FIOLABELGET, &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, "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);
```

FIONCONTIG

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

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

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 **dosFsLib.h**

SEE ALSO **ioLib**, **iosLib**, **dirLib**, **ramDrv**, *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*

envLib

NAME **envLib** – environment variable library

SYNOPSIS

*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

```
STATUS envLibInit
    (BOOL installHooks)

STATUS envPrivateCreate
    (int taskId, int envSource)

STATUS envPrivateDestroy
    (int taskId)

STATUS putenv
    (char *pEnvString)
```

```
char *getenv
    (const char *name)

void envShow
    (int taskId)
```

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 **VX_PRIVATE_ENV** 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

SYNOPSIS *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

```
int errnoGet
    (void)

int errnoOfTaskGet
    (int taskId)

STATUS errnoSet
    (int errorValue)

STATUS errnoOfTaskSet
    (int taskId, int errorValue)
```

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:

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

This creates a file **statTbl.c**, 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*

etherLib

NAME **etherLib** – Ethernet raw I/O routines and hooks

SYNOPSIS *etherOutput***( )** – send a packet on an Ethernet interface
*etherInputHookAdd***( )** – add a routine to receive all Ethernet input packets
*etherInputHookDelete***( )** – delete a network interface input hook routine
*etherOutputHookAdd***( )** – add a routine to receive all Ethernet output packets
*etherOutputHookDelete***( )** – delete a network interface output hook routine
*etherAddrResolve***( )** – resolve an Ethernet address for a specified Internet address

```
STATUS etherOutput
    (struct ifnet *pIf, struct ether_header *pEtherHeader, char *pData,
    int dataLength)

STATUS etherInputHookAdd
    (FUNCPTR inputHook)
```

```
void etherInputHookDelete
    (void)

STATUS etherOutputHookAdd
    (FUNCPTR outputHook)

void etherOutputHookDelete
    (void)

STATUS etherAddrResolve
    (struct ifnet *pIf, char *targetAddr, char *eHdr, int numTries,
    int numTicks)
```

DESCRIPTION

This library provides utilities that give direct access to Ethernet packets. Raw packets can be output directly to an interface using *etherOutput***( )**. Incoming and outgoing packets can be examined or processed using the hooks *etherInputHookAdd***( )** and *etherOutputHookAdd***( )**. The input hook can be used to receive raw packets that are not part of any of the supported network protocols. The input and output hooks can also be used to build network monitoring and testing tools.

Normally, the network should be accessed through the higher-level socket interface provided in **sockLib**. The routines in **etherLib** should rarely, if ever, be necessary for applications.

CAVEAT

The following VxWorks network drivers support both the input-hook and output-hook routines:

if_bp – (WRS & SunOS) backplane network driver
if_eex – Intel EtherExpress 16
if_egl – Interphase Eagle 4207 Ethernet driver
if_ei – Intel 82596 ethernet driver
if_elc – SMC 8013WC Ethernet driver
if_elt – 3Com 3C509 Ethernet driver
if_ene – Novell/Eagle NE2000 network driver
if_es – ETHERSTAR ethernet network driver
if_fn – Fujitsu MB86960 NICE Ethernet driver
if_ln – Advanced Micro Devices Am7990 LANCE Ethernet driver
if_lnsgi – AMD Am7990 LANCE Ethernet (for SGI) driver
if_med – Matrix DB-ETH Ethernet network interface driver
if_qu – Motorola MC68EN360 QUICC network interface driver
if_sm – shared memory backplane network interface driver
if_sn – National Semiconductor DP83932B SONIC Ethernet driver
if_ultra – SMC Elite Ultra Ethernet network interface driver

The following drivers support only the input-hook routines:

if_enp – CMC ENP-10 Ethernet driver
if_ex – Excelan EXOS 202 and 302 Ethernet
if_nic – National Semiconductor SNIC Chip (for HKV30)

if_sl – Serial Line IP (SLIP) network interface driver

The following drivers support only the output-hook routines:

if_ulip – network interface driver for User Level IP (VxSim)

The following drivers do not support either the input-hook or output-hook routines:

if_loop – software loopback network interface driver

INCLUDE FILES **etherLib.h**

SEE ALSO *VxWorks Programmer's Guide: Network*

evbNs16550Sio

NAME **evbNs16550Sio** – NS16550 serial driver for the IBM PPC403GA evaluation

SYNOPSIS *evbNs16550HrdInit***( )** – initialize the NS 16550 chip
*evbNs16550Int***( )** – handle a receiver / transmitter interrupt for the NS 16550 chip

```
void evbNs16550HrdInit
    (EVBNS16550_CHAN *pChan)

void evbNs16550Int
    (EVBNS16550_CHAN *pChan)
```

DESCRIPTION This is the driver for the National NS 16550 UART Chip used on the IBM PPC403GA evaluation board. It uses the SCCs in asynchronous mode only.

USAGE An **EVBNS16550_CHAN** structure is used to describe the chip. The BSP's *sysHwInit***( )** routine typically calls *sysSerialHwInit***( )** which initializes all the register values in the **EVBNS16550_CHAN** structure (except the **SIO_DRV_FUNCS**) before calling *evbNs16550HrdInit***( )**. The BSP's *sysHwInit2***( )** routine typically calls *sysSerialHwInit2***( )** which connects the chip interrupt handler *evbNs16550Int***( )** via *intConnect***( )**.

IOCTL FUNCTIONS This driver responds to the same *ioctl***( )** codes as other serial drivers; for more information, see **sioLib.h**.

INCLUDE FILES **drv/sio/evbNs16550Sio.h**

evtBufferLib

NAME **evtBufferLib** – event buffer manipulation library (WindView)

SYNOPSIS
*evtBufferIsEmpty***( )** – check whether the event buffer is empty (WindView)
*evtBufferAddress***( )** – return the address of the event buffer (WindView)
*evtBufferToFile***( )** – transfer the contents of the event buffer to a file (WindView)
*evtBufferUpLoad***( )** – upload the contents of the event buffer to the host (WindView)

```
BOOL evtBufferIsEmpty
     (void)

char * evtBufferAddress
     (void)

STATUS evtBufferToFile
     (char * filename)

void evtBufferUpLoad
     (void)
```

DESCRIPTION This library contains routines that can be used to manipulate the WindView event buffer.

In post-mortem mode, the WindView event buffer continuously overwrites itself until the system fails or is rebooted. At this point, the user can use these routines to manipulate the event buffer.

The event buffer location and size can be configured with the *wvInstInit***( )** routine.

The *evtBufferUpLoad***( )** routine uses the data transfer routine specified by the variable *dataXferRtn* in the *connRtnSet***( )** routine to upload the event buffer to the host. If the default data transfer routine is used, either WindView or **evtRecv** must be running on the host to collect the event data.

INCLUDE FILES **evtBufferLib.h**

SEE ALSO **connLib**, **wvLib**, **evtRecv**, *WindView User's Guide*

excArchLib

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

SYNOPSIS
*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)
*excVecSet***( )** – set a CPU exception vector (PowerPC)
*excVecGet***( )** – get a CPU exception vector (PowerPC)

```
STATUS excVecInit
    (void)

STATUS excConnect
    (VOIDFUNCPTR * vector, VOIDFUNCPTR routine)

STATUS excIntConnect
    (VOIDFUNCPTR * vector, VOIDFUNCPTR routine)

void excVecSet
    (FUNCPTR * vector, FUNCPTR function)

FUNCPTR excVecGet
    (FUNCPTR * vector)
```

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

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

```
STATUS excInit()

void excHookAdd
    (FUNCPTR excepHook)

void excTask()
```

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 on architecture-dependent exception handling facilities, see **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

SYNOPSIS

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)

```
void fioLibInit
    (void)

int printf
    (const char * fmt, ...)

int printErr
    (const char * fmt, ...)

int fdprintf
    (int fd, const char * fmt, ...)

int sprintf
    (char * buffer, const char * fmt, ...)

int vprintf
    (const char * fmt, va_list vaList)

int vfdprintf
    (int fd, const char * fmt, va_list vaList)

int vsprintf
    (char * buffer, const char * fmt, va_list vaList)

int fioFormatV
    (const char *fmt, va_list vaList, FUNCPTR outRoutine, int outarg)

int fioRead
    (int fd, char * buffer, int maxbytes)

int fioRdString
    (int fd, char string[], int maxbytes)
```

```
int sscanf
    (const char * str, const char * fmt, ...)
```

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 used with these routines, the routine *floatInit*( ) must be called first. If the **INCLUDE_FLOATING_POINT** option is defined in **configAll.h**, *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 **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

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

```
void floatInit
    (void)
```

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 **INCLUDE_FLOATING_POINT** is defined in **configAll.h**. If this option is omitted (i.e., *floatInit*( ) is not called), floating-point format specifications in *printf*( ) and *sscanf*( ) will not be supported.

INCLUDE FILES **math.h**

SEE ALSO **fioLib**

fppArchLib

NAME

fppArchLib – architecture-dependent floating-point coprocessor support

SYNOPSIS

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

```
void fppSave
    (FP_CONTEXT * pFpContext)

void fppRestore
    (FP_CONTEXT * pFpContext)

STATUS fppProbe
    (void)

STATUS fppTaskRegsGet
    (int task, FPREG_SET * pFpRegSet)

STATUS fppTaskRegsSet
    (int task, FPREG_SET * pFpRegSet)
```

DESCRIPTION

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

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

INITIALIZATION

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

NOTE X86

On x86 targets, VxWorks disables the six FPU exceptions that can send an IRQ to the CPU.

INCLUDE FILES

fppLib.h

SEE ALSO

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

fppLib

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

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

```
void fppInit
    (void)
```

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 **INCLUDE_HW_FP** is defined in **configAll.h**.

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

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

```
void fppShowInit
    (void)

void fppTaskRegsShow
    (int task)
```

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***( )**. The facility is included automatically when **INCLUDE_SHOW_ROUTINES** is defined in **configAll.h**.

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

SYNOPSIS *ftpdTask***( )** – FTP server daemon task
*ftpdInit***( )** – initialize the FTP server task
*ftpdDelete***( )** – clean up and finalize the FTP server task

```
STATUS ftpdTask
    (void)

STATUS ftpdInit
    (int stackSize)

void ftpdDelete
    (void)
```

DESCRIPTION This library provides File Transfer Protocol (FTP) service to allow an FTP client to store and retrieve files to and from the VxWorks target. The FTP is defined in Requests For

Comments (RFC) document 959, and this library implements an extented subset of this specification. This implementation of the FTP server understands the following FTP requests:

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 – Reserver 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

NOTE: While the Wind River implementation of the FTP server requests a user ID and password from a client, it accepts any ID and password.

The FTP server is initialized by calling *ftpdInit***( )**. This will create a new task, *ftpdTask***( )**. The *ftpdTask***( )** manages multiple FTP client connections, thus it is possible to have multiple FTP sessions running at the same time. For each session, a server task is spawned (**ftpdWorkTask**) to service the client.

The FTP server is shut down by calling *ftpdDelete***( )**. This reclaims all resources allocated by the FTP servers and cleanly terminates all FTP server processes.

This implementation supports all commands suggested by RFC-959 for a minimal FTP server implementation and also several additional commands.

INCLUDE FILES **ftpdLib.h**

SEE ALSO **ftpLib**, **netDrv**, *RFC-959 File Transfer Protocol*

ftpLib

NAME

ftpLib – File Transfer Protocol (FTP) library

SYNOPSIS

*ftpCommand***( )** – send an FTP command and get the reply
*ftpXfer***( )** – initiate a transfer via FTP
*ftpReplyGet***( )** – 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
*ftpDataConnInit***( )** – initialize an FTP data connection
*ftpDataConnGet***( )** – get a completed FTP data connection

```
int ftpCommand
    (int ctrlSock, char *fmt, int arg1, int arg2, int arg3, int arg4,
    int arg5, int arg6)

STATUS ftpXfer
    (char *host, char *user, char *passwd, char *acct, char *cmd,
    char *dirname, char *filename, int *pCtrlSock, int *pDataSock)

int ftpReplyGet
    (int ctrlSock, BOOL expecteof)

int ftpHookup
    (char *host)

STATUS ftpLogin
    (int ctrlSock, char *user, char *passwd, char *account)

int ftpDataConnInit
    (int ctrlSock)

int ftpDataConnGet
    (int dataSock)
```

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

VxWorks provides an I/O driver, **netDrv**, that allows transparent access to remote files via standard I/O system calls. The FTP facilities of **ftpLib** are primarily used by **netDrv** to access remote files. Thus for most purposes, it is not necessary to be familiar with **ftpLib**.

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*( ), and *ftpCommand*( ) 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;
int dataSock = ERROR;
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);
    }
```

INCLUDE FILES

ftpLib.h

SEE ALSO

netDrv

hostLib

NAME

hostLib – host table subroutine library

SYNOPSIS

*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

```
void hostTblInit
    (void)

STATUS hostAdd
    (char *hostName, char *hostAddr)

STATUS hostDelete
    (char *name, char *addr)

int hostGetByName
    (char *name)

STATUS hostGetByAddr
    (int addr, char *name)

int sethostname
    (char *name, int nameLen)

int gethostname
    (char *name, int nameLen)
```

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 a dot (".") 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_NET_INIT** is defined in **configAll.h**.

INCLUDE FILES

hostLib.h

SEE ALSO

inetLib, *VxWorks Programmer's Guide: Network*

i8250Sio

NAME **i8250Sio** – I8250 serial driver

SYNOPSIS *i8250HrdInit***( )** – initialize the chip
*i8250Int***( )** – handle a receiver / transmitter interrupt

```
void i8250HrdInit
    (I8250_CHAN *pChan)

void i8250Int
    (I8250_CHAN *pChan)
```

DESCRIPTION This is the driver for the Intel 8250 UART Chip used on the PC 386. It uses the SCCs in asynchronous mode only.

USAGE An **I8250_CHAN** structure is used to describe the chip. The BSP's *sysHwInit***( )** routine typically calls *sysSerialHwInit***( )** which initializes all the register values in the **I8250_CHAN** structure (except the **SIO_DRV_FUNCS**) before calling *i8250HrdInit***( )**. The BSP's *sysHwInit2***( )** routine typically calls *sysSerialHwInit2***( )** which connects the chips interrupt handler (i8250Int) via *intConnect***( )**.

IOCTL FUNCTIONS This driver responds to all the same *ioctl***( )** codes as a normal serial driver; for more information, see the comments in **sioLib.h**. As initialized, the available baud rates are 110, 300, 600, 1200, 2400, 4800, 9600, 19200, and 38400.

INCLUDE FILES **drv/sio/i8250Sio.h**

ideDrv

NAME **ideDrv** – IDE disk device driver

SYNOPSIS *ideDrv***( )** – initialize the IDE driver
*ideDevCreate***( )** – create a device for a IDE disk
*ideRawio***( )** – provide raw I/O access

```
STATUS ideDrv
    (int vector, int level, BOOL manualConfig)

BLK_DEV *ideDevCreate
    (int drive, int nBlocks, int blkOffset)
```

```
STATUS ideRawio
    (int drive, IDE_RAW * pIdeRaw)
```

DESCRIPTION This is the driver for the IDE used on the PC 386/486.

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: *ideDrv***( )** to initialize the driver, and *ideDevCreate***( )** to create devices.

Before the driver can be used, it must be initialized by calling *ideDrv***( )**. This routine should be called exactly once, before any reads, writes, or calls to *ideDevCreate***( )**. Normally, it is called from *usrRoot***( )** in **usrConfig.c**.

The routine *ideRawio***( )** provides physical I/O access. Its first argument is a drive number, 0 or 1; the second argument is a pointer to an **IDE_RAW** structure.

NOTE Format is not supported, because IDE disks are already formatted, and bad sectors are mapped.

SEE ALSO *VxWorks Programmer's Guide: I/O System*

ifLib

NAME **ifLib** – network interface library

SYNOPSIS

*ifAddrSet***( )** – set 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
*ifunit***( )** – map an interface name to an interface structure pointer

```
STATUS ifAddrSet
    (char *interfaceName, char *interfaceAddress)

STATUS ifAddrGet
    (char *interfaceName, char *interfaceAddress)

STATUS ifBroadcastSet
    (char *interfaceName, char *broadcastAddress)

STATUS ifBroadcastGet
    (char *interfaceName, char *broadcastAddress)

STATUS ifDstAddrSet
    (char *interfaceName, char *dstAddress)

STATUS ifDstAddrGet
    (char *interfaceName, char *dstAddress)

STATUS ifMaskSet
    (char *interfaceName, int netMask)

STATUS ifMaskGet
    (char *interfaceName, int *netMask)

STATUS ifFlagChange
    (char *interfaceName, int flags, BOOL on)

STATUS ifFlagSet
    (char *interfaceName, int flags)

STATUS ifFlagGet
    (char *interfaceName, int *flags)

STATUS ifMetricSet
    (char *interfaceName, int metric)

STATUS ifMetricGet
    (char *interfaceName, int *pMetric)

int ifRouteDelete
    (char *ifName, int unit)

struct ifnet *ifunit
    (char *ifname)
```

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**.

INCLUDE FILES **ifLib.h**

SEE ALSO **hostLib**, *VxWorks Programmer's Guide: Network*

if_bp

NAME

if_bp – original VxWorks (and SunOS) backplane network interface driver

SYNOPSIS

*bpattach***( )** – publish the **bp** network interface and initialize the driver and device
*bpInit***( )** – initialize the backplane anchor
*bpShow***( )** – display information about the backplane network

```
STATUS bpattach
    (int unit, char *pAnchor, int procNum, int intType, int intArg1,
    int intArg2, int intArg3)

STATUS bpInit
    (char *pAnchor, char *pMem, int memSize, BOOL tasOK)

void bpShow
    (char *bpName, BOOL zero)
```

DESCRIPTION

This module implements the original VxWorks backplane network interface driver.

The backplane driver allows several CPUs to communicate via shared memory. Usually, the first CPU to boot in a card cage is considered the backplane master. This CPU has either dual-ported memory or an additional memory board which all other CPUs can access. Each CPU must be interruptible by another CPU, as well as be able to interrupt all other CPUs. There are three interrupt types: polling, mailboxes, and VMEbus interrupts. Polling is used when there are no hardware interrupts; each CPU spawns a polling task to manage transfers. Mailbox interrupts are the preferred method, because they do not require an interrupt level. VMEbus interrupts offer better performance than polling; however, they may require hardware jumpers.

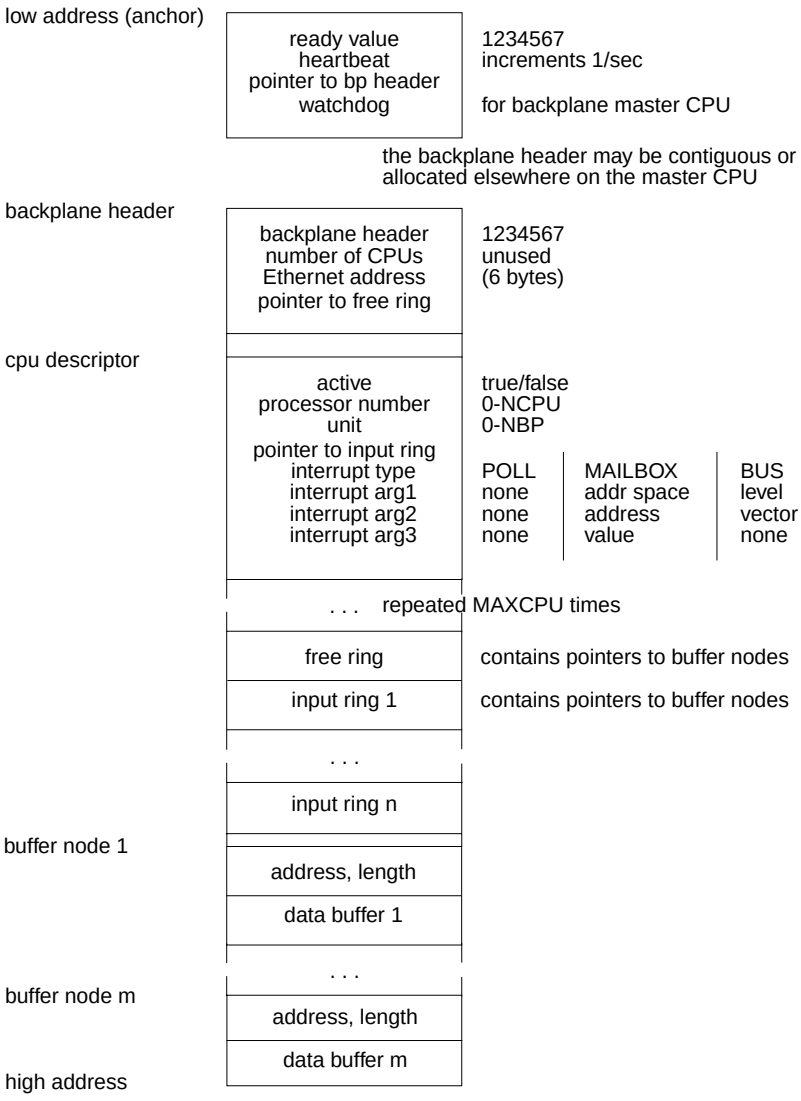
There are three user-callable routines: *bpInit***( )**, *bpattach***( )**, and *bpShow***( )**. The backplane master, usually processor 0, must initialize the backplane memory and structures by first calling *bpInit***( )**. Once the backplane has been initialized, all processors can be attached via *bpattach***( )**. Usually, *bpInit***( )** and *bpattach***( )** are called automatically in **usrConfig.c** when backplane parameters have been specified in the boot line.

The *bpShow***( )** routine is provided as a diagnostic aid to show all the CPUs configured on a backplane.

For more information about SunOS use of this driver, see the *Guide to the VxWorks Backplane Driver for SunOS*.

This driver has been replaced by **if_sm** and is included here for backwards compatibility.

MEMORY LAYOUT All pointers in shared memory are relative to the start of shared memory, since dual-ported memory may appear in different places on different CPUs.



BOARD LAYOUT This device is “software only.” A jumpering diagram is not applicable.

SEE ALSO **ifLib**

if_cpm

NAME

if_cpm – Motorola CPM core network interface driver

SYNOPSIS

*cpmattach***( )** – publish the **cpm** network interface and initialize the driver

```
STATUS cpmattach
    (int unit, SCC * pScc, SCC_REG * pSccReg, VOIDFUNCPTR * ivec,
    SCC_BUF * txBdBase, SCC_BUF * rxBdBase, int txBdNum, int rxBdNum,
    UINT8 * bufBase)
```

DESCRIPTION

This module implements the driver for the Motorola CPM core Ethernet network interface used in the M68EN360 and PPC800-series communications controllers.

The driver is designed to support the Ethernet mode of an SCC residing on the CPM processor core. It is generic in the sense that it does not care which SCC is being used, and it supports up to four individual units per board.

The driver must be given several target-specific parameters, and some external support routines must be provided. These parameters, and the mechanisms used to communicate them to the driver, are detailed below.

This network interface driver does not include support for trailer protocols or data chaining. However, buffer loaning has been implemented in an effort to boost performance. This driver provides support for four individual device units.

This driver maintains cache coherency by allocating buffer space using the *cacheDmaMalloc***( )** routine. It is assumed that cache-safe memory is returned; this driver does not perform cache flushing and invalidating.

BOARD LAYOUT

This device is on-chip. No jumpering diagram is necessary.

EXTERNAL INTERFACE

This driver presents the standard WRS network driver API: the device unit must be attached and initialized with the *cpmattach***( )** routine.

The only user-callable routine is *cpmattach***( )**, which publishes the **cpm** interface and initializes the driver structures.

TARGET-SPECIFIC PARAMETERS

These parameters are passed to the driver via *cpmattach***( )**.

address of SCC parameter RAM

This parameter is the address of the parameter RAM used to control the SCC. Through this address, and the address of the SCC registers (see below), different network interface units are able to use different SCCs without conflict. This parameter points to the internal memory of the chip where the SCC physically

resides, which may not necessarily be the master chip on the target board.

address of SCC registers
: This parameter is the address of the registers used to control the SCC. Through this address, and the address of the SCC parameter RAM (see above), different network interface units are able to use different SCCs without conflict. This parameter points to the internal memory of the chip where the SCC physically resides, which may not necessarily be the master chip on the target board.

interrupt-vector offset
: This driver configures the SCC to generate hardware interrupts for various events within the device. The interrupt-vector offset parameter is used to connect the driver's ISR to the interrupt through a call to *intConnect***( )**.

address of transmit and receive buffer descriptors
: These parameters indicate the base locations of the transmit and receive buffer descriptor (BD) rings. Each BD takes up 8 bytes of dual-ported RAM, and it is the user's responsibility to ensure that all specified BDs will fit within dual-ported RAM. This includes any other BDs the target board may be using, including other SCCs, SMCs, and the SPI device. There is no default for these parameters; they must be provided by the user.

number of transmit and receive buffer descriptors
: The number of transmit and receive buffer descriptors (BDs) used is configurable by the user upon attaching the driver. Each buffer descriptor resides in 8 bytes of the chip's dual-ported RAM space, and each one points to a 1520-byte buffer in regular RAM. There must be a minimum of two transmit and two receive BDs. There is no maximum number of buffers, but there is a limit to how much the driver speed increases as more buffers are added, and dual-ported RAM space is at a premium. If this parameter is "NULL", a default value of 32 BDs is used.

base address of buffer pool
: This parameter is used to notify the driver that space for the transmit and receive buffers need not be allocated, but should be taken from a cache-coherent private memory space provided by the user at the given address. The user should be aware that memory used for buffers must be 4-byte aligned and non-cacheable. All the buffers must fit in the given memory space; no checking is performed. This includes all transmit and receive buffers (see above) and an additional 16 receive loaner buffers. If the number of receive BDs is less than 16, that number of loaner buffers is used. Each buffer is 1520 bytes. If this parameter is "NONE," space for buffers is obtained by calling *cacheDmaMalloc***( )** in *cpmattach***( )**.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires seven external support functions:

`STATUS sysCpmEnetEnable (int unit)`
: This routine is expected to perform any target-specific functions required to enable the Ethernet controller. These functions typically include enabling the

Transmit Enable signal (TENA) and connecting the transmit and receive clocks to the SCC. The driver calls this routine, once per unit, from the *cmpInit*( ) routine.

`void sysCpmEnetDisable (int unit)`
This routine is expected to perform any target-specific functions required to disable the Ethernet controller. This usually involves disabling the Transmit Enable (TENA) signal. The driver calls this routine from the *cpmReset*( ) routine each time a unit is disabled.

`STATUS sysCpmEnetCommand (int unit, UINT16 command)`
This routine is expected to issue a command to the Ethernet interface controller. The driver calls this routine to perform basic commands, such as restarting the transmitter and stopping reception.

`void sysCpmEnetIntEnable (int unit)`
This routine is expected to enable the interrupt for the Ethernet interface specified by *unit*.

`void sysCpmEnetIntDisable (int unit)`
This routine is expected to disable the interrupt for the Ethernet interface specified by *unit*.

`void sysCpmEnetIntClear (int unit)`
This routine is expected to clear the interrupt for the Ethernet interface specified by *unit*.

`STATUS sysCpmEnetAddrGet (int unit, UINT8 * addr)`
The driver expects this routine to provide the 6-byte Ethernet hardware address that will be used by *unit*. This routine must copy the 6-byte address to the space provided by *addr*. This routine is expected to return OK on success, or ERROR. The driver calls this routine, once per unit, from the *cpmInit*( ) routine.

SYSTEM RESOURCE USAGE

This driver requires the following system resources:

– one mutual exclusion semaphore
– one interrupt vector
– 0 bytes in the initialized data section (data)
– 1272 bytes in the uninitialized data section (BSS)

The data and BSS sections are quoted for the CPU32 architecture and may vary for other architectures. The code size (text) varies greatly between architectures, and is therefore not quoted here.

If the driver allocates the memory shared with the Ethernet device unit, it does so by calling the *cacheDmaMalloc*( ) routine. For the default case of 32 transmit buffers, 32 receive buffers, and 16 loaner buffers, the total size requested is 121,600 bytes. If a non-cacheable memory region is provided by the user, the size of this region should be this amount, unless the user has specified a different number of transmit or receive BDs.

This driver can operate only if the shared memory region is non-cacheable, or if the hardware implements bus snooping. The driver cannot maintain cache coherency for the device because the buffers are asynchronously modified by both the driver and the device, and these fields may share the same cache line. Additionally, the chip's dual ported RAM must be declared as non-cacheable memory where applicable.

SEE ALSO **ifLib**, *Motorola MC68EN360 User's Manual*, *Motorola MPC860 User's Manual*, *Motorola MPC821 User's Manual*

if_dc

NAME **if_dc** – DEC 21040 PCI Ethernet LAN network-interface driver

SYNOPSIS *dcattach***( )** – publish the **dc** network interface

```
STATUS dcattach
    (int unit, ULONG devAdrs, int ivec, int ilevel, char * memAdrs,
    ULONG memSize, int memWidth, ULONG pciMemBase, int dcOpMode)
```

DESCRIPTION This library is the DEC 21040-PCI Ethernet 32-bit network-interface driver.

The DEC 21040-PCI Ethernet controller is inherently little-endian because the chip is designed to operate on a PCI bus which is a little-endian bus. The software interface to the driver is divided into three parts. The first part is the PCI configuration registers and their set up. This part is done at the BSP level in the various BSPs which use this driver. The second and third part are dealt with in the driver. The second part of the interface consists of the I/O control registers and their programming. The third part of the interface consists of the descriptors and buffers.

This driver is designed to be moderately generic, operating unmodified across the range of architectures and targets supported by VxWorks. To achieve this, the driver must be given several target-specific parameters, and some external support routines must be provided. These parameters and the mechanisms used to communicate them to the driver are detailed below. If any of the assumptions stated below are not true for your particular hardware, this driver will probably not function correctly with it.

This driver supports up to 4 LANCE units per CPU. The driver can be configured to support big-endian or little-endian architectures. It contains error-recovery code to handle known device errata related to DMA activity.

This driver configures the 10BASE-T interface by default and waits for two seconds to check the status of the link. If the link status is "fail,"it then configures the AUI interface.

Big-endian processors can be connected to the PCI bus through some controllers which take care of hardware byte swapping. In such cases all the registers to which the chip performs DMAs must be swapped and written to. Then when the hardware swaps the accesses, the chip sees them correctly. The chip still must be programmed to operate in little-endian mode as it is on the PCI bus. If the CPU board hardware automatically swaps all the accesses to and from the PCI bus, then the input and output byte streams need not be swapped.

BOARD LAYOUT

This device is on-board. No jumpering diagram is necessary.

EXTERNAL INTERFACE

This driver provides the standard external interface with the following exceptions:

– All initialization is performed within the attach routine; there is no separate initialization routine. Therefore, in the global-interface structure, the function pointer to the initialization routine is NULL.

– The only user-callable routine is *dcattach***( )**, which publishes the **dc** interface and initializes the driver and device.

TARGET-SPECIFIC PARAMETERS

bus mode

This parameter is a global variable that can be modified at run-time.

The LAN control register #0 determines the bus mode of the device, allowing the support of big-endian and little-endian architectures. This parameter, defined as "ULONG dcCSR0Bmr", is the value that will be placed into LANCE control register #0. The default is mode is little-endian. For information about changing this parameter, see the manual *DEC Local Area Network Controller DEC21040 for PCI*.

base address of device registers

This parameter is passed to the driver by *dcattach***( )**.

interrupt vector

This parameter is passed to the driver by *dcattach***( )**.

This driver configures the LANCE device to generate hardware interrupts for various events within the device; thus it contains an interrupt handler routine. The driver calls *intConnect***( )** to connect its interrupt handler to the interrupt vector generated as a result of the LANCE interrupt.

interrupt level

This parameter is passed to the driver by *dcattach***( )**.

Some targets use additional interrupt-controller devices to help organize and service the various interrupt sources. This driver avoids all board-specific knowledge of such devices. During the driver's initialization, the external routine *sysLanIntEnable***( )** is called to perform any board-specific operations required to

allow the servicing of a LANCE interrupt. For a description of *sysLanIntEnable***( )**, see "External Support Requirements" below.

This parameter is passed to the external routine.

shared memory address

This parameter is passed to the driver by *dcattach***( )**.

The LANCE device is a DMA device and typically shares access to some region of memory with the CPU. This driver is designed for systems that directly share memory between the CPU and the LANCE. It assumes that this shared memory is directly available to it without any arbitration or timing concerns.

This parameter can be used to specify an explicit memory region for use by the LANCE. This should be done on hardware that restricts the LANCE to a particular memory region. The constant NONE can be used to indicate that there are no memory limitations, in which case the driver attempts to allocate the shared memory from the system space.

shared memory size

This parameter is passed to the driver by *dcattach***( )**.

This parameter can be used to limit explicitly the amount of shared memory (bytes) this driver uses. The constant NONE can be used to indicate no specific size limitation. This parameter is used only if a specific memory region is provided to the driver.

shared memory width

This parameter is passed to the driver by *dcattach***( )**.

Some target hardware that restricts the shared memory region to a specific location also restricts the access width to this region by the CPU. On these targets, performing an access of an invalid width will cause a bus error.

This parameter can be used to specify the number of bytes of access width to be used by the driver during access to the shared memory. The constant NONE can be used to indicate no restrictions.

Current internal support for this mechanism is not robust; implementation may not work on all targets requiring these restrictions.

shared memory buffer size

This parameter is passed to the driver by *dcattach***( )**.

The driver and LANCE device exchange network data in buffers. This parameter permits the size of these individual buffers to be limited. A value of zero indicates that the default buffer size should be used. The default buffer size is large enough to hold a maximum-size Ethernet packet.

pci Memory base

This parameter is passed to the driver by *dcattach***( )**. This parameter gives the base address of the main memory on the PCI bus.

dcOpMode
: This parameter is passed to the driver by *dcattach***( )**. This parameter gives the mode of initialization of the device. The mode flags are listed below.

  DC_PROMISCUOUS_FLAG 0x01

  DC_MULTICAST_FLAG 0x02

Ethernet address
: This is obtained by the driver by reading an Ethernet ROM register interfaced with the device.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires one external support function:

`void sysLanIntEnable (int level)`
: This routine provides a target-specific enable of the interrupt for the LANCE device. Typically, this involves interrupt-controller hardware, either internal or external to the CPU.

  This routine is called once, from the *dcattach***( )** routine.

SEE ALSO

ifLib, *DECchip 21040 Ethernet LAN Controller for PCI.*

if_eex

NAME

if_eex – Intel EtherExpress 16 network interface driver

SYNOPSIS

*eexattach***( )** – publish the **eex** network interface and initialize the driver and device

```
STATUS eexattach
    (int unit, int port, int ivec, int ilevel, int nTfds, int attachment)
```

DESCRIPTION

This module implements the Intel EtherExpress 16 PC network interface card driver. It is specific to that board as used in PC 386/486 hosts. This driver is written using the device's I/O registers exclusively.

SIMPLIFYING ASSUMPTIONS

This module assumes a little-endian host (80x86); thus, no endian adjustments are needed to manipulate the 82586 data structures (little-endian).

The on-board memory is assumed to be sufficient; thus, no provision is made for additional buffering in system memory.

The “frame descriptor” and “buffer descriptor” structures can be bound into permanent pairs by pointing each FD at a “chain” of one BD of MTU size. The 82586 receive algorithm fills exactly one BD for each FD; it looks to the NEXT FD in line for the next BD.

The transmit and receive descriptor lists are permanently linked into circular queues partitioned into sublists designated by the **EEX_LIST** headers in the driver control structure. Empty partitions have NULL pointer fields. EL bits are set as needed to tell the 82586 where a partition ends. The lists are managed in strict FIFO fashion; thus the link fields are never modified, just ignored if a descriptor is at the end of a list partition.

BOARD LAYOUT

This device is soft-configured. No jumpering diagram is required.

EXTERNAL INTERFACE

This driver provides the standard external interface with the following exceptions. All initialization is performed within the attach routine and there is no separate initialization routine. Therefore, in the global interface structure, the function pointer to the *init***( )** routine is NULL.

There is one user-callable routine, *eexattach***( )**. For details on usage, see the manual entry for this routine.

EXTERNAL SUPPORT REQUIREMENTS

None.

SYSTEM RESOURCE USAGE

– one mutual exclusion semaphore
– one interrupt vector
– one watchdog timer.
– 8 bytes in the initialized data section (data)
– 912 bytes in the uninitialized data section (bss)

The data and bss sections are quoted for the MC68020 architecture and may vary for other architectures. The code size (text) will vary widely between architectures, and is thus not quoted here.

The device contains on-board buffer memory; no system memory is required for buffering.

TUNING HINTS

The only adjustable parameter is the number of TFDs to create in adapter buffer memory. The total number of TFDs and RFDs is 21, given full-frame buffering and the sizes of the auxiliary structures. *eexattach***( )** requires at least **MIN_NUM_RFDS** RFDs to exist. More than ten TFDs is not sensible in typical circumstances.

SEE ALSO

ifLib

if_ei

NAME **if_ei** – Intel 82596 Ethernet network interface driver

SYNOPSIS *eiattach***( )** – publish the **ei** network interface and initialize the driver and device

```
STATUS eiattach
    (int unit, int ivec, UINT8 sysbus, char * memBase, int nTfds, int nRfds)
```

DESCRIPTION This module implements the Intel 82596 Ethernet network interface driver.

This driver is designed to be moderately generic, operating unmodified across the range of architectures and targets supported by VxWorks. To achieve this, this driver must be given several target-specific parameters, and some external support routines must be provided. These parameters, and the mechanisms used to communicate them to the driver, are detailed below.

This driver can run with the device configured in either big-endian or little-endian modes. Error recovery code has been added to deal with some of the known errata in the A0 version of the device. This driver supports up to four individual units per CPU.

BOARD LAYOUT This device is on-board. No jumpering diagram is necessary.

EXTERNAL INTERFACE

This driver provides the standard external interface with the following exceptions. All initialization is performed within the attach routine; there is no separate initialization routine. Therefore, in the global interface structure, the function pointer to the initialization routine is NULL.

The only user-callable routine is *eiattach***( )**, which publishes the **ei** interface and initializes the driver and device.

TARGET-SPECIFIC PARAMETERS

the *sysbus* value

This parameter is passed to the driver by *eiattach***( )**.

The Intel 82596 requires this parameter during initialization. This parameter tells the device about the system bus, hence the name “sysbus.”To determine the correct value for a target, refer to the document *Intel 32-bit Local Area Network (LAN) Component User’s Manual.*

interrupt vector

This parameter is passed to the driver by *eiattach***( )**.

The Intel 82596 generates hardware interrupts for various events within the device; thus it contains an interrupt handler routine. This driver calls *intConnect***( )** to connect its interrupt handler to the interrupt vector generated as a result of the 82596 interrupt.

shared memory address

This parameter is passed to the driver by *eiattach***( )**.

The Intel 82596 device is a DMA type device and typically shares access to some region of memory with the CPU. This driver is designed for systems that directly share memory between the CPU and the 82596.

This parameter can be used to specify an explicit memory region for use by the 82596. This should be done on targets that restrict the 82596 to a particular memory region. The constant NONE can be used to indicate that there are no memory limitations, in which case, the driver attempts to allocate the shared memory from the system space.

number of Receive and Transmit Frame Descriptors

These parameters are passed to the driver by *eiattach***( )**.

The Intel 82596 accesses frame descriptors in memory for each frame transmitted or received. The number of frame descriptors at run-time can be configured using these parameters.

Ethernet address

This parameter is obtained by a call to an external support routine.

During initialization, the driver needs to know the Ethernet address for the Intel 82596 device. The driver calls the external support routine, *sysEnetAddrGet***( )**, to obtain the Ethernet address. For a description of *sysEnetAddrGet***( )**, see "External Support Requirements" below.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires seven external support functions:

`STATUS sysEnetAddrGet (int unit, char *pCopy)`

This routine provides the six-byte Ethernet address used by *unit*. It must copy the six-byte address to the space provided by *pCopy*. This routine returns OK, or ERROR if it fails. The driver calls this routine, once per unit, using *eiattach***( )**.

`STATUS sys596Init (int unit)`

This routine performs any target-specific initialization required before the 82596 is initialized. Typically, it is empty. This routine must return OK, or ERROR if it fails. The driver calls this routine, once per unit, using *eiattach***( )**.

`void sys596Port (int unit, int cmd, UINT32 addr)`

This routine provides access to the special port function of the 82596. It delivers the command and address arguments to the port of the specified unit. The driver calls this routine primarily during initialization, but may also call it during error recovery procedures.

`void sys596ChanAtn (int unit)`

This routine provides the channel attention signal to the 82596, for the specified *unit*. The driver calls this routine frequently throughout all phases of operation.

`void sys596IntEnable (int unit), void sys596IntDisable (int unit)`
These routines enable or disable the interrupt from the 82596 for the specified *unit*. Typically, this involves interrupt controller hardware, either internal or external to the CPU. Since the 82596 itself has no mechanism for controlling its interrupt activity, these routines are vital to the correct operation of the driver. The driver calls these routines throughout normal operation to protect certain critical sections of code from interrupt handler intervention.

`void sys596IntAck (int unit)`
This routine must perform any required interrupt acknowledgment or clearing. Typically, this involves an operation to some interrupt control hardware. Note that the INT signal from the 82596 behaves in an "edge-triggered" mode; therefore, this routine typically clears a latch within the control circuitry. The driver calls this routine from the interrupt handler.

SYSTEM RESOURCE USAGE

When implemented, this driver requires the following system resources:

– one mutual exclusion semaphore
– one interrupt vector
– one watchdog timer.
– 8 bytes in the initialized data section (data)
– 912 bytes in the uninitialized data section (BSS)

The above data and BSS requirements are for the MC68020 architecture and may vary for other architectures. Code size (text) varies greatly between architectures and is therefore not quoted here.

The driver uses *cacheDmaMalloc***( )** to allocate memory to share with the 82596. The fixed-size pieces in this area total 160 bytes. The variable-size pieces in this area are affected by the configuration parameters specified in the *eiattach***( )** call. The size of one RFD (Receive Frame Descriptor) is 1536 bytes. The size of one TFD (Transmit Frame Descriptor) is 1534 bytes. For more information about RFDs and TFDs, see the *Intel 82596 User's Manual.*

The 82596 can be operated only if this shared memory region is non-cacheable or if the hardware implements bus snooping. The driver cannot maintain cache coherency for the device because fields within the command structures are asynchronously modified by both the driver and the device, and these fields may share the same cache line.

TUNING HINTS

The only adjustable parameters are the number of TFDs and RFDs that will be created at run-time. These parameters are given to the driver when *eiattach***( )** is called. There is one TFD and one RFD associated with each transmitted frame and each received frame respectively. For memory-limited applications, decreasing the number of TFDs and RFDs may be desirable. Increasing the TFDs provides no performance benefit after a certain point. Increasing the RFDs provides more buffering before packets are dropped. This can be useful if there are tasks running at a higher priority than the net task.

SEE ALSO

ifLib, *Intel 82596 User's Manual, Intel 32-bit Local Area Network (LAN) Component Manual*

if_eitp

NAME **if_eitp** – Intel 82596 Ethernet network interface driver for the TP41V

SYNOPSIS *eitpattach***( )** – publish the **ei** network interface for the TP41V and initialize the driver and device

```
STATUS eitpattach
    (int unit, int ivec, UINT8 sysbus, char * memBase, int nTfds, int nRfds)
```

DESCRIPTION This module implements the Intel 82596 Ethernet network interface driver.

This driver is a custom version of the generic **ei** driver, to support the Tadpole TP41V. Use the information from manual page for **if_ei**, except for the following differences:

– The name of the attach routine is *eitpattach***( )**. The arguments are the same.

– The following external support routines are not required: *sys596Init***( )**, *sys596IntEnable***( )**, *sys596IntDisable***( )**, and *sys596IntAck***( )**.

SEE ALSO **if_ei**, **ifLib**, *Intel 82596 User's Manual*

if_elc

NAME **if_elc** – SMC 8013WC Ethernet network interface driver

SYNOPSIS *elcattach***( )** – publish the **elc** network interface and initialize the driver and device
*elcShow***( )** – display statistics for the SMC 8013WC **elc** network interface

```
STATUS elcattach
    (int unit, int ioAddr, int ivec, int ilevel, int memAddr, int memSize,
    int config)
```

```
void elcShow
    (int unit, BOOL zap)
```

DESCRIPTION This module implements the SMC 8013WC network interface driver.

BOARD LAYOUT The W1 jumper should be set in position SOFT. The W2 jumper should be set in position NONE/SOFT.

CONFIGURATION

The I/O address, RAM address, RAM size, and IRQ levels are defined in **config.h**. The I/O address must match the one stored in EEROM. The configuration software supplied by the manufacturer should be used to set the I/O address.

IRQ levels 2,3,4,5,7,9,10,11,15 are supported. Thick Ethernet (AUI) and Thin Ethernet (BNC) are configurable by changing the macro **CONFIG_ELC** in **config.h**.

EXTERNAL INTERFACE

The only user-callable routines are *elcattach***( )** and *elcShow***( )**:

*elcattach***( )**
publishes the **elc** interface and initializes the driver and device.

*elcShow***( )**
displays statistics that are collected in the interrupt handler.

if_elt

NAME

if_elt – 3Com 3C509 Ethernet network interface driver

SYNOPSIS

*eltattach***( )** – publish the **elt** interface and initialize the driver and device
*eltShow***( )** – display statistics for the 3C509 **elt** network interface

```
STATUS eltattach
    (int unit, int port, int ivec, int intLevel, int nRxFrames,
    int attachment, char *ifName)

void eltShow
    (int unit, BOOL zap)
```

DESCRIPTION

This module implements the 3Com 3C509 network adapter driver.

The 3C509 (EtherLink® III) is not well-suited for use in real-time systems. Its meager on-board buffering (4K total; 2K transmit, 2K receive) forces the host processor to service the board at a high priority. 3Com makes a virtue of this necessity by adding fancy lookahead support and adding the label “Parallel Tasking” to the outside of the box. Using 3Com’s drivers, this board will look good in benchmarks that measure raw link speed. The board is greatly simplified by using the host CPU as a DMA controller.

BOARD LAYOUT

This device is soft-configured by a DOS-hosted program supplied by the manufacturer. No jumpering diagram is required.

EXTERNAL INTERFACE

This driver provides the standard external interface with the following exceptions. All initialization is performed within the attach routine and there is no separate initialization routine. Thus, in the global interface structure, the function pointer to the initialization routine is NULL.

There are two user-callable routines:

*eltattach***( )**
publishes the **elt** interface and initializes the driver and device.

*eltShow***( )**
displays statistics that are collected in the interrupt handler.

See the manual entries for these routines for more detail.

SYSTEM RESOURCE USAGE

– one mutual exclusion semaphore
– one interrupt vector
– 16 bytes in the uninitialized data section (bss)
– 180 bytes (plus overhead) of malloc'ed memory per unit
– 1530 bytes (plus overhead) of malloc'ed memory per frame buffer, minimum 5 frame buffers.

SHORTCUTS

The EISA and MCA versions of the board are not supported.

Attachment selection assumes the board is in power-on reset state; a warm restart will not clear the old attachment selection out of the hardware, and certain new selections may not clear it either. For example, if RJ45 was selected, the system is warm-booted, and AUI is selected, the RJ45 connector is still functional.

Attachment type selection is not validated against the board's capabilities, even though there is a register that describes which connectors exist.

The loaned buffer cluster type is **MC_EI**; no new type is defined yet.

Although it seems possible to put the transmitter into a non-functioning state, it is not obvious either how to do this or how to detect the resulting state. There is therefore no transmit watchdog timer.

No use is made of the tuning features of the board; it is possible that proper dynamic tuning would reduce or eliminate the receive overruns that occur when receiving under task control (instead of in the ISR).

TUNING HINTS

More receive buffers (than the default 20) could help by allowing more loaning in cases of massive reception; four per receiving TCP connection plus four extras should be considered a minimum.

SEE ALSO

ifLib

if_ene

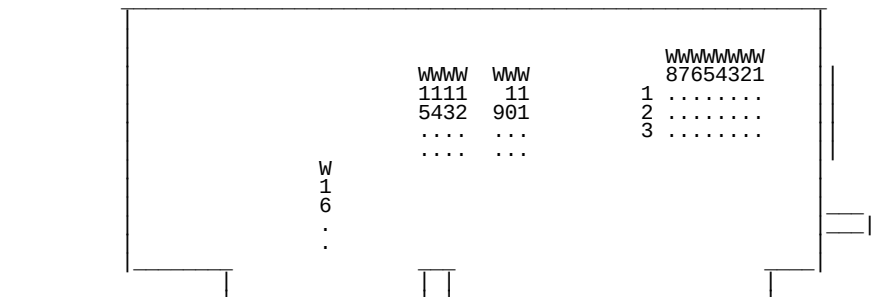
NAME **if_ene** – Novell/Eagle NE2000 network interface driver

SYNOPSIS *eneattach***( )** – publish the **ene** network interface and initialize the driver and device
*eneShow***( )** – display statistics for the NE2000 **ene** network interface

```
STATUS eneattach
    (int unit, int ioAddr, int ivec, int ilevel)

void eneShow
    (int unit, BOOL zap)
```

DESCRIPTION This module implements the Novell/Eagle NE2000 network interface driver. There is one user-callable routine, *eneattach***( )**.

BOARD LAYOUT The diagram below shows the relevant jumpers for VxWorks configuration. Other compatible boards will be jumpered differently; many are jumperless.



```
    W1..W8  1-2 position selects AUI ("DIX") connector
            2-3 position selects BNC (10BASE2) connector
    W9..W11 YYN  I/O address 300h, no boot ROM
            NYN  I/O address 320h, no boot ROM
            YNN  I/O address 340h, no boot ROM
            NNN  I/O address 360h, no boot ROM
            YYY  I/O address 300h, boot ROM at paragraph 0c800h
            NYY  I/O address 320h, boot ROM at paragraph 0cc00h
            YNY  I/O address 340h, boot ROM at paragraph 0d000h
            NNY  I/O address 360h, boot ROM at ??? (invalid configuration?)
    W12     Y    IRQ 2 (or 9 if you prefer)
    W13     Y    IRQ 3
    W14     Y    IRQ 4
    W15     Y    IRQ 5 (note that only one of W12..W15 may be installed)
    W16     Y    normal ISA bus timing
            N    timing for COMPAQ 286 portable, PS/2 Model 30-286, C&T
chipset
```

EXTERNAL INTERFACE

There are two user-callable routines:

*eneattach***( )**
publishes the **ene** interface and initializes the driver and device.

*eneShow***( )**
displays statistics that are collected in the interrupt handler.

See the manual entries for these routines for more detail.

SYSTEM RESOURCE USAGE

– one interrupt vector
– 16 bytes in the uninitialized data section (bss)
– 1752 bytes (plus overhead) of malloc'ed memory per unit attached

CAVEAT

This driver does not enable the twisted-pair connector on the Taiwanese ETHER-16 compatible board.

if_enp

NAME

if_enp – CMC ENP 10/L Ethernet network interface driver

SYNOPSIS

*enpattach***( )** – publish the **enp** network interface and initialize the driver and device

```
STATUS enpattach
    (int unit, char *addr, int ivec, int ilevel, int enpAddrAm)
```

DESCRIPTION

This module implements the CMC ENP 10/L Ethernet network interface driver.

This driver is designed to be moderately generic, operating unmodified across the range of architectures and targets supported by VxWorks. To achieve this, the driver must be given several target-specific parameters. These parameters, and the mechanisms used to communicate them to the driver, are detailed below. This driver supports up to four individual units per CPU.

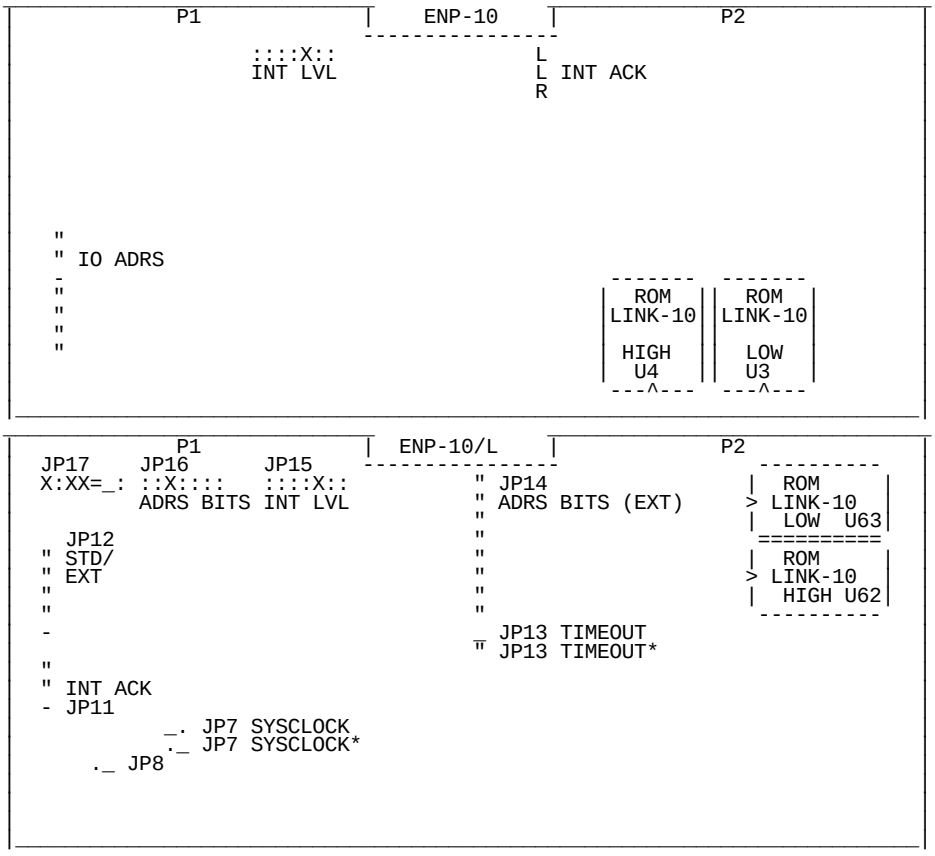
BOARD LAYOUT

The diagrams below show the relevant jumpers for VxWorks configuration. Default values are: I/O address 0x00de0000, Standard Addressing (A24), interrupt level 3.

EXTERNAL INTERFACE

This driver provides the standard external interface with the following exceptions. All initialization is performed within the attach routine; there is no separate initialization routine. Therefore, in the global interface structure, the function pointer to the initialization routine is NULL.

The only user-callable routine is *enpattach***( )**, which publishes the **enp** interface and initializes the driver and device.



TARGET-SPECIFIC PARAMETERS

base VME address of ENP memory

This parameter is passed to the driver by *enpattach***( )**. The ENP board presents an area of its memory to the VME bus. This address is jumper selectable on the ENP. This parameter is the same as the address selected on the ENP.

VME address modifier code

This parameter is passed to the driver by *enpattach***( )**. It specifies the AM (address modifier) code to use when the driver accesses the VME address space of the ENP board.

interrupt vector

This parameter is passed to the driver by *enpattach***( )**. It specifies the interrupt vector to be used by the driver to service an interrupt from the ENP board. The driver will connect the interrupt handler to this vector by calling *intConnect***( )**.

interrupt level

This parameter is passed to the driver by *enpattach***( )**. It specifies the interrupt level that is associated with the interrupt vector. The driver enables the interrupt from the ENP by calling *sysIntEnable***( )** and passing this parameter.

SYSTEM RESOURCE USAGE

When implemented, this driver requires the following system resources:

– one mutual exclusion semaphore
– one interrupt vector
– 76 bytes in the initialized data section (data)
– 808 bytes in the uninitialized data section (BSS)

The above data and BSS requirements are for the MC68020 architecture and may vary for other architectures. Code size (text) varies greatly between architectures and is therefore not quoted here.

The ENP board provides the buffers for all packets transmitted and received. Therefore, the driver does not require any system memory to share with the device. This also eliminates all data cache coherency issues.

SEE ALSO **ifLib**

---

if_ex

NAME **if_ex** – Excelan EXOS 201 / 202 / 302 Ethernet network interface driver

SYNOPSIS *exattach***( )** – publish the **ex** network interface and initialize the driver and device

```
STATUS exattach
    (int unit, char *pDev, int ivec, int ilevel, int exDmaAm, int exAdrsAm)
```

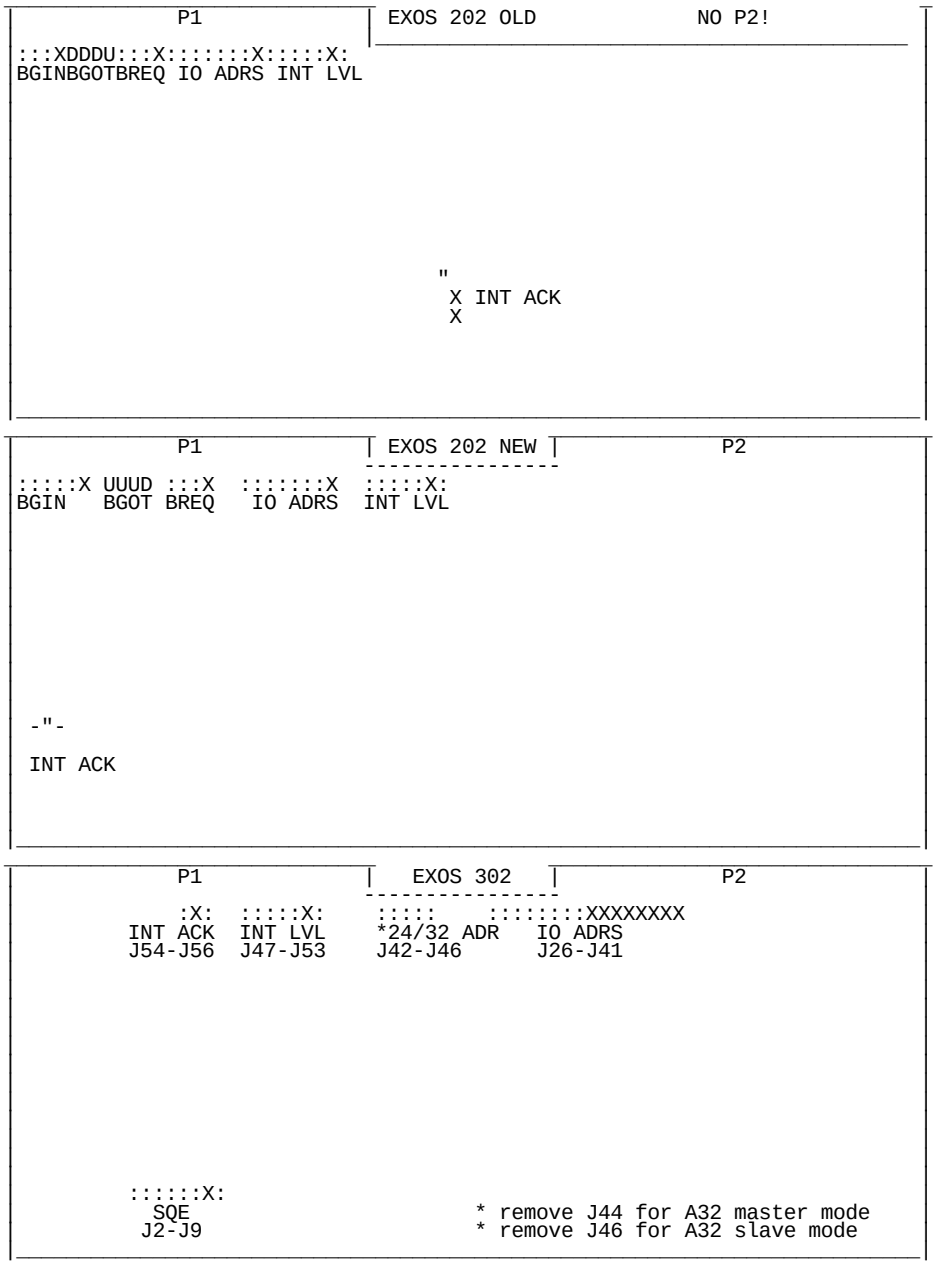
DESCRIPTION This module implements the Excelan EXOS 201 / 202 / 302 Ethernet interface driver.

This driver is designed to be moderately generic, operating unmodified across the range of architectures and targets supported by VxWorks. To achieve this, the driver must be given several target-specific parameters. These parameters, and the mechanisms used to communicate them to the driver, are detailed below. This driver supports up to four individual units per CPU.

All packet control information and data moves between the EXOS board and the target board in a shared memory region. This shared memory must reside on the target board, since the EXOS board does not present any memory to the VME bus. Therefore, this driver will obtain an area of local memory, and assumes that this area of memory can be accessed from the VME bus.

BOARD LAYOUT

The diagram below shows the relevant jumpers for VxWorks configuration. Default values are: I/O address 0x00ff0000, Standard Addressing (A24), interrupt level 2.



EXTERNAL INTERFACE

This driver provides the standard external interface with the following exceptions. All initialization is performed within the attach routine; there is no separate initialization routine. Therefore, in the global interface structure, the function pointer to the initialization routine is NULL.

The only user-callable routine is *exattach***( )**, which publishes the **ex** interface and initializes the driver and device.

TARGET-SPECIFIC PARAMETERS

base VME address of EXOS I/O ports

This parameter is passed to the driver by *exattach***( )**. The EXOS board presents a small set of I/O ports onto the VME bus. This address is jumper selectable on the EXOS board. This parameter is the same as the address selected on the EXOS board.

VME address modifier code, EXOS access

This parameter is passed to the driver by *exattach***( )**. It specifies the AM (address modifier) code to use when the driver accesses the VME address space (ports) of the EXOS board.

VME address modifier code, target access

This parameter is passed to the driver by *exattach***( )**. It specifies the AM code that the EXOS board needs to use when it accesses the shared memory on the target board.

interrupt vector

This parameter is passed to the driver by *exattach***( )**. It specifies the interrupt vector to be used by the driver to service an interrupt from the EXOS board. The driver connects the interrupt handler to this vector by calling *intConnect***( )**.

interrupt level

This parameter is passed to the driver by *exattach***( )**. It specifies the interrupt level that is associated with the interrupt vector. The driver enables the interrupt from the EXOS by calling *sysIntEnable***( )** with this parameter.

SYSTEM RESOURCE USAGE

When implemented, this driver requires the following system resources:

– one interrupt vector
– 8 bytes in the initialized data section (data)
– 668 bytes in the uninitialized data section (BSS)

The above data and BSS requirements are for the MC68020 architecture and may vary for other architectures. Code size (text) varies greatly between architectures and is therefore not quoted here.

This driver uses *cacheDmaMalloc***( )** to allocate the memory to be shared with the EXOS board. The size requested is 3512 bytes.

This driver can only be operated if this shared memory region is non-cacheable. The driver cannot maintain cache coherency for the shared memory because asynchronous modifications by the EXOS board may share cache lines with locations being operated on by the driver.

SEE ALSO **ifLib**

if_fei

NAME **if_fei** – Intel 82557 Ethernet network interface driver

SYNOPSIS *feiattach***( )** – publish the **fei** network interface

```
STATUS feiattach
    (int unit, char * memBase, int nCFD, int nRFD, int nRFDLoan)
```

DESCRIPTION This module implements the Intel 82557 Ethernet network interface driver.

This driver is designed to be moderately generic, operating unmodified across the entire range of architectures and targets supported by VxWorks. This driver must be given several target-specific parameters, and some external support routines must be provided. These parameters, and the mechanisms used to communicate them to the driver, are detailed below.

This driver supports up to four individual units.

EXTERNAL INTERFACE

The only user-callable routine is *feiattach***( )**, which publishes the **fei** interface and performs some initialization.

After calling *feiattach***( )** to publish the interface, an initialization routine must be called to bring the device up to an operational state. The initialization routine is not a user-callable routine; upper layers call it when the interface flag is set to **UP**, or when the interface's IP address is set.

TARGET-SPECIFIC PARAMETERS

shared memory address

This parameter is passed to the driver via *feiattach***( )**.

The Intel 82557 device is a DMA-type device and typically shares access to some region of memory with the CPU. This driver is designed for systems that directly share memory between the CPU and the 82557.

This parameter can be used to specify an explicit memory region for use by the

82557. This should be done on targets that restrict the 82557 to a particular memory region. The constant **NONE** can be used to indicate that there are no memory limitations, in which case the driver attempts to allocate the shared memory from the system space.

number of Command, Receive, and Loanable-Receive Frame Descriptors
These parameters are passed to the driver via *feiattach***( )**.

The Intel 82557 accesses frame descriptors (and their associated buffers) in memory for each frame transmitted or received. The number of frame descriptors can be configured at run-time using these parameters.

Ethernet address
This parameter is obtained by a call to an external support routine.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires four external support functions:

```
STATUS sys557Init (int unit, BOARD_INFO *pBoard)
```

This routine performs any target-specific initialization required before the 82557 device is initialized by the driver. The driver calls this routine every time it wants to [re]initialize the device. This routine returns OK, or ERROR if it fails.

SYSTEM RESOURCE USAGE

The driver uses *cacheDmaMalloc***( )** to allocate memory to share with the 82557. The size of this area is affected by the configuration parameters specified in the *feiattach***( )** call. The size of one RFD (Receive Frame Descriptor) is is the same as one CFD (Command Frame Descriptor): 1536 bytes. For more information about RFDs and CFDs, see the *Intel 82557 User's Manual.*

Either the shared memory region must be non-cacheable, or else the hardware must implement bus snooping. The driver cannot maintain cache coherency for the device because fields within the command structures are asynchronously modified by both the driver and the device, and these fields may share the same cache line.

Additionally, this version of the driver does not handle virtual-to-physical or physical-to-virtual memory mapping.

TUNING HINTS

The only adjustable parameters are the number of Frame Descriptors that will be created at run-time. These parameters are given to the driver when *feiattach***( )** is called. There is one CFD and one RFD associated with each transmitted frame and each received frame, respectively. For memory-limited applications, decreasing the number of CFDs and RFDs may be desirable. Increasing the number of CFDs will provide no performance benefit after a certain point. Increasing the number of RFDs will provide more buffering before packets are dropped. This can be useful if there are tasks running at a higher priority than the net task.

SEE ALSO

ifLib, *Intel 82557 User's Manual*

if_fn

NAME **if_fn** – Fujitsu MB86960 NICE Ethernet network interface driver

SYNOPSIS *fnattach***( )** – publish the **fn** network interface and initialize the driver and device

```
STATUS fnattach
    (int unit)
```

DESCRIPTION This module implements the Fujitsu MB86960 NICE Ethernet network interface driver.

This driver is non-generic and has only been run on the Fujitsu SPARClite Evaluation Board. It currently supports only unit number zero. The driver must be given several target-specific parameters, and some external support routines must be provided. These parameters, and the mechanisms used to communicate them to the driver, are detailed below.

BOARD LAYOUT This device is on-board. No jumpering diagram is necessary.

EXTERNAL INTERFACE

This driver provides the standard external interface with the following exceptions. All initialization is performed within the attach routine; there is no separate initialization routine. Therefore, in the global interface structure, the function pointer to the initialization routine is NULL.

The only user-callable routine is *fnattach***( )**, which publishes the **fn** interface and initializes the driver and device.

TARGET-SPECIFIC PARAMETERS

External support routines provide all parameters:

device I/O address
: This parameter specifies the base address of the device's I/O register set. This address is assumed to live in SPARClite alternate address space.

interrupt vector
: This parameter specifies the interrupt vector to be used by the driver to service an interrupt from the NICE device. The driver will connect the interrupt handler to this vector by calling *intConnect***( )**.

Ethernet address
: This parameter specifies the unique, six-byte address assigned to the VxWorks target on the Ethernet.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires five external support functions:

`char *sysEnetIOAddrGet (int unit)`
This routine returns the base address of the NICE control registers. The driver calls this routine once, using *fnattach***( )**.

`int sysEnetVectGet (int unit)`
This routine returns the interrupt vector number to be used to connect the driver's interrupt handler. The driver calls this routine once, using *fnattach***( )**.

`STATUS sysEnetAddrGet (int unit, char *pCopy)`
This routine provides the six-byte Ethernet address used by *unit*. It must copy the six-byte address to the space provided by *pCopy*. It returns OK, or ERROR if it fails. The driver calls this routine once, using *fnattach***( )**.

`void sysEnetIntEnable (int unit), void sysEnetIntDisable (int unit)`
These routines enable or disable the interrupt from the NICE for the specified *unit*. Typically, this involves interrupt controller hardware, either internal or external to the CPU. The driver calls these routines only during initialization, using *fnattach***( )**.

SYSTEM RESOURCE USAGE

When implemented, this driver requires the following system resources:

– one mutual exclusion semaphore
– one interrupt vector
– 3944 bytes in text section (text)
– 0 bytes in the initialized data section (data)
– 3152 bytes in the uninitialized data section (BSS)

The above data and BSS requirements are for the SPARClite architecture and may vary for other architectures. Code size (text) varies greatly between architectures and is therefore not quoted here.

The NICE device maintains a private buffer for all packets transmitted and received. Therefore, the driver does not require any system memory to share with the device. This also eliminates all data cache coherency issues.

SEE ALSO

ifLib

if_ln

NAME **if_ln** – AMD Am7990 LANCE Ethernet driver

SYNOPSIS *lnattach***( )** – publish the **ln** network interface and initialize the driver and device

```
STATUS lnattach
    (int unit, char *devAdrs, int ivec, int ilevel, char *memAdrs,
    ULONG memSize, int memWidth, int spare, int spare2)
```

DESCRIPTION This module implements the Advanced Micro Devices Am7990 LANCE Ethernet network interface driver.

This driver is designed to be moderately generic, operating unmodified across the range of architectures and targets supported by VxWorks. To achieve this, the driver must be given several target-specific parameters, and some external support routines must be provided. These parameters, and the mechanisms used to communicate them to the driver, are detailed below. If any of the assumptions stated below are not true for your particular hardware, this driver will probably not function correctly with it.

This driver supports only one LANCE unit per CPU. The driver can be configured to support big-endian or little-endian architectures. It contains error recovery code to handle known device errata related to DMA activity.

BOARD LAYOUT This device is on-board. No jumpering diagram is necessary.

EXTERNAL INTERFACE

This driver provides the standard external interface with the following exceptions. All initialization is performed within the attach routine; there is no separate initialization routine. Therefore, in the global interface structure, the function pointer to the initialization routine is NULL.

The only user-callable routine is *lnattach***( )**, which publishes the **ln** interface and initializes the driver and device.

TARGET-SPECIFIC PARAMETERS

bus mode

This parameter is a global variable that can be modified at run-time.

The LANCE control register #3 determines the bus mode of the device, allowing the support of big-endian and little-endian architectures. This parameter, defined as “u_short lnCSR_3B”, is the value that will be placed into LANCE control register #3. The default value supports Motorola-type buses. For information about changing this parameter, see the manual *Advanced Micro Devices Local Area Network Controller Am7990 (LANCE).*

base address of device registers

This parameter is passed to the driver by *lnattach***( )**. It indicates to the driver where to find the RDP register.

The LANCE presents two registers to the external interface, the RDP (register data port) and RAP (register address port) registers. This driver assumes that these two registers occupy two unique addresses in a memory space that is directly accessible by the CPU executing this driver. The driver assumes that the RDP register is mapped at a lower address than the RAP register; the RDP register is therefore considered the "base address."

interrupt vector

This parameter is passed to the driver by *lnattach***( )**.

This driver configures the LANCE device to generate hardware interrupts for various events within the device; thus it contains an interrupt handler routine. The driver calls *intConnect***( )** to connect its interrupt handler to the interrupt vector generated as a result of the LANCE interrupt.

interrupt level

This parameter is passed to the driver by *lnattach***( )**.

Some targets use additional interrupt controller devices to help organize and service the various interrupt sources. This driver has no board-specific knowledge of such devices. During driver initialization, the external routine *sysLanIntEnable***( )** is called to perform any board-specific operations required to allow servicing LANCE interrupts; for a description of this routine, see "External Support Requirements" below.

This parameter is passed to the external routine.

shared memory address

This parameter is passed to the driver by *lnattach***( )**.

The LANCE device is a DMA type of device and typically shares access to some region of memory with the CPU. This driver is designed for systems that directly share memory between the CPU and the LANCE. It assumes that this shared memory is directly available to it without any arbitration or timing concerns.

This parameter can be used to specify an explicit memory region for use by the LANCE. This should be done on hardware that restricts the LANCE to a particular memory region. The constant NONE can be used to indicate that there are no memory limitations, in which case, the driver attempts to allocate the shared memory from the system space.

shared memory size

This parameter is passed to the driver by *lnattach***( )**.

Specify this parameter to limit explicitly the amount of shared memory this driver can use. The value **NONE** indicates no specific size limitation. This parameter is used only if a specific memory region is provided to the driver.

shared memory width

This parameter is passed to the driver by *lnattach*( ).

Some target hardware that restricts the shared memory region to a specific location also restricts the access width to this region by the CPU. On these targets, performing an access of an invalid width will cause a bus error.

This parameter can be used to specify the number of bytes of access width to be used by the driver during access to the shared memory. The constant NONE can be used to indicate no restrictions.

Current internal support for this mechanism is not robust; implementation may not work on all targets requiring these restrictions.

shared memory buffer size

This parameter is passed to the driver by *lnattach*( ).

The driver and LANCE device exchange network data in buffers. This parameter permits the size of these individual buffers to be limited. A value of zero indicates that the default buffer size should be used. The default buffer size is large enough to hold a maximum-size Ethernet packet.

Use of this parameter should be rare. Network performance will be affected, since the target will no longer be able to receive all valid packet sizes.

Ethernet address

This parameter is obtained directly from a global memory location.

During initialization, the driver needs to know the Ethernet address for the LANCE device. The driver assumes that this address is available in a global, six-byte character array, lnEnetAddr[]. This array is typically created and stuffed by the BSP code.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires one external support function:

`void sysLanIntEnable (int level)`

This routine provides a target-specific enable of the interrupt for the LANCE device. Typically, this involves interrupt controller hardware, either internal or external to the CPU.

This routine is called once, from the *lnattach*( ) routine.

SYSTEM RESOURCE USAGE

When implemented, this driver requires the following system resources:

– one mutual exclusion semaphore
– one interrupt vector
– 24 bytes in the initialized data section (data)
– 208 bytes in the uninitialized data section (BSS)

The above data and BSS requirements are for the MC68020 architecture and may vary for other architectures. Code size (text) varies greatly between architectures and is therefore not quoted here.

If the driver is not given a specific region of memory via the *lnattach***( )** routine, then it calls *cacheDmaMalloc***( )** to allocate the memory to be shared with the LANCE. The size requested is 80,542 bytes. If a memory region is provided to the driver, the size of this region is adjustable to suit user needs.

The LANCE can only be operated if the shared memory region is write-coherent with the data cache. The driver cannot maintain cache coherency for the device for data that is written by the driver because fields within the shared structures are asynchronously modified by both the driver and the device, and these fields may share the same cache line.

SEE ALSO **ifLib**, *Advanced Micro Devices Local Area Network Controller Am7990 (LANCE)*

if_loop

NAME **if_loop** – software loopback network interface driver

SYNOPSIS *loattach***( )** – publish the **lo** network interface and initialize the driver and pseudo-device

```
STATUS loattach
    (void)
```

DESCRIPTION This module implements the software loopback network interface driver. The only user-callable routine is *loattach***( )**, which publishes the **lo** interface and initializes the driver and device.

This interface is used for protocol testing and timing. By default, the loopback interface is accessible at Internet address 127.0.0.1.

BOARD LAYOUT This device is “software only.” A jumpering diagram is not applicable.

SEE ALSO **ifLib**

if_mbc

NAME

if_mbc – Motorola 68EN302 network-interface driver

SYNOPSIS

*mbcattach***( )** – publish the **mbc** network interface and initialize the driver
*mbcIntr***( )** – network interface interrupt handler

```
STATUS mbcattach
    (int unit, void * pEmBase, int inum, int txBdNum, int rxBdNum,
    int dmaParms, UINT8 * bufBase)

void mbcIntr
    (int unit)
```

DESCRIPTION

This is a driver for the Ethernet controller on the 68EN302 chip. The device supports a 16-bit interface, data rates up to 10 Mbps, a dual-ported RAM, and transparent DMA. The dual-ported RAM is used for a 64-entry CAM table, and a 128-entry buffer descriptor table. The CAM table is used to set the Ethernet address of the Ethernet device or to program multicast addresses. The buffer descriptor table is partitioned into fixed-size transmit and receive tables. The DMA operation is transparent and transfers data between the internal FIFOs and external buffers pointed to by the receive- and transmit-buffer descriptors during transmits and receives.

The driver currently supports one Ethernet module controller, but it can be extended to support multiple controllers when needed. An Ethernet module is initialized by calling *mbcattach***( )**.

The driver supports buffer loaning for performance and input/output hook routines. It does not support multicast addresses.

The driver requires that the memory used for transmit and receive buffers be allocated in cache-safe RAM area.

A glitch in the EN302 Rev 0.1 device causes the Ethernet transmitter to lock up from time to time. The driver uses a watchdog timer to reset the Ethernet device when the device runs out of transmit buffers and cannot recover within 20 clock ticks.

BOARD LAYOUT

This device is on-chip. No jumpering diagram is necessary.

EXTERNAL INTERFACE

This driver presents the standard WRS network driver API: first the device unit must be attached with the *mbcattach***( )** routine, then it must be initialized with the *mbcInit***( )** routine.

The only user-callable routine is *mbcattach***( )**, which publishes the **mbc** interface and initializes the driver structures.

TARGET-SPECIFIC PARAMETERS

Ethernet module base address

This parameter is passed to the driver via *mbcattach***( )**.

This parameter is the base address of the Ethernet module. The driver addresses all other Ethernet device registers as offsets from this address.

interrupt vector number

This parameter is passed to the driver via *mbcattach***( )**.

The driver configures the Ethernet device to use this parameter while generating interrupt ack cycles. The interrupt service routine *mbcIntr***( )** is expected to be attached to the corresponding interrupt vector externally, typically in *sysHwInit2***( )**.

number of transmit and receive buffer descriptors

These parameters are passed to the driver via *mbcattach***( )**.

The number of transmit and receive buffer descriptors (BDs) used is configurable by the user while attaching the driver. Each BD is 8 bytes in size and resides in the chip's dual-ported memory, while its associated buffer, 1520 bytes in size, resides in cache-safe conventional RAM. A minimum of 2 receive and 2 transmit BDs should be allocated. If this parameter is NULL, a default of 32 BDs will be used. The maximum number of BDs depends on how the dual-ported BD RAM is partitioned. The 128 BDs in the dual-ported BD RAM can partitioned into transmit and receive BD regions with 8, 16, 32, or 64 transmit BDs and corresponding 120, 112, 96, or 64 receive BDs.

Ethernet DMA parameters

This parameter is passed to the driver via *mbcattach***( )**.

This parameter is used to specify the settings of burst limit, water-mark, and transmit early, which control the Ethernet DMA, and is used to set the EDMA register.

base address of the buffer pool

This parameter is passed to the driver via *mbcattach***( )**.

This parameter is used to notify the driver that space for the transmit and receive buffers need not be allocated, but should be taken from a cache-coherent private memory space provided by the user at the given address. The user should be aware that memory used for buffers must be 4-byte aligned and non-cacheable. All the buffers must fit in the given memory space; no checking will be performed. This includes all transmit and receive buffers (see above) and an additional 16 receive loaner buffers, unless the number of receive BDs is less than 16, in which case that number of loaner buffers will be used. Each buffer is 1520 bytes. If this parameter is **NONE**, space for buffers will be obtained by calling *cacheDmaMalloc***( )** in *cpmattach***( )**.

EXTERNAL SUPPORT REQUIREMENTS

The driver requires the following support functions:

`STATUS sysEnetAddrGet (int unit, UINT8 * addr)`
The driver expects this routine to provide the six-byte Ethernet hardware address that will be used by *unit*. This routine must copy the six-byte address to the space provided by *addr*. This routine is expected to return OK on success, or ERROR. The driver calls this routine, during device initialization, from the *cpmInit***( )** routine.

SYSTEM RESOURCE USAGE

The driver requires the following system resource:

– one mutual exclusion semaphore
– one interrupt vector
– one watchdog timer
– 0 bytes in the initialized data section (data)
– 296 bytes in the uninitialized data section (bss)

The data and BSS sections are quoted for the CPU32 architecture.

If the driver allocates the memory shared with the Ethernet device unit, it does so by calling the *cacheDmaMalloc***( )** routine. For the default case of 32 transmit buffers, 32 receive buffers, and 16 loaner buffers, the total size requested is 121,600 bytes. If a non-cacheable memory region is provided by the user, the size of this region should be this amount, unless the user has specified a different number of transmit or receive BDs.

This driver can only operate if the shared memory region is non-cacheable, or if the hardware implements bus snooping. The driver cannot maintain cache coherency for the device because the buffers are asynchronously modified by both the driver and the device, and these fields may share the same cache line. Additionally, the chip's dual-ported RAM must be declared as non-cacheable memory where applicable.

SEE ALSO

ifLib, *Motorola MC68EN302 User's Manual, Motorola MC68EN302 Device Errata, May 30, 1996*

if_nic

NAME

if_nic – National Semiconductor SNIC Chip (for HKV30) network interface driver

SYNOPSIS

*nicattach***( )** – publish the **nic** network interface and initialize the driver and device

```
STATUS nicattach
    (int unit, NIC_DEVICE *pNic, int ivec)
```

DESCRIPTION This module implements the National Semiconductor 83901 SNIC Ethernet network interface driver.

This driver is non-generic and is for use on the Heurikon HKV30 board. Only unit number zero is supported. The driver must be given several target-specific parameters. These parameters, and the mechanisms used to communicate them to the driver, are detailed below.

BOARD LAYOUT This device is on-board. No jumpering diagram is necessary.

EXTERNAL INTERFACE

This driver provides the standard external interface with the following exceptions. All initialization is performed within the attach routine; there is no separate initialization routine. Therefore, in the global interface structure, the function pointer to the initialization routine is NULL.

The only user-callable routine is *nicattach***( )**, which publishes the **nic** interface and initializes the driver and device.

TARGET-SPECIFIC PARAMETERS

device I/O address

This parameter is passed to the driver by *nicattach***( )**. It specifies the base address of the device's I/O register set.

interrupt vector

This parameter is passed to the driver by *nicattach***( )**. It specifies the interrupt vector to be used by the driver to service an interrupt from the SNIC device. The driver will connect the interrupt handler to this vector by calling *intConnect***( )**.

SYSTEM RESOURCE USAGE

When implemented, this driver requires the following system resources:

– one mutual exclusion semaphore
– one interrupt vector
– 0 bytes in the initialized data section (data)
– 1702 bytes in the uninitialized data section (BSS)

The above data and BSS requirements are for the MC68020 architecture and may vary for other architectures. Code size (text) varies greatly between architectures and is therefore not quoted here.

The HKV30 provides a private buffer for all packets transmitted and received. Thus, the driver does not require any system memory to share with the device. This also eliminates all data cache coherency issues.

SEE ALSO **ifLib**

if_qu

NAME **if_qu** – Motorola MC68EN360 QUICC network interface driver

SYNOPSIS *quattach***( )** – publish the **qu** network interface and initialize driver structures

```
STATUS quattach
    (int unit, UINT32 quAddr, int ivec, int sccNum, int txBdNum,
    int rxBdNum, UINT32 txBdBase, UINT32 rxBdBase, UINT32 bufBase)
```

DESCRIPTION This module implements the Motorola MC68EN360 QUICC Ethernet network interface driver.

This driver is designed to support the Ethernet mode of a SCC residing on the MC68EN360 processor chip. It is generic in the sense that it does not care which SCC is being used, and it supports up to four individual units per board. To achieve this goal, the driver must be given several target-specific parameters, and some external support routines must be provided. These parameters, and the mechanisms used to communicate them to the driver, are detailed below.

This network interface driver does not include support for trailer protocols or data chaining. However, buffer loaning has been implemented in an effort to boost performance. Support for four individual device units was designed into this driver.

This driver maintains cache coherency by allocating buffer space using the *cacheDmaMalloc***( )** routine. This is provided for boards that use the MC68EN360 in ’040 companion mode where it is attached to processors with data cache space.

Due to a lack of suitable hardware, the multiple unit support and ’040 companion mode support have not been tested.

BOARD LAYOUT This device is on-chip. No jumpering diagram is necessary.

EXTERNAL INTERFACE

This driver provides the standard external interface; first the device unit must be attached by the *quattach***( )** routine, then it must be initialized by the *quInit***( )** routine.

The only user-callable routine is *quattach***( )**, which publishes the **qu** interface and initializes the driver structures.

TARGET-SPECIFIC PARAMETERS

base address of MC68EN360 internal memory

This parameter is passed to the driver by *quattach***( )**.

This parameter indicates the address at which the MC68EN360 presents its internal memory (also known as the dual-ported RAM base address). With this address, and the SCC number (see below), the driver is able to compute the

location of the SCC parameter RAM and the SCC register map. This parameter should point to the internal memory of the QUICC chip where the SCC resides physically, which may not necessarily be the master QUICC on the target board.

interrupt vector

This parameter is passed to the driver by *quattach***( )**.

This driver configures the MC68EN360 device to generate hardware interrupts for various events within the device. Therefore, this driver contains an interrupt handler routine. This driver will call the VxWorks system function *intConnect***( )** to connect its interrupt handler to the interrupt vector generated as a result of the MC68EN360 interrupt.

SCC number used

This parameter is passed to the driver by *quattach***( )**.

This driver is written to support four individual device units. At the time that this driver was written, Motorola had released information stating that any of the SCCs on the MC68EN360 may be used in Ethernet mode. Thus, the multiple units supported by this driver may reside on different MC68EN360 chips, or may be on different SCCs within a single MC68EN360. This parameter is used to explicitly state which SCC is being used (SCC1 is most commonly used, thus this parameter most often equals “1”).

number of transmit and receive buffer descriptors

These parameters are passed to the driver by *quattach***( )**.

The number of transmit and receive buffer descriptors (BDs) used is configurable by the user upon attaching the driver. Each buffer descriptor resides in 8 bytes of the MC68EN360’s dual ported RAM space, and each one points to a 1520 byte buffer in regular RAM. There must be a minimum of two transmit and two receive BDs, and there is no maximum, although over a certain amount will not speed up the driver and will waste valuable dual ported RAM space. If this parameter is “NULL”a default value of “32” BDs will be used.

offset of transmit and receive buffer descriptors

These parameters are passed to the driver by *quattach***( )**.

These parameters indicate the base location of the transmit and receive buffer descriptors (BDs). They are offsets in bytes from the base address of MC68EN360 internal memory (see above). Each BD takes up 8 bytes of dual ported RAM, and it is the user’s responsibility to ensure that all specified BDs will fit within dual ported RAM. This includes any other BDs the target board may be using, including other SCCs, SMCs, and the SPI device. There is no default for these parameters; they must be provided by the user.

base address of buffer pool

This parameter is passed to the driver by *quattach***( )**.

This parameter is used to notify the driver that space for the transmit and receive

buffers need not be allocated, but should be taken from a cache-coherent private memory space provided by the user at the given address. The user should be aware that memory used for buffers must be 4-byte aligned and non-cacheable. All the buffers must* fit in the given memory space; no checking will be performed. This includes all transmit and receive buffers (see above) and an additional 16 receive loaner buffers, unless the number of receive BDs is less than 16, in which case that number of loaner buffers will be used. Each buffer is 1520 bytes. If this parameter is "NONE", space for buffers will be obtained by calling *cacheDmaMalloc***( )** in *quattach***( )**.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires three external support functions:

`STATUS sys360EnetEnable (int unit, UINT32 regBase)`

This routine is expected to perform any target specific functions required to enable the Ethernet controller. These functions typically include enabling the Transmit Enable signal (TENA) and connecting the transmit and receive clocks to the SCC. The driver calls this routine, once per unit, from the *quInit***( )** routine.

`void sys360EnetDisable (int unit, UINT32 regBase)`

This routine is expected to perform any target specific functions required to disable the Ethernet controller. This usually involves disabling the Transmit Enable (TENA) signal. The driver calls this routine from the *quReset***( )** routine each time a unit is disabled.

`STATUS sys360EnetAddrGet (int unit, u_char * addr)`

The driver expects this routine to provide the six byte Ethernet hardware address that will be used by this unit. This routine must copy the six byte address to the space provided by *addr*. This routine is expected to return OK on success, or ERROR. The driver calls this routine, once per unit, from the *quInit***( )** routine.

SYSTEM RESOURCE USAGE

When implemented, this driver requires the following system resources:

– one mutual exclusion semaphore
– one interrupt vector
– 0 bytes in the initialized data section (data)
– 1272 bytes in the uninitialized data section (BSS)

The data and BSS sections are quoted for the CPU32 architecture and may vary for other architectures. The code size (text) will varies greatly between architectures, and is therefore not quoted here.

If the driver allocates the memory to share with the Ethernet device unit, it does so by calling the *cacheDmaMalloc***( )** routine. For the default case of 32 transmit buffers, 32 receive buffers, and 16 loaner buffers, the total size requested is 121,600 bytes. If a non-cacheable memory region is provided by the user, the size of this region should be this amount, unless the user has specified a different number of transmit or receive BDs.

This driver can only operate if this memory region is non-cacheable, or if the hardware implements bus snooping. The driver cannot maintain cache coherency for the device because the buffers are asynchronously modified by both the driver and the device, and these fields may share the same cache line. Additionally, the MC68EN360 dual ported RAM must be declared as non-cacheable memory where applicable (e.g., when attached to a 68040 processor).

SEE ALSO **ifLib**, *Motorola MC68360 User's Manual*

if_sl

NAME **if_sl** – Serial Line IP (SLIP) network interface driver

SYNOPSIS

*slipInit***( )** – initialize a SLIP interface
*slipBaudSet***( )** – set the baud rate for a SLIP interface
*slattach***( )** – publish the **sl** network interface and initialize the driver and device
*slipDelete***( )** – delete a SLIP interface

```
STATUS slipInit
    (int unit, char *devName, char *myAddr, char *peerAddr, int baud,
    BOOL compressEnable, BOOL compressAllow, int mtu)

STATUS slipBaudSet
    (int unit, int baud)

STATUS slattach
    (int unit, int fd, BOOL compressEnable, BOOL compressAllow, int mtu)

STATUS slipDelete
    (int unit)
```

DESCRIPTION This module implements the VxWorks Serial Line IP (SLIP) network interface driver. Support for compressed TCP/IP headers (CSLIP) is included.

The SLIP driver enables VxWorks to talk to other machines over serial connections by encapsulating IP packets into streams of bytes suitable for serial transmission.

USER-CALLABLE ROUTINES

SLIP devices are initialized using *slipInit***( )**. Its parameters specify the Internet address for both sides of the SLIP point-to-point link, the name of the tty device on the local host, and options to enable CSLIP header compression. The *slipInit***( )** routine calls *slattach***( )** to attach the SLIP interface to the network. The *slipDelete***( )** routine deletes a specified SLIP interface.

LINK-LEVEL PROTOCOL

SLIP is a simple protocol that uses four token characters to delimit each packet:

- **END** (0300)
- **ESC** (0333)
- **TRANS_END** (0334)
- **TRANS_ESC** (0335)

The **END** character denotes the end of an IP packet. The **ESC** character is used with **TRANS_END** and **TRANS_ESC** to circumvent potential occurrences of **END** or **ESC** within a packet. If the **END** character is to be embedded, SLIP sends "**ESC TRANS_END**" to avoid confusion between a SLIP-specific **END** and actual data whose value is **END**. If the **ESC** character is to be embedded, then SLIP sends "**ESC TRANS_ESC**'' to avoid confusion. (Note that the SLIP **ESC** is not the same as the ASCII ESC.)

On the receiving side of the connection, SLIP uses the opposite actions to decode the SLIP packets. Whenever an **END** character is received, SLIP assumes a full IP packet has been received and sends it up to the IP layer.

IMPLEMENTATION

The write side of a SLIP connection is an independent task. Each SLIP interface has its own output task that sends SLIP packets over a particular tty device channel. Whenever a packet is ready to be sent out, the SLIP driver activates this task by giving a semaphore. When the semaphore is available, the output task performs packetization (as explained above) and writes the packet to the tty device.

The receiving side is implemented as a "hook" into the tty driver. A tty *ioctl***( )** request, **FIOPROTOHOOK**, informs the tty driver to call the SLIP interrupt routine every time a character is received from a serial port. By tracking the number of characters and watching for the **END** character, the number of calls to *read***( )** and context switching time have been reduced. The SLIP interrupt routine will queue a call to the SLIP read routine only when it knows that a packet is ready in the tty driver's ring buffer. The SLIP read routine will read a whole SLIP packet at a time and process it according to the SLIP framing rules. When a full IP packet is decoded out of a SLIP packet, it is queued to IP's input queue.

CSLIP compression is implemented to decrease the size of the TCP/IP header information, thereby improving the data to header size ratio. CSLIP manipulates header information just before a packet is sent and just after a packet is received. Only TCP/IP headers are compressed and uncompressed; other protocol types are sent and received normally. A functioning CSLIP driver is required on the peer (destination) end of the physical link in order to carry out a CSLIP "conversation." Multiple units are supported by this driver. Each individual unit may have CSLIP support disabled or enabled, independent of the state of other units.

BOARD LAYOUT

No hardware is directly associated with this driver; therefore, a jumpering diagram is not applicable.

SEE ALSO **ifLib**, **tyLib**, John Romkey: RFC-1055, *A Nonstandard for Transmission of IP Datagrams Over Serial Lines: SLIP*, Van Jacobson: RFC-1144, entitled *Compressing TCP/IP Headers for Low-Speed Serial Links*

ACKNOWLEDGEMENT

This program is based on original work done by Rick Adams of The Center for Seismic Studies and Chris Torek of The University of Maryland. The CSLIP enhancements are based on work done by Van Jacobson of University of California, Berkeley for the "cslip-2.7" release.

if_sm

NAME **if_sm** – shared memory backplane network interface driver

SYNOPSIS *smIfAttach***( )** – publish the **sm** interface and initialize the driver and device

```
STATUS smIfAttach
    (int unit, SM_ANCHOR * pAnchor, int maxInputPkts, int intType,
    int intArg1, int intArg2, int intArg3, int ticksPerBeat, int numLoan)
```

DESCRIPTION This module implements the VxWorks shared memory backplane network interface driver.

This driver is designed to be moderately generic, operating unmodified across the range of hosts and targets supported by VxWorks. To achieve this, the driver must be given several target-specific parameters, and some external support routines must be provided. These parameters are detailed below.

The only user-callable routine is *smIfAttach***( )**, which publishes the **sm** interface and initializes the driver and device.

This driver is layered between the shared memory packet library and the network modules. The backplane driver gives CPUs residing on a common backplane the ability to communicate using IP (via shared memory).

This driver is used both under VxWorks and other host operating systems, e.g., SunOs.

BOARD LAYOUT This device is "software only." There is no jumpering diagram required.

TARGET-SPECIFIC PARAMETERS

local address of anchor

This parameter is passed to the driver by *smIfAttach***( )**. It is the local address by which the local CPU accesses the shared memory anchor.

maximum number of input packets
: This parameter is passed to the driver by *smIfAttach***( )**. It specifies the maximum number of incoming shared memory packets that can be queued to this CPU at one time.

method of notification
: These parameters are passed to the driver by *smIfAttach***( )**. Four parameters can be used to allow a CPU to announce the method by which it is to be notified of input packets that have been queued to it.

heartbeat frequency
: This parameter is passed to the driver by *smIfAttach***( )**. It specifies the frequency of the shared memory anchor's heartbeat, which is expressed in terms of the number of CPU ticks on the local CPU corresponding to one heartbeat period.

number of buffers to loan
: This parameter is passed to the driver by *smIfAttach***( )**. When the value is non-zero, this parameter specifies the number of shared memory packets available to be loaned out.

SEE ALSO **ifLib**, **smNetLib**

if_sn

NAME **if_sn** – National Semiconductor DP83932B SONIC Ethernet network interface driver

SYNOPSIS *snattach***( )** – publish the **sn** network interface and initialize the driver and device

```
STATUS snattach
    (int unit, char * pDevRegs, int ivec)
```

DESCRIPTION This module implements the National Semiconductor DP83932 SONIC Ethernet network interface driver.

This driver is designed to be moderately generic, operating unmodified across the range of architectures and targets supported by VxWorks. To achieve this, the driver must be given several target-specific parameters, and some external support routines must be provided. These parameters, and the mechanisms used to communicate them to the driver, are detailed below. If any of the assumptions stated below are not true for your particular hardware, this driver will probably not function correctly with it. This driver supports up to four individual units per CPU.

BOARD LAYOUT This device is on-board. No jumpering diagram is necessary.

EXTERNAL INTERFACE

This driver provides the standard external interface with the following exceptions. All initialization is performed within the attach routine; there is no separate initialization routine. Therefore, in the global interface structure, the function pointer to the initialization routine is NULL.

There is one user-callable routine, *snattach*( ); for details, see the manual entry for this routine.

TARGET-SPECIFIC PARAMETERS

device I/O address

This parameter is passed to the driver by *snattach*( ). It specifies the base address of the device's I/O register set.

interrupt vector

This parameter is passed to the driver by *snattach*( ). It specifies the interrupt vector to be used by the driver to service an interrupt from the SONIC device. The driver will connect the interrupt handler to this vector by calling *intConnect*( ).

Ethernet address

This parameter is obtained by calling an external support routine. It specifies the unique, six-byte address assigned to the VxWorks target on the Ethernet.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires five external support functions:

`void sysEnetInit (int unit)`

This routine performs any target-specific operations that must be executed before the SONIC device is initialized. The driver calls this routine, once per unit, from *snattach*( ).

`STATUS sysEnetAddrGet (int unit, char *pCopy)`

This routine provides the six-byte Ethernet address used by *unit*. It must copy the six-byte address to the space provided by *pCopy*. This routine returns OK, or ERROR if it fails. The driver calls this routine, once per unit, from *snattach*( ).

`void sysEnetIntEnable (int unit), void sysEnetIntDisable (int unit)`

These routines enable or disable the interrupt from the SONIC device for the specified *unit*. Typically, this involves interrupt controller hardware, either internal or external to the CPU. The driver calls these routines only during initialization, from *snattach*( ).

`void sysEnetIntAck (int unit)`

This routine performs any interrupt acknowledgement or clearing that may be required. This typically involves an operation to some interrupt control hardware. The driver calls this routine from the interrupt handler.

DEVICE CONFIGURATION

Two global variables, **snDcr** and **snDcr2**, are used to set the SONIC device configuration registers. By default, the device is programmed in 32-bit mode with zero wait states. If these values are not suitable, the **snDcr** and **snDcr2** variables should be modified before calling *snattach***( )**. See the SONIC manual to change these parameters.

SYSTEM RESOURCE USAGE

When implemented, this driver requires the following system resources:

– one interrupt vector
– 0 bytes in the initialized data section (data)
– 696 bytes in the uninitialized data section (BSS)

The above data and BSS requirements are for the MC68020 architecture and may vary for other architectures. Code size (text) varies greatly between architectures and is therefore not quoted here.

This driver uses *cacheDmaMalloc***( )** to allocate the memory to be shared with the SONIC device. The size requested is 117,188 bytes.

The SONIC device can only be operated if the shared memory region is write-coherent with the data cache. The driver cannot maintain cache coherency for the device for data that is written by the driver because fields within the shared structures are asynchronously modified by the driver and the device, and these fields may share the same cache line.

SEE ALSO

ifLib

if_ulip

NAME

if_ulip – network interface driver for User Level IP (VxSim)

SYNOPSIS

*ulipInit***( )** – initialize the ULIP interface (VxSim)
*ulattach***( )** – attach a ULIP interface to a list of network interfaces (VxSim)
*ulipDelete***( )** – delete a ULIP interface (VxSim)

```
STATUS ulipInit
    (int unit, char *myAddr, char *peerAddr, int procnum)

STATUS ulattach
    (int unit)

STATUS ulipDelete
    (int unit)
```

DESCRIPTION

This module implements the VxWorks User Level IP (ULIP) network driver. The ULIP driver allows VxWorks under UNIX to talk to other machines by handing off IP packets to the UNIX host for processing.

The ULIP driver is automatically included and initialized by the VxSim BSPs; normally there is no need for applications to use these routines directly.

USER-CALLABLE ROUTINES

When initializing the device, it is necessary to specify the Internet address for both sides of the ULIP point-to-point link (local side and the remote side of the connection) using *ulipInit***( )**.

```
STATUS ulipInit
    (
    int unit,       /* ULIP unit number (0 – NULIP-1) */
    char *myAddr,   /* IP address of the interface */
    char *peerAddr, /* IP address of the remote peer interface */
    int procnum     /* processor number to map to ULIP interface */
    )
```

For example, the following initializes a ULIP device whose Internet address is 127.0.1.1:

```
ulipInit (0, "127.0.1.1", "147.11.1.132", 1);
```

The standard network interface call is:

```
STATUS ulattach
    (
    int unit  /* unit number */
    )
```

However, it should not be called. The following call will delete the first ULIP interface from the list of network interfaces:

```
ulipDelete (0);     /* unit number */
```

Up to NULIP(2) units may be created.

SEE ALSO

VxSim User's Guide

if_ultra

NAME **if_ultra** – SMC Elite Ultra Ethernet network interface driver

SYNOPSIS *ultraattach***( )** – publish the **ultra** network interface and initialize the driver and device
*ultraShow***( )** – display statistics for the **ultra** network interface

```
STATUS ultraattach
    (int unit, int ioAddr, int ivec, int ilevel, int memAddr, int memSize,
    int config)

void ultraShow
    (int unit, BOOL zap)
```

DESCRIPTION This module implements the SMC Elite Ultra Ethernet network interface driver.

This driver supports single transmission and multiple reception. The Current register is a write pointer to the ring. The Bound register is a read pointer from the ring. This driver gets the Current register at the interrupt level and sets the Bound register at the task level. The interrupt is never masked at the task level.

CONFIGURATION The W1 jumper should be set in the position of “Software Configuration”. The defined I/O address in **config.h** must match the one stored in EEROM. The RAM address, the RAM size, and the IRQ level are defined in **config.h**. IRQ levels 2,3,5,7,10,11,15 are supported.

EXTERNAL INTERFACE

The only user-callable routines are *ultraattach***( )** and *ultraShow***( )**:

*ultraattach***( )**
publishes the **ultra** interface and initializes the driver and device.

*ultraShow***( )**
displays statistics that are collected in the interrupt handler.

inetLib

NAME **inetLib** – Internet address manipulation routines

SYNOPSIS *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 Internet address from network and host numbers

inet_makeaddr( ) – form 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 Internet network number from string to address
inet_ntoa_b( ) – convert network address to dot notation and store in buffer
inet_ntoa( ) – convert network address to dot notation

```
u_long inet_addr
    (char *inetString)

int inet_lnaof
    (int inetAddress)

void inet_makeaddr_b
    (int netAddr, int hostAddr, struct in_addr *pInetAddr)

struct in_addr inet_makeaddr
    (int netAddr, int hostAddr)

int inet_netof
    (struct in_addr inetAddress)

void inet_netof_string
    (char *inetString, char *netString)

u_long inet_network
    (char *inetString)

void inet_ntoa_b
    (struct in_addr inetAddress, char *pString)

char *inet_ntoa
    (struct in_addr inetAddress)
```

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 dot 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 dot notation or as a 4-byte number. Values specified using the dot notation take one of the following forms:

a.b.c.d
a.b.c
a.b
a

When 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**" are ordered from left to right.

When 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".

When 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".

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as parts in a dot notation may be decimal, octal, or hexadecimal, as specified in the C language. That is, a leading 0x or 0X implies hexadecimal; otherwise, a leading 0 implies octal. With no leading 0, the number is interpreted as decimal.

INCLUDE FILES **inetLib.h**, **inet.h**

SEE ALSO UNIX BSD 4.3 manual entry for **inet(3N)**, *VxWorks Programmer's Guide: Network*

inflateLib

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

SYNOPSIS *inflate***( )** – inflate compressed code

```
int inflate
    (Byte * src, Byte * dest, int nBytes)
```

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

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

intArchLib

NAME

intArchLib – architecture-dependent interrupt library

SYNOPSIS

intLevelSet( ) – set the interrupt level (MC680x0, SPARC, i960, x86)
intLock( ) – lock out interrupts
intUnlock( ) – cancel interrupt locks
intEnable( ) – enable corresponding interrupt bits (MIPS, PowerPC)
intDisable( ) – disable corresponding interrupt bits (MIPS, PowerPC)
intCRGet( ) – read the contents of the cause register (MIPS)
intCRSet( ) – write the contents of the cause register (MIPS)
intSRGet( ) – read the contents of the status register (MIPS)
intSRSet( ) – update the contents of the status register (MIPS)
intConnect( ) – connect a C routine to a hardware interrupt
intHandlerCreate( ) – construct an interrupt handler for a C routine (MC680x0, SPARC, i960, x86, MIPS)
intLockLevelSet( ) – set the current interrupt lock-out level (MC680x0, SPARC, i960, x86)
intLockLevelGet( ) – get the current interrupt lock-out level (MC680x0, SPARC, i960, x86)
intVecBaseSet( ) – set the vector (trap) base address (MC680x0, SPARC, i960, x86, MIPS)
intVecBaseGet( ) – get the vector (trap) base address (MC680x0, SPARC, i960, x86, MIPS)
intVecSet( ) – set a CPU vector (trap) (MC680x0, SPARC, i960, x86, MIPS)
intVecGet( ) – get an interrupt vector (MC680x0, SPARC, i960, x86, MIPS)
intVecTableWriteProtect( ) – write-protect exception vector table (MC680x0, SPARC, i960, x86)

```
int intLevelSet
    (int level)

int intLock
    (void)

void intUnlock
    (int lockKey)

int intEnable
    (int level)

int intDisable
    (int level)

int intCRGet
    (void)

void intCRSet
    (int value)

int intSRGet
    (void)
```

```
int intSRSet
    (int value)

STATUS intConnect
    (VOIDFUNCPTR * vector, VOIDFUNCPTR routine, int parameter)

FUNCPTR intHandlerCreate
    (FUNCPTR routine, int parameter)

void intLockLevelSet
    (int newLevel)

int intLockLevelGet
    (void)

void intVecBaseSet
    (FUNCPTR * baseAddr)

FUNCPTR *intVecBaseGet
    (void)

void intVecSet
    (FUNCPTR * vector, FUNCPTR function)

FUNCPTR intVecGet
    (FUNCPTR * vector)

STATUS intVecTableWriteProtect
    (void)
```

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 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***( )** (MC680x0, SPARC, i960, and i386/i486 only). The routine *intLevelSet***( )** changes the current interrupt level of the processor (MC680x0, SPARC, and i960).

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 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

SYNOPSIS *intContext***( )** – determine if the current state is in interrupt or task context
*intCount***( )** – get the current interrupt nesting depth

```
BOOL intContext
     (void)

int intCount
     (void)
```

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

SYNOPSIS

creat( ) – create a file
unlink( ) – delete a file (POSIX)
remove( ) – remove a file (ANSI)
open( ) – open a file
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

```
int creat
    (const char *name, int flag)

STATUS unlink
    (char *name)

STATUS remove
    (const char *name)

int open
    (const char *name, int flags, int mode)

STATUS close
    (int fd)

int rename
    (const char *oldname, const char *newname)

int read
    (int fd, char *buffer, size_t maxbytes)

int write
    (int fd, char *buffer, size_t nbytes)
```

```
int ioctl
    (int fd, int function, int arg)

int lseek
    (int fd, long offset, int whence)

STATUS ioDefPathSet
    (char *name)

void ioDefPathGet
    (char *pathname)

STATUS chdir
    (char *pathname)

char *getcwd
    (char *buffer, int size)

char *getwd
    (char *pathname)

void ioGlobalStdSet
    (int stdFd, int newFd)

int ioGlobalStdGet
    (int stdFd)

void ioTaskStdSet
    (int taskId, int stdFd, int newFd)

int ioTaskStdGet
    (int taskId, int stdFd)

BOOL isatty
    (int fd)
```

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*

ioMmuMicroSparcLib

NAME **ioMmuMicroSparcLib** – microSparc I/II I/O DMA library

SYNOPSIS *ioMmuMicroSparcInit***( )** – initialize the microSparc I/II I/O MMU data structures
*ioMmuMicroSparcMap***( )** – map the I/O MMU for microSparc I/II (TMS390S10/MB86904)

```
STATUS ioMmuMicroSparcInit
    (void * physBase, UINT range)

STATUS ioMmuMicroSparcMap
    (UINT dvmaAdrs, void * physBase, UINT size)
```

DESCRIPTION This library contains the SPARC architecture-specific functions *ioMmuMicroSparcInit***( )** and *ioMmuMicroSparcMap***( )**, needed to set up the I/O mapping for S-Bus DMA devices using the TI TMS390S10 and the MicroSparc II Mb86904 architecture.

INCLUDE FILES **arch/sparc/microSparc.h**

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

1

iosLib

NAME

iosLib – I/O system library

SYNOPSIS

*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

```
STATUS iosInit
    (int max_drivers, int max_files, char *nullDevName)

int iosDrvInstall
    (FUNCPTR pCreate, FUNCPTR pDelete, FUNCPTR pOpen, FUNCPTR pClose,
    FUNCPTR pRead, FUNCPTR pWrite, FUNCPTR pIoctl)

STATUS iosDrvRemove
    (int drvnum, BOOL forceClose)

STATUS iosDevAdd
    (DEV_HDR *pDevHdr, char *name, int drvnum)

void iosDevDelete
    (DEV_HDR *pDevHdr)

DEV_HDR *iosDevFind
    (char *name, char **pNameTail)

int iosFdValue
    (int fd)
```

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 *VxWorks Programmer's Guide: I/O System*.

INCLUDE FILES

iosLib.h

SEE ALSO

intLib, **ioLib**, *VxWorks Programmer's Guide: I/O System*

iosShow

NAME

iosShow – I/O system show routines

SYNOPSIS

*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

```
void iosShowInit
    (void)

void iosDrvShow
    (void)

void iosDevShow
    (void)

void iosFdShow
    (void)
```

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*

kernelLib

NAME

kernelLib – VxWorks kernel library

SYNOPSIS

*kernelInit***( )** – initialize the kernel
*kernelVersion***( )** – return the kernel revision string
*kernelTimeSlice***( )** – enable round-robin selection

```
void kernelInit
    (FUNCPTR rootRtn, unsigned rootMemSize, char * pMemPoolStart,
    char * pMemPoolEnd, unsigned intStackSize, int lockOutLevel)

char *kernelVersion
    (void)
```

```
STATUS kernelTimeSlice
    (int ticks)
```

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 routine *tickAnnounce***( )** will increment the task’s time-slice count. When the specified time-slice

interval is completed, the counter is cleared 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.

If a higher priority task preempts a task during its time-slice, the time-slice of the preempted task count is not changed for the duration of the preemption. If preemption is disabled during round-robin scheduling, the time-slice count of the executing task is not incremented.

INCLUDE FILES **kernelLib.h**

SEE ALSO **taskLib**, **intLib**, *VxWorks Programmer's Guide: Basic OS*

ledLib

NAME **ledLib** – line-editing library

SYNOPSIS *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

```
int ledOpen
    (int inFd, int outFd, int histSize)

STATUS ledClose
    (int led_id)

int ledRead
    (int led_id, char *string, int maxBytes)

void ledControl
    (int led_id, int inFd, int outFd, int histSize)
```

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.
^H	– Same as **h**.
*n***l**	– (letter el) Move right *n* characters.
SPACE	– 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 SPACE	– 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 SPACE – Delete character.
dl – Delete character.
dw – Delete word.
dd – Delete entire line.
d$ – Delete everything from cursor to end of line.
D – Same as **d$**.
p – Put last deletion after the cursor.
P – Put last deletion before the cursor.
u – Undo last command.
~ – Toggle case, lower to upper or vice versa.

Special commands:

CTRL+U – Delete line and leave edit mode.
CTRL+L – Redraw line.
CTRL+D – Complete symbol name.
RETURN – Give line to 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*

loadLib

NAME **loadLib** – object module loader

SYNOPSIS

*loadModule***( )** – load an object module into memory
*loadModuleAt***( )** – load an object module into memory

```
MODULE_ID loadModule
    (int fd, int symFlag)

MODULE_ID loadModuleAt
    (int fd, int symFlag, char **ppText, char **ppData, char **ppBss)
```

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

SYNOPSIS

*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

```
void loginInit
    (void)
```

```
STATUS loginUserAdd
    (char name[MAX_LOGIN_NAME_LEN+1], char passwd[80])

STATUS loginUserDelete
    (char *name, char *passwd)

STATUS loginUserVerify
    (char *name, char *passwd)

void loginUserShow
    (void)

STATUS loginPrompt
    (char *userName)

void loginStringSet
    (char *newString)

void loginEncryptInstall
    (FUNCPTR rtn, int var)

STATUS loginDefaultEncrypt
    (char *in, char *out)
```

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 **INCLUDE_SECURITY** is defined in **configAll.h**. Defining **INCLUDE_SECURITY** also adds a single default user to the login table. The default user and password are defined in **configAll.h** 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

SYNOPSIS

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

```
STATUS logInit
    (int fd, int maxMsgs)

int logMsg
    (char *fmt, int arg1, int arg2, int arg3, int arg4, int arg5, int arg6)
```

```
void logFdSet
    (int fd)

STATUS logFdAdd
    (int fd)

STATUS logFdDelete
    (int fd)

void logTask
    (void)
```

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 interrupt service routine as well as from tasks. Normal I/O, such as *printf*( ) output to a serial port, cannot be done from an interrupt service routine.

DEFERRED LOGGING

Print formatting is performed within the context of *logTask*( ), rather than the context of the task calling *logMsg*( ). Since formatting can require considerable stack space, this can reduce stack sizes for tasks that only need to do I/O for error output.

However, this also means that the arguments to *logMsg*( ) are not interpreted at the time of the call to *logMsg*( ), but rather are interpreted at some later time by *logTask*( ). This means that the arguments to *logMsg*( ) should not be pointers to volatile entities. For example, pointers to dynamic or changing strings and buffers should not be passed as arguments to be formatted. Thus the following would not give the desired results:

```
doLog (which)
    {
    char string [100];
    strcpy (string, which ? "hello" : "goodbye");
    ...
    logMsg (string);
    }
```

By the time *logTask***( )** formats the message, the stack frame of the caller may no longer exist and the pointer *string* may no longer be valid. On the other hand, the following is correct since the string pointer passed to the *logTask***( )** always points to a static string:

```
doLog (which)
    {
    char *string;
    string = which ? "hello" : "goodbye";
    ...
    logMsg (string);
    }
```

INITIALIZATION To initialize the message logging facilities, *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*

lptDrv

NAME **lptDrv** – parallel chip device driver for the IBM-PC LPT

SYNOPSIS *lptDrv***( )** – initialize the LPT driver
*lptDevCreate***( )** – create a device for an LPT port
*lptShow***( )** – show LPT statistics

```
STATUS lptDrv
    (int channels, LPT_RESOURCE *pResource)

STATUS lptDevCreate
    (char *name, int channel)

void lptShow
    (UINT channel)
```

DESCRIPTION This is the driver for the LPT used on the IBM-PC. If **INCLUDE_LPT** is defined, the driver initializes the LPT on the PC.

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: *lptDrv***( )** to initialize the driver, and *lptDevCreate***( )** to create devices.

There are one other callable routines: *lptShow***( )** to show statistics. The argument to *lptShow***( )** is a channel number, 0 to 2.

Before the driver can be used, it must be initialized by calling *lptDrv***( )**. This routine should be called exactly once, before any reads, writes, or calls to *lptDevCreate***( )**. Normally, it is called from *usrRoot***( )** in **usrConfig.c**. The first argument to *lptDrv***( )** is a number of channels, 0 to 2. The second argument is a pointer to the resource table. Definitions of members of the resource table structure are:

```
int  ioBase;        /* IO base address */
int  intVector;     /* interrupt vector */
int  intLevel;      /* interrupt level */
BOOL autofeed;      /* TRUE if enable autofeed */
int  busyWait;      /* loop count for BUSY wait */
int  strobeWait;    /* loop count for STROBE wait */
int  retryCnt;      /* retry count */
int  timeout;       /* timeout second for syncSem */
```

IOCTL FUNCTIONS This driver responds to two functions: **LPT_SETCONTROL** and **LPT_GETSTATUS**. The argument for **LPT_SETCONTROL** is a value of the control register. The argument for **LPT_GETSTATUS** is a integer pointer where a value of the status register is stored.

SEE ALSO *VxWorks Programmer's Guide: I/O System*

lstLib

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

SYNOPSIS
*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

```
void lstInit
     (LIST *pList)

void lstAdd
     (LIST *pList, NODE *pNode)

void lstConcat
     (LIST *pDstList, LIST *pAddList)

int lstCount
     (LIST *pList)

void lstDelete
     (LIST *pList, NODE *pNode)

void lstExtract
     (LIST *pSrcList, NODE *pStartNode, NODE *pEndNode, LIST *pDstList)

NODE *lstFirst
     (LIST *pList)

NODE *lstGet
     (LIST *pList)

void lstInsert
     (LIST *pList, NODE *pPrev, NODE *pNode)

NODE *lstLast
     (LIST *pList)

NODE *lstNext
     (NODE *pNode)

NODE *lstNth
     (LIST *pList, int nodenum)

NODE *lstPrevious
     (NODE *pNode)

NODE *lstNStep
     (NODE *pNode, int nStep)

int lstFind
     (LIST *pList, NODE *pNode)

void lstFree
     (LIST *pList)
```

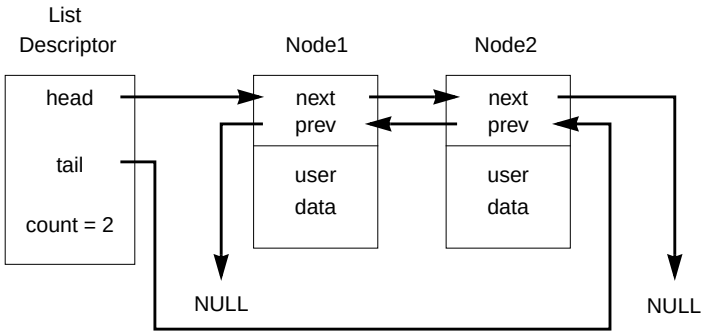
DESCRIPTION

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

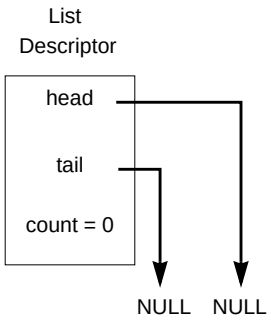
nodes in the list, and a count of the number of nodes in the list. The nodes in the list can be any user-defined structure, but they must reserve space for two pointers as their first elements. Both the forward and backward chains are terminated with a NULL pointer.

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

SYNOPSIS

*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

```
STATUS m2IcmpInit
    (void)

STATUS m2IcmpGroupInfoGet
    (M2_ICMP * pIcmpInfo)

STATUS m2IcmpDelete
    (void)
```

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**.

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

SYNOPSIS

*m2IfInit***( )** – initialize MIB-II interface-group routines
*m2IfGroupInfoGet***( )** – get the MIB-II interface-group scalar variables
*m2IfTblEntryGet***( )** – get a MIB-II interface-group table entry
*m2IfTblEntrySet***( )** – set the state of a MIB-II interface entry to UP or DOWN
*m2IfDelete***( )** – delete all resources used to access the interface group

```
STATUS m2IfInit
    (FUNCPTR pTrapRtn, void * pTrapArg)

STATUS m2IfGroupInfoGet
    (M2_INTERFACE * pIfInfo)

STATUS m2IfTblEntryGet
    (int search, M2_INTERFACETBL * pIfReqEntry)

STATUS m2IfTblEntrySet
    (M2_INTERFACETBL * pIfTblEntry)

STATUS m2IfDelete
    (void)
```

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**.

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**

m2IpLib

NAME **m2IpLib** – MIB-II IP-group API for SNMP agents

SYNOPSIS

*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

```
STATUS m2IpInit
    (int maxRouteTableSize)
```

```
STATUS m2IpGroupInfoGet
    (M2_IP * pIpInfo)

STATUS m2IpGroupInfoSet
    (unsigned int varToSet, M2_IP * pIpInfo)

STATUS m2IpAddrTblEntryGet
    (int search, M2_IPADDRTBL * pIpAddrTblEntry)

STATUS m2IpAtransTblEntryGet
    (int search, M2_IPATRANSTBL * pReqIpAtEntry)

STATUS m2IpAtransTblEntrySet
    (M2_IPATRANSTBL * pReqIpAtEntry)

STATUS m2IpRouteTblEntryGet
    (int search, M2_IPROUTETBL * pIpRouteTblEntry)

STATUS m2IpRouteTblEntrySet
    (int varToSet, M2_IPROUTETBL * pIpRouteTblEntry)

STATUS m2IpDelete
    (void)
```

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**.

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

SYNOPSIS *m2Init***( )** – initialize the SNMP MIB-2 library
*m2Delete***( )** – delete all the MIB-II library groups

```
STATUS m2Init
    (char * pMib2SysDescr, char * pMib2SysContact, char * pMib2SysLocation,
    M2_OBJECTID * pMib2SysObjectId, FUNCPTR pTrapRtn, void * pTrapArg,
    int maxRouteTableSize)

STATUS m2Delete
    (void)
```

DESCRIPTION This library provides Management Information Base (MIB-II, defined in RFC 1213) services for applications wishing to have access to MIB parameters.

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)",
                "1010 Atlantic Avenue 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**

m2SysLib

NAME

m2SysLib – MIB-II system-group API for SNMP agents

SYNOPSIS

*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

```
STATUS m2SysInit
    (char * pMib2SysDescr, char * pMib2SysContact, char * pMib2SysLocation,
    M2_OBJECTID * pObjectId)

STATUS m2SysGroupInfoGet
    (M2_SYSTEM * pSysInfo)

STATUS m2SysGroupInfoSet
    (unsigned int varToSet, M2_SYSTEM * pSysInfo)

STATUS m2SysDelete
    (void)
```

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**.

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**

m2TcpLib

NAME **m2TcpLib** – MIB-II TCP-group API for SNMP agents

SYNOPSIS

*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

```
STATUS m2TcpInit
    (void)

STATUS m2TcpGroupInfoGet
    (M2_TCPINFO * pTcpInfo)

STATUS m2TcpConnEntryGet
    (int search, M2_TCPCONNTBL * pReqTcpConnEntry)

STATUS m2TcpConnEntrySet
    (M2_TCPCONNTBL * pReqTcpConnEntry)

STATUS m2TcpDelete
    (void)
```

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**.

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; /* Local 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**

m2UdpLib

NAME **m2UdpLib** – MIB-II UDP-group API for SNMP agents

SYNOPSIS *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

```
STATUS m2UdpInit
    (void)

STATUS m2UdpGroupInfoGet
    (M2_UDP * pUdpInfo)

STATUS m2UdpTblEntryGet
    (int search, M2_UDPTBL * pUdpEntry)

STATUS m2UdpDelete
    (void)
```

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**.

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**

m68302Sio

NAME **m68302Sio** – Motorola MC68302 bimodal tty driver

SYNOPSIS *m68302SioInit***( )** – initialize a **M68302_CP**
*m68302SioInit2***( )** – initialize a **M68302_CP** (part 2)

```
void m68302SioInit
    (M68302_CP * pCp)

void m68302SioInit2
    (M68302_CP * pCp)
```

DESCRIPTION This is the driver for the internal communications processor (CP) of the Motorola MC68302.

USER-CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. Before the driver can be used, it must be initialized by calling the routines *m68302SioInit***( )** and *m68302SioInit2***( )**. Normally, they are called by *sysSerialHwInit***( )** and *sysSerialHwInit2***( )** in **sysSerial.c**

This driver uses 408 bytes of buffer space as follows:

128 bytes for portA tx buffer
128 bytes for portB tx buffer
128 bytes for portC tx buffer
8 bytes for portA rx buffers (8 buffers, 1 byte each)
8 bytes for portB rx buffers (8 buffers, 1 byte each)
8 bytes for portC rx buffers (8 buffers, 1 byte each)

The buffer pointer in the **m68302cp** structure points to the buffer area, which is usually specified as **IMP_BASE_ADDR**.

IOCTL FUNCTIONS This driver responds to the same *ioctl***( )** codes as a normal tty driver; for more information, see the manual entry for **tyLib**. The available baud rates are 300, 600, 1200, 2400, 4800, 9600 and 19200.

SEE ALSO **ttyDrv**, **tyLib**

INCLUDE FILES **drv/sio/m68302Sio.h sioLib.h**

m68332Sio

NAME **m68332Sio** – Motorola MC68332 tty driver

SYNOPSIS *m68332DevInit***( )** – initialize the SCC
*m68332Int***( )** – handle an SCC interrupt

```
void m68332DevInit
     (M68332_CHAN *pChan)

void m68332Int
     (M68332_CHAN *pChan)
```

DESCRIPTION This is the driver for the Motorola MC68332 on-chip UART. It has only one serial channel.

USAGE A **M68332_CHAN** structure is used to describe the chip. The BSP's *sysHwInit***( )** routine typically calls *sysSerialHwInit***( )**, which initializes all the values in the **M68332_CHAN** structure (except the **SIO_DRV_FUNCS**) before calling *m68332DevInit***( )**. The BSP's *sysHwInit2***( )** routine typically calls *sysSerialHwInit2***( )**, which connects the chips interrupt (m68332Int) via *intConnect***( )**.

INCLUDE FILES **drv/sio/m68332Sio.h**

m68360Sio

NAME **m68360Sio** – Motorola MC68360 SCC UART serial driver

SYNOPSIS *m68360DevInit***( )** – initialize the SCC
*m68360Int***( )** – handle an SCC interrupt

```
void m68360DevInit
     (M68360_CHAN *pChan)

void m68360Int
     (M68360_CHAN *pChan)
```

DESCRIPTION This is the driver for the SCC's in the internal Communications Processor (CP) of the Motorola MC68360. This driver only supports the SCC's in asynchronous UART mode.

USAGE A m68360_CHAN structure is used to describe the chip. The BSP's *sysHwInit***( )** routine typically calls *sysSerialHwInit***( )** which initializes all the values in the **M68360_CHAN** structure (except the **SIO_DRV_FUNCS**) before calling *m68360DevInit***( )**. The BSP's

sysHwInit2( ) routine typically calls *sysSerialHwInit2*( ) which connects the chips interrupt (m68360Int) via *intConnect*( ).

INCLUDE FILES **drv/sio/m68360Sio.h**

m68562Sio

NAME **m68562Sio** – MC68562 DUSCC serial driver

SYNOPSIS *m68562HrdInit*( ) – initialize the DUSCC
m68562RxTxErrInt( ) – handle a receiver / transmitter error interrupt
m68562RxInt( ) – handle a receiver interrupt
m68562TxInt( ) – handle a transmitter interrupt

```
void m68562HrdInit
    (M68562_QUSART *pQusart)

void m68562RxTxErrInt
    (M68562_CHAN *pChan)

void m68562RxInt
    (M68562_CHAN *pChan)

void m68562TxInt
    (M68562_CHAN *pChan)
```

DESCRIPTION This is the driver for the MC68562 DUSCC serial chip. It uses the DUSCC in asynchronous mode only.

USAGE A **M68562_QUSART** structure is used to describe the chip. This data structure contains **M68562_CHAN** structures which describe the chip's serial channels. The BSP's *sysHwInit*( ) routine typically calls *sysSerialHwInit*( ) which initializes all the values in the **M68562_QUSART** structure (except the **SIO_DRV_FUNCS**) before calling *m68562HrdInit*( ). The BSP's *sysHwInit2*( ) routine typically calls *sysSerialHwInit2*( ) which connects the chips interrupts (m68562RxTxErrInt, m68562RxInt, and m68562TxInt) via *intConnect*( ).

IOCTL This driver responds to the same *ioctl*( ) codes as a normal serial driver. See the file **sioLib.h** for more information.

INCLUDE FILES **drv/sio/m68562Sio.h**

m68681Sio

NAME

m68681Sio – M68681 serial communications driver

SYNOPSIS

*m68681DevInit***( )** – intialize a **M68681_DUART**
*m68681DevInit2***( )** – intialize a **M68681_DUART**, part 2
*m68681ImrSetClr***( )** – set and clear bits in the DUART interrupt-mask register
*m68681Imr***( )** – return the current contents of the DUART interrupt-mask register
*m68681AcrSetClr***( )** – set and clear bits in the DUART auxiliary control register
*m68681Acr***( )** – return the contents of the DUART auxiliary control register
*m68681OprSetClr***( )** – set and clear bits in the DUART output port register
*m68681Opr***( )** – return the current state of the DUART output port register
*m68681OpcrSetClr***( )** – set and clear bits in the DUART output port configuration register
*m68681Opcr***( )** – return the state of the DUART output port configuration register
*m68681Int***( )** – handle all DUART interrupts in one vector

```
void m68681DevInit
     (M68681_DUART * pDuart)

void m68681DevInit2
     (M68681_DUART * pDuart)

void m68681ImrSetClr
     (M68681_DUART * pDuart, UCHAR setBits, UCHAR clearBits)

UCHAR m68681Imr
     (M68681_DUART * pDuart)

void m68681AcrSetClr
     (M68681_DUART * pDuart, UCHAR setBits, UCHAR clearBits)

UCHAR m68681Acr
     (M68681_DUART * pDuart)

void m68681OprSetClr
     (M68681_DUART * pDuart, UCHAR setBits, UCHAR clearBits)

UCHAR m68681Opr
     (M68681_DUART * pDuart)

void m68681OpcrSetClr
     (M68681_DUART * pDuart, UCHAR setBits, UCHAR clearBits)

UCHAR m68681Opcr
     (M68681_DUART * pDuart)

void m68681Int
     (M68681_DUART * pDuart)
```

DESCRIPTION

This is the driver for the M68681 DUART. This device includes two universal asynchronous receiver/transmitters, a baud rate generator, and a counter/timer device. This driver module provides control of the two serial channels and the baud-rate generator. The counter timer is controlled by a separate driver, **src/drv/timer/m68681Timer.c**.

A **M68681_DUART** structure is used to describe the chip. This data structure contains two **M68681_CHAN** structures which describe the chip's two serial channels. The **M68681_DUART** structure is defined in **m68681Sio.h**.

Only asynchronous serial operation is supported by this driver. The default serial settings are 8 data bits, 1 stop bit, no parity, 9600 baud, and software flow control. These default settings can be overridden on a channel-by-channel basis by setting the **M68681_CHAN** options and **baudRate** fields to the desired values before calling *m68681DevInit***( )**. See **sioLib.h** for option values. The defaults for the module can be changed by redefining the macros **M68681_DEFAULT_OPTIONS** and **M68681_DEFAULT_BAUD** and recompiling this driver.

This driver supports baud rates of 75, 110, 134.5, 150, 300, 600, 1200, 2000, 2400, 4800, 1800, 9600, 19200, and 38400.

USAGE

The BSP's *sysHwInit***( )** routine typically calls *sysSerialHwInit***( )** which initializes all the hardware addresses in the **M68681_DUART** structure before calling *m68681DevInit***( )**. This enables the chip to operate in polled mode, but not in interrupt mode. Calling *m68681DevInit2***( )** from the *sysSerialHwInit2***( )** routine allows interrupts to be enabled and interrupt-mode operation to be used.

The following example shows the first part of the initialization thorugh calling *m68681DevInit***( )**:

```
include "drv/sio/m68681Sio.h"
M68681_DUART myDuart;   /* my device structure */
define MY_VEC  (71)    /* use single vector, #71 */
sysSerialHwInit()
    {
    /* initialize the register pointers for portA */
    myDuart.portA.mr    = M68681_MRA;
    myDuart.portA.sr    = M68681_SRA;
    myDuart.portA.csr   = M68681_CSRA;
    myDuart.portA.cr    = M68681_CRA;
    myDuart.portA.rb    = M68681_RHRA;
    myDuart.portA.tb    = M68681_THRA;
    /* initialize the register pointers for portB */
    myDuart.portB.mr = M68681_MRB;
    ...
    /* initialize the register pointers/data for main duart */
    myDuart.ivr         = MY_VEC;
    myDuart.ipcr        = M68681_IPCR;
```

```
    myDuart.acr         = M68681_ACR;
    myDuart.isr         = M68681_ISR;
    myDuart.imr         = M68681_IMR;
    myDuart.ip          = M68681_IP;
    myDuart.opcr        = M68681_OPCR;
    myDuart.sopbc       = M68681_SOPBC;
    myDuart.ropbc       = M68681_ROPBC;
    myDuart.ctroff      = M68681_CTROFF;
    myDuart.ctron       = M68681_CTRON;
    myDuart.ctlr        = M68681_CTLR;
    myDuart.ctur        = M68681_CTUR;
    m68681DevInit (&myDuart);
    }
```

The BSP's *sysHwInit2***( )** routine typically calls *sysSerialHwInit2***( )** which connects the chips interrupts via *intConnect***( )** to the single interrupt handler *m68681Int***( )**. After the interrupt service routines are connected, the user then calls *m68681DevInit2***( )** to allow the driver to turn on interrupt enable bits, as shown in the following example:

```
sysSerialHwInit2 ()
    {
    /* connect single vector for 68681 */
    intConnect (INUM_TO_IVEC(MY_VEC), m68681Int, (int)&myDuart);
    ...
    /* allow interrupts to be enabled */
    m68681DevInit2 (&myDuart);
    }
```

SPECIAL CONSIDERATIONS

The CLOCAL hardware option presumes that OP0 and OP1 output bits are wired to the CTS outputs for channel 0 and channel 1 respectively. If not wired correctly, then the user must not select the CLOCAL option. CLOCAL is not one of the default options for this reason.

This driver does not manipulate the output port or its configuration register in any way. If the user selects the CLOCAL option, then the output port bit must be wired correctly or the hardware flow control will not function correctly.

INCLUDE FILES

drv/sio/m68681Sio.h

m68901Sio

NAME **m68901Sio** – MC68901 MFP tty driver

SYNOPSIS *m68901DevInit***( )** – initialize a **M68901_CHAN** structure

```
void m68901DevInit
    (M68901_CHAN * pChan)
```

DESCRIPTION This is the SIO driver for the Motorola MC68901 Multi-Function Peripheral (MFP) chip.

USER-CALLABLE ROUTINES Most of the routines in this driver are accessible only through the I/O system. However, one routine must be called directly: *m68901DevInit***( )** initializes the driver. Normally, it is called by *sysSerialHwInit***( )** in **sysSerial.c**

IOCTL FUNCTIONS This driver responds to the same *ioctl***( )** codes as other tty drivers; for more information, see the manual entry for **tyLib**.

SEE ALSO **tyLib**

mathALib

NAME **mathALib** – C interface library to high-level math functions

SYNOPSIS
*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

```
double acos
     (double x)

double asin
     (double x)

double atan
     (double x)
```

```
double atan2
    (double y, double x)

double cbrt
    (double x)

double ceil
    (double v)

double cos
    (double x)

double cosh
    (double x)

double exp
    (double x)

double fabs
    (double v)

double floor
    (double v)

double fmod
    (double x, double y)

double infinity
    (void)

int irint
    (double x)

int iround
    (double x)

double log
    (double x)

double log10
    (double x)

double log2
    (double x)

double pow
    (double x, double y)

double round
    (double x)

double sin
    (double x)
```

```
void sincos
     (double x, double *sinResult, double *cosResult)

double sinh
     (double x)

double sqrt
     (double x)

double tan
     (double x)

double tanh
     (double x)

double trunc
     (double x)

float acosf
     (float x)

float asinf
     (float x)

float atanf
     (float x)

float atan2f
     (float y, float x)

float cbrtf
     (float x)

float ceilf
     (float v)

float cosf
     (float x)

float coshf
     (float x)

float expf
     (float x)

float fabsf
     (float v)

float floorf
     (float v)

float fmodf
     (float x, float y)
```

```
float infinityf
    (void)

int irintf
    (float x)

int iroundf
    (float x)

float logf
    (float x)

float log10f
    (float x)

float log2f
    (float x)

float powf
    (float x, float y)

float roundf
    (float x)

float sinf
    (float x)

void sincosf
    (float x, float *sinResult, float *cosResult)

float sinhf
    (float x)

float sqrtf
    (float x)

float tanf
    (float x)

float tanhf
    (float x)

float truncf
    (float x)
```

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. The architecture-specific appendices of the *VxWorks Programmer's Guide* list any math functions that are not available.

INCLUDE FILES **math.h**

SEE ALSO **ansiMath**, **fppLib**, **floatLib**, **mathHardLib**, **mathSoftLib**, Kernighan & Ritchie: *The C Programming Language, 2nd Edition*, *VxWorks Programmer's Guide:* Architecture-specific Appendices

mathHardLib

NAME **mathHardLib** – hardware floating-point math library

SYNOPSIS *mathHardInit***( )** – initialize hardware floating-point math support

```
void mathHardInit()
```

DESCRIPTION This library provides support routines for using hardware floating-point units with high-level math functions. The high-level functions include triginometric 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

SYNOPSIS *mathSoftInit***( )** – initialize software floating-point math support

```
void mathSoftInit()
```

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

mb86940Sio

NAME **mb86940Sio** – MB 86940 UART tty driver

SYNOPSIS *mb86940DevInit***( )** – install the driver function table

```
void mb86940DevInit
    (MB86940_CHAN *pChan)
```

DESCRIPTION This is the driver for the SPARClite MB86930 on-board serial ports.

USAGE A **MB86940_CHAN** structure is used to describe the chip.

The BSP's *sysHwInit***( )** routine typically calls *sysSerialHwInit***( )**, which initializes all the values in the **MB86940_CHAN** structure (except the **SIO_DRV_FUNCS**) before calling *mb86940DevInit***( )**. The BSP's *sysHwInit2***( )** routine typically calls *sysSerialHwInit2***( )**, which connects the chips interrupts via *intConnect***( )**.

IOCTL FUNCTIONS The UARTs use timer 3 output to generate the following baud rates: 110, 150, 300, 600, 1200, 2400, 4800, 9600, and 19200. Note that the UARTs will operate at the same baud rate.

INCLUDE FILES **drv/sio/mb86940Sio.h**

mb87030Lib

NAME **mb87030Lib** – Fujitsu MB87030 SCSI Protocol Controller (SPC) library

SYNOPSIS

*mb87030CtrlCreate***( )** – create a control structure for an MB87030 SPC
*mb87030CtrlInit***( )** – initialize a control structure for an MB87030 SPC
*mb87030Show***( )** – display the values of all readable MB87030 SPC registers

```
MB_87030_SCSI_CTRL *mb87030CtrlCreate
    (UINT8 *spcBaseAdrs, int regOffset, UINT clkPeriod, int spcDataParity,
    FUNCPTR spcDMABytesIn, FUNCPTR spcDMABytesOut)

STATUS mb87030CtrlInit
    (MB_87030_SCSI_CTRL *pSpc, int scsiCtrlBusId, UINT defaultSelTimeOut,
    int scsiPriority)

STATUS mb87030Show
    (SCSI_CTRL *pScsiCtrl)
```

DESCRIPTION This is the I/O driver for the Fujitsu MB87030 SCSI Protocol Controller (SPC) chip. It is designed to work in conjunction with **scsiLib**.

USER-CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. Two routines, however, must be called directly: *mb87030CtrlCreate***( )** to create a controller structure, and *mb87030CtrlInit***( )** to initialize the controller structure.

INCLUDE FILES **mb87030.h**

SEE ALSO **scsiLib**, *Fujitsu Small Computer Systems Interface MB87030 Synchronous/Asynchronous Protocol Controller Users Manual, VxWorks Programmer's Guide: I/O System*

memDrv

NAME **memDrv** – pseudo memory device driver

SYNOPSIS *memDrv***( )** – install a memory driver
*memDevCreate***( )** – create a memory device

```
STATUS memDrv
    (void)

STATUS memDevCreate
    (char * name, char * base, int length)
```

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. This driver does not implement a file system as does **ramDrv**. The **ramDrv** driver must be given memory over which it has absolute control; **memDrv** simply provides a high-level method of reading and writing bytes in absolute memory locations through I/O calls.

USER-CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. Two routines, however, must be called directly: *memDrv***( )** to initialize the driver, and *memDevCreate***( )** to create 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 The memory driver responds to the *ioctl***( )** codes **FIOSEEK** and **FIOWHERE**.

SEE ALSO *VxWorks Programmer's Guide: I/O System*

memLib

NAME **memLib** – full-featured memory partition manager

SYNOPSIS *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

```
STATUS memPartOptionsSet
    (PART_ID partId, unsigned options)

void *memalign
    (unsigned alignment, unsigned size)

void * valloc
    (unsigned size)

void *memPartRealloc
    (PART_ID partId, char *pBlock, unsigned nBytes)

int memPartFindMax
    (PART_ID partId)

void memOptionsSet
    (unsigned options)

void *calloc
    (size_t elemNum, size_t elemSize)

void *realloc
    (void *pBlock, size_t newSize)

STATUS cfree
    (char *pBlock)

int memFindMax
    (void)
```

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 options 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

SYNOPSIS

*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)

```
PART_ID memPartCreate
    (char *pPool, unsigned poolSize)

STATUS memPartAddToPool
    (PART_ID partId, char *pPool, unsigned poolSize)

void *memPartAlignedAlloc
    (PART_ID partId, unsigned nBytes, unsigned alignment)

void *memPartAlloc
    (PART_ID partId, unsigned nBytes)

STATUS memPartFree
    (PART_ID partId, char *pBlock)

void memAddToPool
    (char *pPool, unsigned poolSize)

void *malloc
    (size_t nBytes)

void free
    (void *ptr)
```

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**, which is declared 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
68K	4	8
SPARC	8	12
MIPS	8	12
i960	16	16

INCLUDE FILES **memLib.h**, **stdlib.h**

SEE ALSO **memLib**, **smMemLib**

memShow

NAME **memShow** – memory show routines

SYNOPSIS

*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

```
void memShowInit
    (void)

void memShow
    (int type)

STATUS memPartShow
    (PART_ID partId, int type)
```

```
STATUS memPartInfoGet
    (PART_ID partId, MEM_PART_STATS * ppartStats)
```

DESCRIPTION This library contains memory partition information display routines.

SEE ALSO **memLib**, **memPartLib**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

---

mmanPxLib

NAME **mmanPxLib** – memory management library (POSIX)

SYNOPSIS *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)

```
int mlockall
    (int flags)

int munlockall
    (void)

int mlock
    (const void * addr, size_t len)

int munlock
    (const void * addr, size_t len)
```

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

mmuL64862Lib

NAME **mmuL64862Lib** – LSI Logic L64862 MBus-to-SBus Interface: I/O DMA library (SPARC)

SYNOPSIS *mmuL64862DmaInit***( )** – initialize the L64862 I/O MMU DMA data structures (SPARC)

```
STATUS mmuL64862DmaInit
    (void *vrtBase, void *vrtTop, UINT range)
```

DESCRIPTION This library contains the architecture-specific routine *mmuL64862DmaInit***( )**, needed to set up the I/O mapping for S-Bus DMA devices using the LSI Logic L64862 architecture.

INCLUDE FILES **arch/sparc/l64862.h**

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

mmuSparcILib

NAME **mmuSparcILib** – ROM MMU initialization (SPARC)

SYNOPSIS *mmuSparcRomInit***( )** – initialize the MMU for the ROM (SPARC)

```
STATUS mmuSparcRomInit
    (int * mmuTableAdrs, int mmuRomPhysAdrs, int romInitAdrs)
```

DESCRIPTION This library contains routines that are called by SPARC boot ROMs to initialize the translation tables while still in "boot state." When the board comes up, all instruction fetches from the boot ROMs bypass the MMU, thus allowing code in the ROMs to initialize the MMU tables with mappings for RAM, I/O devices, and other memory devices.

*mmuSparcRomInit***( )** is called from *romInit***( )**. The translation tables are initialized according to the mappings found in **sysPhysMemDesc**, which is contained in **memDesc.c** in the BSP. Note that these mappings are also used by **vmLib** or **vmBaseLib** when VxWorks creates global virtual memory at system initialization time. New ROMs may need to be built if these tables are modified.

moduleLib

NAME

moduleLib – object module management library

SYNOPSIS

*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

```
MODULE_ID moduleCreate
    (char * name, int format, int flags)

STATUS moduleDelete
    (MODULE_ID moduleId)

STATUS moduleShow
    (char * moduleNameOrId, int options)

SEGMENT_ID moduleSegGet
    (MODULE_ID moduleId)

SEGMENT_ID moduleSegFirst
    (MODULE_ID moduleId)

SEGMENT_ID moduleSegNext
    (SEGMENT_ID segmentId)

STATUS moduleCreateHookAdd
    (FUNCPTR moduleCreateHookRtn)

STATUS moduleCreateHookDelete
    (FUNCPTR moduleCreateHookRtn)

MODULE_ID moduleFindByName
    (char * moduleName)
```

```
MODULE_ID moduleFindByNameAndPath
    (char * moduleName, char * pathName)

MODULE_ID moduleFindByGroup
    (int groupNumber)

int moduleIdListGet
    (MODULE_ID * idList, int maxModules)

STATUS moduleInfoGet
    (MODULE_ID moduleId, MODULE_INFO * pModuleInfo)

STATUS moduleCheck
    (int options)

char * moduleNameGet
    (MODULE_ID moduleId)

int moduleFlagsGet
    (MODULE_ID moduleId)
```

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 information about *symFlag* and the loader, see the reference 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

SYNOPSIS

*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

```
STATUS mountdInit
    (int priority, int stackSize, FUNCPTR authHook, int nExports,
    int options)

STATUS nfsExport
    (char * directory, int id, BOOL readOnly, int options)

STATUS nfsUnexport
    (char * dirName)
```

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** in **configAll.h** calls *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 **dosFsLib** file systems are supported; RT11 file systems cannot be exported. 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, define **INCLUDE_NFS_SERVER** in your **configAll.h** file, and rebuild VxWorks.

To initialize a file system that is to be exported, set the **DOS_OPT_EXPORT** option in the **DOS_VOL_CONFIG** structure used for initialization. You can do this directly in the *dosFsDevInit***( )** call, or indirectly with *dosFsDevInitOptionsSet***( )** or *dosFsMkfsOptionsSet***( )**.

Example

The following example illustrates how to initialize and export an existing dosFs file system.

First, initialize the block device containing your file system (identified by *pBlockDevice* below).

Then execute the following code on the target:

```
dosFsDevInitOptionsSet (DOS_OPT_EXPORT);      /* make exportable */
dosFsDevInit ("/export", pBlockDevice, NULL); /* initialize on VxWorks */
nfsExport ("/export", 0, FALSE, 0);           /* make available remotely */
```

This initializes the DOS file system, and 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.)

Note that DOS file names are normally limited to 8 characters with a three character extension. You can use an additional initialization option, **DOS_OPT_LONGNAMES**, to enable the VxWorks extension that allows file names up to forty characters long. Replace the *dosFsDevInitOptionsSet***( )** call in the example above with the following:

```
dosFsMkfsOptionsSet (DOS_OPT_EXPORT | DOS_OPT_LONGNAMES);
```

The variables **dosFsUserId**, **dosFsGroupId**, and **dosFsFileMode** can be set before initialization to specify ownership and permissions as reported over NFS, but they are not required. The defaults appear in the **dosFsLib** manual entry. DOS file systems do not provide for permissions, user IDs, and group IDs on a per-file basis; these variables specify this information for all files on an entire DOS file system.

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)

SYNOPSIS

*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)

```
int mqPxLibInit
    (int hashSize)

mqd_t mq_open
    (const char *mqName, int oflags, ...)

ssize_t mq_receive
    (mqd_t mqdes, void *pMsg, size_t msgLen, int *pMsgPrio)

int mq_send
    (mqd_t mqdes, const void *pMsg, size_t msgLen, int msgPrio)

int mq_close
    (mqd_t mqdes)

int mq_unlink
    (const char * mqName)

int mq_notify
    (mqd_t mqdes, const struct sigevent * pNotification)

int mq_setattr
    (mqd_t mqdes, const struct mq_attr * pMqStat,
    struct mq_attr * pOldMqStat)

int mq_getattr
    (mqd_t mqdes, struct mq_attr * pMqStat)
```

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.

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

SYNOPSIS *mqPxShowInit*( ) – initialize the POSIX message queue show facility

```
STATUS mqPxShowInit
    (void)
```

DESCRIPTION This library provides a show routine for POSIX objects.

msgQLib

NAME **msgQLib** – message queue library

SYNOPSIS *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

```
MSG_Q_ID msgQCreate
    (int maxMsgs, int maxMsgLength, int options)

STATUS msgQDelete
    (MSG_Q_ID msgQId)

STATUS msgQSend
    (MSG_Q_ID msgQId, char * buffer, UINT nBytes, int timeout, int priority)

int msgQReceive
    (MSG_Q_ID msgQId, char * buffer, UINT maxNBytes, int timeout)

int msgQNumMsgs
    (MSG_Q_ID msgQId)
```

DESCRIPTION

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

CREATING AND USING MESSAGE QUEUES

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

INCLUDE FILES **msgQLib.h**

SEE ALSO **pipeDrv**, **msgQSmLib**, *VxWorks Programmer's Guide: Basic OS*

msgQShow

NAME **msgQShow** – message queue show routines

SYNOPSIS

msgQShowInit( ) – initialize the message queue show facility
msgQInfoGet( ) – get information about a message queue
msgQShow( ) – show information about a message queue

```
void msgQShowInit
    (void)

STATUS msgQInfoGet
    (MSG_Q_ID msgQId, MSG_Q_INFO * pInfo)

STATUS msgQShow
    (MSG_Q_ID msgQId, int level)
```

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 **INCLUDE_SHOW_ROUTINES** is defined in **configAll.h**.

INCLUDE FILES **msgQLib.h**

SEE ALSO **pipeDrv**, *VxWorks Programmer's Guide: Basic OS*

msgQSmLib

NAME **msgQSmLib** – shared memory message queue library (VxMP Opt.)

SYNOPSIS *msgQSmCreate***( )** – create and initialize a shared memory message queue

```
MSG_Q_ID msgQSmCreate
    (int maxMsgs, int maxMsgLength, int options)
```

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**, defined in **configAll.h**.

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***( )**, which is found in **src/config/usrSmObj.c**. This is done automatically from the root task, *usrRoot***( )**, in **usrConfig.c** if **INCLUDE_SM_OBJ** is defined in **configAll.h**.

AVAILABILITY This module is provided with the unbundled shared memory objects 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*

ncr5390Lib

NAME **ncr5390Lib** – NCR5390 SCSI-Bus Interface Controller library (SBIC)

SYNOPSIS *ncr5390CtrlInit***( )** – initialize the user-specified fields in an ASC structure
*ncr5390Show***( )** – display the values of all readable NCR5390 chip registers

```
STATUS ncr5390CtrlInit
    (int *pAsc, int scsiCtrlBusId, UINT defaultSelTimeOut, int scsiPriority)

int ncr5390Show
    (int *pScsiCtrl)
```

DESCRIPTION This library contains the main interface routines to the SCSI-Bus Interface Controllers (SBIC). These routines simply switch the calls to the SCSI-1 or SCSI-2 drivers, implemented in **ncr5390Lib1.c** or **ncr5390Lib2.c** as configured by the BSP.

In order to configure the SCSI-1 driver, which depends upon **scsi1Lib**, the *ncr5390CtrlCreate***( )** routine, defined in **ncr5390Lib1**, must be invoked. Similarly *ncr5390CtrlCreateScsi2***( )**, defined in **ncr5390Lib2** and dependent on **scsi2Lib**, must be called to configure and initialize the SCSI-2 driver.

INCLUDE FILES **ncr5390.h**, **ncr5390_1.h**, **ncr5390_2.h**

ncr5390Lib1

NAME **ncr5390Lib1** – NCR 53C90 Advanced SCSI Controller (ASC) library (SCSI-1)

SYNOPSIS *ncr5390CtrlCreate***( )** – create a control structure for an NCR 53C90 ASC

```
NCR_5390_SCSI_CTRL *ncr5390CtrlCreate
    (UINT8 *baseAdrs, int regOffset, UINT clkPeriod, FUNCPTR ascDmaBytesIn,
    FUNCPTR ascDmaBytesOut)
```

DESCRIPTION This is the I/O driver for the NCR 53C90 Advanced SCSI Controller (ASC). It is designed to work in conjunction with **scsiLib**.

USER-CALLABLE ROUTINES

Most routines in this driver are accessible only through the I/O system. The only exception is the *ncr5390CtrlCreate***( )** which creates a controller structure.

INCLUDE FILES **ncr5390.h**

SEE ALSO **scsiLib**, *NCR 53C90A, 53C90B Advanced SCSI Controller, VxWorks Programmer's Guide: I/O System*

ncr5390Lib2

NAME **ncr5390Lib2** – NCR 53C90 Advanced SCSI Controller (ASC) library (SCSI-2)

SYNOPSIS *ncr5390CtrlCreateScsi2***( )** – create a control structure for an NCR 53C90 ASC

```
NCR_5390_SCSI_CTRL *ncr5390CtrlCreateScsi2
    (UINT8* baseAdrs, int regOffset, UINT clkPeriod,
    UINT sysScsiDmaMaxBytes, FUNCPTR sysScsiDmaStart,
    FUNCPTR sysScsiDmaAbort, int sysScsiDmaArg)
```

DESCRIPTION This is the I/O driver for the NCR 53C90 Advanced SCSI Controller (ASC). It is designed to work in conjunction with **scsiLib**.

USER-CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. The only exception in this portion of the driver is the *ncr5390CtrlCreateScsi2***( )** which creates a controller structure.

INCLUDE FILES **ncr5390.h**

SEE ALSO **scsiLib**, *NCR 53C90A, 53C90B Advanced SCSI Controller, VxWorks Programmer's Guide: I/O System*

ncr710Lib

NAME

ncr710Lib – NCR 53C710 SCSI I/O Processor (SIOP) library (SCSI-1)

SYNOPSIS

*ncr710CtrlCreate***( )** – create a control structure for an NCR 53C710 SIOP
*ncr710CtrlInit***( )** – initialize a control structure for an NCR 53C710 SIOP
*ncr710SetHwRegister***( )** – set hardware-dependent registers for the NCR 53C710 SIOP
*ncr710Show***( )** – display the values of all readable NCR 53C710 SIOP registers

```
NCR_710_SCSI_CTRL *ncr710CtrlCreate
    (UINT8 *baseAdrs, UINT freqValue)

STATUS ncr710CtrlInit
    (NCR_710_SCSI_CTRL *pSiop, int scsiCtrlBusId, int scsiPriority)

STATUS ncr710SetHwRegister
    (SIOP *pSiop, NCR710_HW_REGS *pHwRegs)

STATUS ncr710Show
    (SCSI_CTRL *pScsiCtrl)
```

DESCRIPTION

This is the I/O driver for the NCR 53C710 SCSI I/O Processor (SIOP). It is designed to work with **scsi1Lib**. It also runs in conjunction with a script program for the NCR 53C710 chip. This script uses the NCR 53C710 DMA function for data transfers. This driver supports cache functions through **cacheLib**.

USER-CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. Three routines, however, must be called directly: *ncr710CtrlCreate***( )** to create a controller structure, and *ncr710CtrlInit***( )** to initialize it. The NCR 53C710 hardware registers need to be configured according to the hardware implementation. If the default configuration is not proper, the routine *ncr710SetHwRegister***( )** should be used to properly configure the registers.

INCLUDE FILES

ncr710.h, **ncr710_1.h**, **ncr710Script.h**, **ncr710Script1.h**

SEE ALSO

scsiLib, **scsi1Lib**, **cacheLib**, *NCR 53C710 SCSI I/O Processor Programming Guide, VxWorks Programmer's Guide: I/O System*

ncr710Lib2

NAME

ncr710Lib2 – NCR 53C710 SCSI I/O Processor (SIOP) library (SCSI-2)

SYNOPSIS

*ncr710CtrlCreateScsi2***( )** – create a control structure for the NCR 53C710 SIOP
*ncr710CtrlInitScsi2***( )** – initialize a control structure for the NCR 53C710 SIOP
*ncr710SetHwRegisterScsi2***( )** – set hardware-dependent registers for the NCR 53C710
*ncr710ShowScsi2***( )** – display the values of all readable NCR 53C710 SIOP registers

```
NCR_710_SCSI_CTRL *ncr710CtrlCreateScsi2
    (UINT8 *baseAdrs, UINT clkPeriod)

STATUS ncr710CtrlInitScsi2
    (NCR_710_SCSI_CTRL *pSiop, int scsiCtrlBusId, int scsiPriority)

STATUS ncr710SetHwRegisterScsi2
    (SIOP *pSiop, NCR710_HW_REGS *pHwRegs)

STATUS ncr710ShowScsi2
    (SCSI_CTRL *pScsiCtrl)
```

DESCRIPTION

This is the I/O driver for the NCR 53C710 SCSI I/O Processor (SIOP). It is designed to work with **scsi2Lib**. This driver runs in conjunction with a script program for the NCR 53C710 chip. The script uses the NCR 53C710 DMA function for data transfers. This driver supports cache functions through **cacheLib**.

USER-CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. Three routines, however, must be called directly. *ncr710CtrlCreateScsi2***( )** creates a controller structure and *ncr710CtrlInitScsi2***( )** initializes it. The NCR 53C710 hardware registers need to be configured according to the hardware implementation. If the default configuration is not correct, the routine *ncr710SetHwRegisterScsi2***( )** must be used to properly configure the registers.

INCLUDE FILES

ncr710.h, **ncr710_2.h**, **ncr710Script.h**, **ncr710Script2.h**

SEE ALSO

scsiLib, **scsi2Lib**, **cacheLib**, *VxWorks Programmer's Guide: I/O System*

ncr810Lib

NAME

ncr810Lib – NCR 53C8xx PCI SCSI I/O Processor (SIOP) library (SCSI-2)

SYNOPSIS

*ncr810CtrlCreate***( )** – create a control structure for the NCR 53C8xx SIOP
*ncr810CtrlInit***( )** – initialize a control structure for the NCR 53C8xx SIOP
*ncr810SetHwRegister***( )** – set hardware-dependent registers for the NCR 53C8xx SIOP
*ncr810Show***( )** – display values of all readable NCR 53C8xx SIOP registers

```
NCR_810_SCSI_CTRL *ncr810CtrlCreate
    (UINT8 *baseAdrs, UINT clkPeriod, UINT16 devType)

STATUS ncr810CtrlInit
    (NCR_810_SCSI_CTRL *pSiop, int scsiCtrlBusId)

STATUS ncr810SetHwRegister
    (SIOP *pSiop, NCR810_HW_REGS *pHwRegs)

STATUS ncr810Show
    (SCSI_CTRL *pScsiCtrl)
```

DESCRIPTION

This is the I/O driver for the NCR 53C8xx PCI SCSI I/O Processors (SIOP), supporting the NCR 53C810 and the NCR 53C825 SCSI controllers. It is designed to work with **scsiLib** and **scsi2Lib**. This driver runs in conjunction with a script program for the NCR 53C8xx controllers. These scripts use DMA transfers for all data, messages, and status. This driver supports cache functions through **cacheLib**.

USER-CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. Three routines, however, must be called directly. *ncr810CtrlCreate***( )** creates a controller structure and *ncr810CtrlInit***( )** initializes it. The NCR 53C8xx hardware registers need to be configured according to the hardware implementation. If the default configuration is not correct, the routine *ncr810SetHwRegister***( )** must be used to properly configure the registers.

INCLUDE FILES

ncr810.h, **ncr810Script.h** and **scsiLib.h**

SEE ALSO

scsiLib, **scsi2Lib**, **cacheLib**, *SYM53C825 PCI-SCSI I/O Processor Data Manual, SYM53C810 PCI-SCSI I/O Processor Data Manual, NCR 53C8XX Family PCI-SCSI I/O Processors Programming Guide, VxWorks Programmer's Guide: I/O System*

nec765Fd

NAME **nec765Fd** – NEC 765 floppy disk device driver

SYNOPSIS *fdDrv***( )** – initialize the floppy disk driver
*fdDevCreate***( )** – create a device for a floppy disk
*fdRawio***( )** – provide raw I/O access

```
STATUS fdDrv
    (int vector, int level)

BLK_DEV *fdDevCreate
    (int drive, int fdType, int nBlocks, int blkOffset)

STATUS fdRawio
    (int drive, int fdType, FD_RAW *pFdRaw)
```

DESCRIPTION This is the driver for the NEC 765 Floppy Chip used on the PC 386/486.

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: *fdDrv***( )** to initialize the driver, and *fdDevCreate***( )** to create devices. Before the driver can be used, it must be initialized by calling *fdDrv***( )**. This routine should be called exactly once, before any reads, writes, or calls to *fdDevCreate***( )**. Normally, it is called from *usrRoot***( )** in **usrConfig.c**.

The routine *fdRawio***( )** allows physical I/O access. Its first argument is a drive number, 0 to 3; the second argument is a type of diskette; the third argument is a pointer to the **FD_RAW** structure, which is defined in **nec765Fd.h**.

Interleaving is not supported when the driver formats.

Two types of diskettes are currently supported: 3.5” 2HD 1.44MB and 5.25” 2HD 1.2MB. You can add additional diskette types to the **fdTypes[]** table in **sysLib.c**.

SEE ALSO *VxWorks Programmer’s Guide: I/O System*

netDrv

NAME

netDrv – network remote file I/O driver

SYNOPSIS

*netDrv***( )** – install the network remote file driver
*netDevCreate***( )** – create a remote file device

```
STATUS netDrv
    (void)

STATUS netDevCreate
    (char *devName, char *host, int protocol)
```

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, or *netDevCreate***( )** calls. Initialization is performed automatically when **INCLUDE_NETWORK** is defined in **configAll.h**.

CREATING NETWORK DEVICES

To access files on a remote host, a network device must be created by calling *netDevCreate***( )**. 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. 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 *netDevCreate***( )**.

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 fieds of the stat structure: st_dev, st_mode, and st_size.

```
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.

INCLUDE FILES

netDrv.h

SEE ALSO

remLib, **netLib**, **sockLib**, *hostAdd***( )**, *VxWorks Programmer’s Guide: Network*

netLib

NAME **netLib** – network interface library

SYNOPSIS *netLibInit***( )** – initialize the network package
*netTask***( )** – network task entry point

```
STATUS netLibInit
    (void)

void netTask
    (void)
```

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_NETWORK** is defined in **configAll.h**.

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***( )**, *VxWorks Programmer's Guide: Network*

netShow

NAME **netShow** – network information display routines

SYNOPSIS *ifShow***( )** – display the attached network interfaces
*icmpstatShow***( )** – display statistics for ICMP
*inetstatShow***( )** – display all active connections for Internet protocol sockets
*ipstatShow***( )** – display IP statistics
*mbufShow***( )** – report mbuf statistics
*netShowInit***( )** – initialize network show routines
*tcpDebugShow***( )** – display debugging information for the TCP protocol
*tcpstatShow***( )** – display all statistics for the TCP protocol
*udpstatShow***( )** – display statistics for the UDP protocol
*arpShow***( )** – display entries in the system ARP table

arptabShow( ) – display the known ARP entries
routestatShow( ) – display routing statistics
routeShow( ) – display host and network routing tables
hostShow( ) – display the host table

```
void ifShow
     (char *ifName)

void icmpstatShow
     (void)

void inetstatShow
     (void)

void ipstatShow
     (BOOL zero)

void mbufShow
     (void)

void netShowInit
     (void)

void tcpDebugShow
     (int numPrint, int verbose)

void tcpstatShow
     (void)

void udpstatShow
     (void)

void arpShow
     (void)

void arptabShow
     (void)

void routestatShow
     (void)

void routeShow
     (void)

void hostShow
     (void)
```

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 Doublas 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 in **configAll.h**.

SEE ALSO **ifLib**, *VxWorks Programmer's Guide: Network*

nfsdLib

NAME **nfsdLib** – Network File System (NFS) server library

SYNOPSIS *nfsdInit***( )** – initialize the NFS server
*nfsdStatusGet***( )** – get the status of the NFS server
*nfsdStatusShow***( )** – show the status of the NFS server

```
STATUS nfsdInit
    (int nServers, int nExportedFs, int priority, FUNCPTR authHook,
    FUNCPTR mountAuthHook, int options)

STATUS nfsdStatusGet
    (NFS_SERVER_STATUS * serverStats)

STATUS nfsdStatusShow
    (int options)
```

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***( )**. Normally, this is done by defining **INCLUDE_NFS_SERVER** in **configAll.h**, so that *nfsdInit***( )** is called during the boot process.

Currently, only **dosFsLib** file systems are supported; RT11 file systems cannot be exported. File systems are exported with the *nfsExport***( )** call.

To create and export a file system, define **INCLUDE_NFS_SERVER** in **configAll.h**, 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** in **configAll.h**, and rebuild VxWorks.

Use *nfsExport*( ) to export file systems. For an example, see **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 discussion about authorization hooks in the reference entries for *nfsdInit*( ) and *mountdInit*( ).

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. For more information, see the reference entry for **dosFsLib**.

TASKS

Several NFS tasks are created by *nfsdInit*( ):

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

SYNOPSIS

*nfsDrv***( )** – install 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

```
STATUS nfsDrv
    (void)

STATUS nfsMount
    (char *host, char *fileSystem, char *localName)

STATUS nfsMountAll
    (char *host, char *clientName, BOOL quiet)

void nfsDevShow
    (void)

STATUS nfsUnmount
    (char *localName)

int nfsDevListGet
    (unsigned long nfsDevList[], int listSize)

STATUS nfsDevInfoGet
    (unsigned long nfsDevHandle, NFS_DEV_INFO * pnfsInfo)
```

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 in **configAll.h**.

CREATING NFS DEVICES

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 following call creates the device **/d0/**:

from code: `nfsMount ("wrs", "/d0/", NULL);`

from the shell: `nfsMount "wrs", "/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);
```

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***( )**, *VxWorks Programmer's Guide: Network*

nfsLib

NAME

nfsLib – Network File System (NFS) library

SYNOPSIS

*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

```
void nfsHelp
    (void)

STATUS nfsExportShow
    (char *hostName)

void nfsAuthUnixPrompt
    (void)

void nfsAuthUnixShow
    (void)

void nfsAuthUnixSet
    (char *machname, int uid, int gid, int ngids, int *aup_gids)

void nfsAuthUnixGet
    (char *machname, int *pUid, int *pGid, int *pNgids, int *gids)

void nfsIdSet
    (int uid)
```

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. NFS is enabled when **INCLUDE_NFS** is defined in the VxWorks configuration header file **config/all/configAll.h**:

```
#define 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 *VxWorks Programmer's Guide: Network*.

Normal use of NFS requires no more than 2000 bytes of stack.

NFS USER IDENTIFICATION

NFS is built on top of RPC and uses a type of RPC authentication known as **AUTH_UNIX**, which is passed onto 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**, *VxWorks Programmer's Guide: Network*

ns16550Sio

NAME **ns16550Sio** – NS 16550 UART tty driver

SYNOPSIS

*ns16550DevInit***( )** – intialize an NS16550 channel
*ns16550IntWr***( )** – handle a transmitter interrupt
*ns16550IntRd***( )** – handle a receiver interrupt
*ns16550IntEx***( )** – miscellaneous interrupt processing
*ns16550Int***( )** – interrupt level processing

```
void ns16550DevInit
     (NS16550_CHAN * pChan)

void ns16550IntWr
     (NS16550_CHAN *pChan)

void ns16550IntRd
     (NS16550_CHAN *pChan)

void ns16550IntEx
     (NS16550_CHAN *pChan)

void ns16550Int
     (NS16550_CHAN *pChan)
```

DESCRIPTION This is the device driver for the ns16550 UART.

USAGE A **NS16550_CHAN** data structure is used to describe the chip.

The BSP's *sysHwInit***( )** routine typically calls *sysSerialHwInit***( )**, which initializes all the values in the **NS16550_CHAN** structure (except the **SIO_DRV_FUNCS**) before calling *ns16550DevInit***( )**. The BSP's *sysHwInit2***( )** routine typically calls *sysSerialHwInit2***( )**, which connects the chips interrupts via *intConnect***( )** (either the single interrupt **ns16550Int** or the three interrupts **ns16550IntWr**, **ns16550IntRd**, and **ns16550IntEx**).

INCLUDE FILES **drv/sio/ns16552Sio.h**

passFsLib

NAME **passFsLib** – pass-through (to UNIX) file system library (VxSim)

SYNOPSIS *passFsDevInit***( )** – associate a device with passFs file system functions
*passFsInit***( )** – prepare to use the passFs library

```
void *passFsDevInit
    (char *devName)

STATUS passFsInit
    (int nPassfs)
```

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**

pccardLib

NAME **pccardLib** – PC CARD enabler library

SYNOPSIS

*pccardMount***( )** – mount a DOS file system
*pccardMkfs***( )** – initialize a device and mount a DOS file system
*pccardAtaEnabler***( )** – enable the PCMCIA-ATA device
*pccardSramEnabler***( )** – enable the PCMCIA-SRAM driver
*pccardEltEnabler***( )** – enable the PCMCIA Etherlink III card

```
STATUS pccardMount
    (int sock, char *pName)

STATUS pccardMkfs
    (int sock, char *pName)

STATUS pccardAtaEnabler
    (int sock, ATA_RESOURCE *pAtaResource, int numEnt, FUNCPTR showRtn)

STATUS pccardSramEnabler
    (int sock, SRAM_RESOURCE *pSramResource, int numEnt, FUNCPTR showRtn)

STATUS pccardEltEnabler
    (int sock, ELT_RESOURCE *pEltResource, int numEnt, FUNCPTR showRtn)
```

DESCRIPTION This library provides generic facilities for enabling PC CARD. Each PC card device driver needs to provide an enabler routine and a CSC interrupt handler. The enabler routine must be in the **pccardEnabler** structure. Each PC card driver has its own resource

structure, **xxResources**. The ATA PC card driver resource structure is **ataResources** in **sysLib**, which also supports a local IDE disk. The resource structure has a PC card common resource structure in the first member. Other members are device-driver dependent resources.

The PCMCIA chip initialization routines *tcicInit***( )** and *pcicInit***( )** are included in the PCMCIA chip table **pcmciaAdapter**. This table is scanned when the PCMCIA library is initialized. If the initialization routine finds the PCMCIA chip, it registers all function pointers of the **PCMCIA_CHIP** structure.

A memory window defined in **pcmciaMemwin** is used to access the CIS of a PC card through the routines in **cisLib**.

SEE ALSO **pcmciaLib**, **cisLib**, **tcic**, **pcic**

pcic

NAME **pcic** – Intel 82365SL PCMCIA host bus adaptor chip library

SYNOPSIS *pcicInit***( )** – initialize the PCIC chip

```
STATUS pcicInit
    (int ioBase, int intVec, int intLevel, FUNCPTR showRtn)
```

DESCRIPTION This library contains routines to manipulate the PCMCIA functions on the Intel 82365 series PCMCIA chip. The following compatible chips are also supported:

– Cirrus Logic PD6712/20/22
– Vadem VG468
– VLSI 82c146
– Ricoh RF5C series

The initialization routine *pcicInit***( )** is the only global function and is included in the PCMCIA chip table **pcmciaAdapter**. If *pcicInit***( )** finds the PCIC chip, it registers all function pointers of the **PCMCIA_CHIP** structure.

pcicShow

NAME **pcicShow** – Intel 82365SL PCMCIA host bus adaptor chip show library

SYNOPSIS *pcicShow***( )** – show all configurations of the PCIC chip

```
void pcicShow
    (int sock)
```

DESCRIPTION This is a driver show routine for the Intel 82365 series PCMCIA chip. *pcicShow***( )** is the only global function and is installed in the PCMCIA chip table **pcmciaAdapter** in *pcmciaShowInit***( )**.

pcmciaLib

NAME **pcmciaLib** – generic PCMCIA event-handling facilities

SYNOPSIS *pcmciaInit***( )** – initialize the PCMCIA event-handling package
*pcmciad***( )** – handle task-level PCMCIA events

```
STATUS pcmciaInit
    (void)

void pcmciad
    (void)
```

DESCRIPTION This library provides generic facilities for handling PCMCIA events.

USER-CALLABLE ROUTINES

Before the driver can be used, it must be initialized by calling *pcmciaInit***( )**. This routine should be called exactly once, before any PC card device driver is used. Normally, it is called from *usrRoot***( )** in **usrConfig.c**.

The *pcmciaInit***( )** routine performs the following actions:

– Creates a message queue.

– Spawns a PCMCIA daemon, which handles jobs in the message queue.

– Finds out which PCMCIA chip is installed and fills out the **PCMCIA_CHIP** structure.

– Connects the CSC (Card Status Change) interrupt handler.

– Searches all sockets for a PC card. If a card is found, it:

- gets CIS (Card Information Structure) information from a card
- determines what type of PC card is in the socket
- allocates a resource for the card if the card is supported
- enables the card

– Enables the CSC interrupt.

The CSC interrupt handler performs the following actions:

– Searches all sockets for CSC events.

– Calls the PC card's CSC interrupt handler, if there is a PC card in the socket.

– If the CSC event is a hot insertion, it asks the PCMCIA daemon to call *cisGet***( )** at task level. This call reads the CIS, determines the type of PC card, and initializes a device driver for the card.

– If the CSC event is a hot removal, it asks the PCMCIA daemon to call *cisFree***( )** at task level. This call de-allocates resources.

pcmciaShow

NAME

pcmciaShow – PCMCIA show library

SYNOPSIS

*pcmciaShowInit***( )** – initialize all show routines for PCMCIA drivers
*pcmciaShow***( )** – show all configurations of the PCMCIA chip

```
void pcmciaShowInit
     (void)

void pcmciaShow
     (int sock)
```

DESCRIPTION

This library provides a show routine that shows the status of the PCMCIA chip and the PC card.

pingLib

NAME

pingLib – Packet InterNet Grouper (PING) library

SYNOPSIS

*pingLibInit***( )** – initialize the *ping***( )** utility
*ping***( )** – test that a remote host is reachable

```
STATUS pingLibInit
    (void)

STATUS ping
    (char * host, int numPackets, ulong_t options)
```

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 in **configAll.h**.

pipeDrv

NAME

pipeDrv – pipe I/O driver

SYNOPSIS

*pipeDrv***( )** – initialize the pipe driver
*pipeDevCreate***( )** – create a pipe device

```
STATUS pipeDrv
    (void)

STATUS pipeDevCreate
    (char *name, int nMessages, int nBytes)
```

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 **INCLUDE_PIPES** is defined in **configAll.h**.

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);
```

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*

ppc403Sio

NAME

ppc403Sio – ppc403GA serial driver

SYNOPSIS

*ppc403DummyCallback***( )** – dummy callback routine
*ppc403DevInit***( )** – initialize the serial port unit
*ppc403IntWr***( )** – handle a transmitter interrupt
*ppc403IntRd***( )** – handle a receiver interrupt
*ppc403IntEx***( )** – handle error interrupts

```
STATUS ppc403DummyCallback
    (void)

void ppc403DevInit
    (PPC403_CHAN * pChan)

void ppc403IntWr
    (PPC403_CHAN * pChan)

void ppc403IntRd
    (PPC403_CHAN * pChan)

void ppc403IntEx
    (PPC403_CHAN * pChan)
```

DESCRIPTION

This is the driver for PPC403GA serial port on the on-chip peripheral bus. The SPU (serial port unit) consists of three main elements: receiver, transmitter, and baud-rate generator. For details, refer to the *PPC403GA Embedded Controller User's Manual.*

USAGE

A **PPC403_CHAN** structure is used to describe the chip. This data structure contains the single serial channel. The BSP's *sysHwInit***( )** routine typically calls *sysSerialHwInit***( )** which initializes all the values in the **PPC403_CHAN** structure (except the **SIO_DRV_FUNCS**) before calling *ppc403DevInit***( )**. The BSP's *sysHwInit2***( )** routine typically calls *sysSerialHwInit2***( )** which connects the chip interrupt routines *ppc403IntWr***( )** and *ppc403IntRd***( )** via *intConnect***( )**.

IOCTL FUNCTIONS

This driver responds to the same *ioctl***( )** codes as other SIO drivers; for more information, see **sioLib.h**.

INCLUDE FILES

drv/sio/ppc403Sio.h

ppc860Sio

NAME

ppc860Sio – Motorola MPC800 SMC UART serial driver

SYNOPSIS

*ppc860DevInit***( )** – initialize the SMC
*ppc860Int***( )** – handle an SMC interrupt

```
void ppc860DevInit
    (PPC860SMC_CHAN *pChan)

void ppc860Int
    (PPC860SMC_CHAN *pChan)
```

DESCRIPTION

This is the driver for the SMCs in the internal Communications Processor (CP) of the Motorola MPC68860/68821. This driver only supports the SMCs in asynchronous UART mode.

USAGE

A **PPC800SMC_CHAN** structure is used to describe the chip. The BSP's *sysHwInit***( )** routine typically calls *sysSerialHwInit***( )**, which initializes all the values in the **PPC860SMC_CHAN** structure (except the **SIO_DRV_FUNCS**) before calling *ppc860DevInit***( )**.

The BSP's *sysHwInit2***( )** routine typically calls *sysSerialHwInit2***( )** which connects the chip's interrupts via *intConnect***( )**.

INCLUDE FILES

drv/sio/ppc860Sio.h

pppHookLib

NAME

pppHookLib – PPP hook library

SYNOPSIS

*pppHookAdd***( )** – add a hook routine on a unit basis
*pppHookDelete***( )** – delete a hook routine on a unit basis

```
STATUS pppHookAdd
    (int unit, FUNCPTR hookRtn, int hookType)

STATUS pppHookDelete
    (int unit, int hookType)
```

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 **INCLUDE_PPP** is defined in **configAll.h**.

INCLUDE FILES **pppLib.h**

SEE ALSO **pppLib**, *VxWorks Programmer's Guide: Network*

pppLib

NAME **pppLib** – Point-to-Point Protocol library

SYNOPSIS *pppInit***( )** – initialize a PPP network interface
*pppDelete***( )** – delete a PPP network interface

```
int pppInit
    (int unit, char *devname, char *local_addr, char *remote_addr,
    int baud, PPP_OPTIONS *pOptions, char *fOptions)

void pppDelete
    (int unit)
```

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 athentication, 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*'s ring buffer and processes it according to PPP's framing rules. The packet is then queued either to the IP's input queue or to the PPP daemon task's 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

SYNOPSIS *pppSecretAdd***( )** – add a secret to the PPP authentication secrets table
*pppSecretDelete***( )** – delete a secret from the PPP authentication secrets table

```
STATUS pppSecretAdd
    (char * client, char * server, char * secret, char * addrs)

STATUS pppSecretDelete
    (char * client, char * server, char * secret)
```

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 **INCLUDE_PPP** is defined in **configAll.h**.

INCLUDE FILES **pppLib.h**

SEE ALSO **pppLib**, **pppShow**, *VxWorks Programmer's Guide: Network*

pppShow

NAME

pppShow – Point-to-Point Protocol show routines

SYNOPSIS

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

```
void pppInfoShow
    (void)

STATUS pppInfoGet
    (int unit, PPP_INFO *pInfo)

void pppstatShow
    (void)

STATUS pppstatGet
    (int unit, PPP_STAT *pStat)

void pppSecretShow
    (void)
```

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 **INCLUDE_PPP** is defined in **configAll.h**.

INCLUDE FILES

pppLib.h

SEE ALSO

pppLib, *VxWorks Programmer's Guide: Network*

proxyArpLib

NAME

proxyArpLib – proxy Address Resolution Protocol (ARP) library

SYNOPSIS

*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 enabled ports

```
STATUS proxyArpLibInit
    (int clientSizeLog2, int portSizeLog2)

STATUS proxyNetCreate
    (char * proxyAddr, char * mainAddr)

STATUS proxyNetDelete
    (char * proxyAddr)

void proxyNetShow
    (void)

STATUS proxyPortFwdOn
    (int port)

STATUS proxyPortFwdOff
    (int port)

void proxyPortShow
    (void)
```

DESCRIPTION

This library provides transparent network access by using the Address Resolution Protocol (ARP) to make logically distinct networks appear as one logical network (i.e., the networks share the same address space). This module implements a proxy ARP scheme which provides an alternate method (to subnets) of access to the WRS backplane.

This module implements the proxy server. The proxy server is the multi-homed target which provides network transparency over the backplane by watching for and answering ARP requests.

This implementation supports only a single tier of backplane networks (i.e., only targets on directly attached interfaces are proxied for). Only one proxy server resides on a particular backplane network.

This library is initialized by calling *proxyArpLibInit***( )**. Proxy networks are created by calling *proxyNetCreate***( )** and deleted by calling *proxyNetDelete***( )**. The *proxyNetShow***( )** routine displays the proxy and main networks and the clients that reside on them.

A VxWorks backplane target registers itself as a target (proxy client) on the proxy network by calling *proxyReg***( )**. It unregisters itself by calling *proxyUnreg***( )**. These routines are provided in **proxyLib**.

To minimize and control backplane (proxy network) broadcast traffic, the proxy server must be configured to pass through broadcasts to a certain set of destination ports. Ports are enabled with the call *proxyPortFwdOn***( )** and are disabled with the call *proxyPortFwdOff***( )**. To see the ports currently enabled use *proxyPortShow***( )**. By default, only the BOOTP server port is enabled.

Refer to the *VxWorks Programmer's Guide* for more information about proxy ARP.

INCLUDE FILES **proxyArpLib.h**

SEE ALSO **proxyLib**, RFC 925, RFC 1027, RFC 826, *VxWorks Programmer's Guide: Network*

proxyLib

NAME **proxyLib** – proxy Address Resolution Protocol (ARP) client library

SYNOPSIS *proxyReg***( )** – register a proxy client
*proxyUnreg***( )** – unregister a proxy client

```
STATUS proxyReg
    (char * ifName, char * proxyAddr)

STATUS proxyUnreg
    (char * ifName, char * proxyAddr)
```

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*.

INCLUDE FILES **proxyArpLib.h**

SEE ALSO **proxyArpLib**, *VxWorks Programmer's Guide: Network*

ptyDrv

NAME

ptyDrv – pseudo-terminal driver

SYNOPSIS

*ptyDrv***( )** – initialize the pseudo-terminal driver
*ptyDevCreate***( )** – create a pseudo terminal

```
STATUS ptyDrv
    (void)

STATUS ptyDevCreate
    (char *name, int rdBufSize, int wrtBufSize)
```

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 routines in this driver are accessible only through the I/O system. However, two routines must be called directly: *ptyDrv***( )** to initialize the driver, and *ptyDevCreate***( )** to create devices.

INITIALIZING THE DRIVER

Before using the driver, it must be initialized by calling *ptyDrv***( )**. This routine must be called before any reads, writes, or calls to *ptyDevCreate***( )**.

CREATING PSEUDO-TERMINAL DEVICES

Before a pseudo-terminal can be used, it must be created by calling *ptyDevCreate***( )**:

```
STATUS ptyDevCreate
    (
    char  *name,      /* name of pseudo terminal      */
    int   rdBufSize,  /* size of terminal read buffer */
    int   wrtBufSize  /* size of write buffer         */
    )
```

For instance, to create the device pair "/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.

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*

ramDrv

NAME **ramDrv** – RAM disk driver

SYNOPSIS *ramDrv***( )** – prepare a RAM disk driver for use (optional)
*ramDevCreate***( )** – create a RAM disk device

```
STATUS ramDrv
    (void)

BLK_DEV *ramDevCreate
    (char *ramAddr, int bytesPerBlk, int blksPerTrack, int nBlocks,
    int blkOffset)
```

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

SYNOPSIS *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

```
RAW_VOL_DESC *rawFsDevInit
    (char *volName, BLK_DEV *pBlkDev)

STATUS rawFsInit
    (int maxFiles)

void rawFsModeChange
    (RAW_VOL_DESC *vdptr, int newMode)

void rawFsReadyChange
    (RAW_VOL_DESC *vdptr)

STATUS rawFsVolUnmount
    (RAW_VOL_DESC *vdptr)
```

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**.

To enable this initialization, define **INCLUDE_RAWFS** in **configAll.h**; *rawFsInit*( ) will then be 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 block device description structure (**BLK_DEV**). This must be initialized before calling *rawFsDevInit*( ). In the **BLK_DEV** structure, the driver includes the addresses of five routines 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 checks the status of the the device, and one that resets the device. The **BLK_DEV** structure also contains fields that describe the physical configuration of the device. For more information about defining block devices, see the *VxWorks Programmer's Guide: I/O System*.

The *rawFsDevInit*( ) routine is used to associate a device with the **rawFsLib** functions. The *volName* 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 *pBlkDev* parameter that *rawFsDevInit***( )** expects is a pointer to the **BLK_DEV** structure describing the device and contains the addresses of the required driver functions. The syntax of the *rawFsDevInit***( )** routine is as follows:

```
rawFsDevInit
    (
    char     *volName,  /* name to be used for volume   */
    BLK_DEV  *pBlkDev   /* pointer to device descriptor */
    )
```

Unlike the VxWorks DOS and RT-11 file systems, 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 entries for *dosFsMkfs***( )** and *rt11Mkfs***( )**).

When **rawFsLib** receives a request from the I/O system, after *rawFsDevInit***( )** has been called, it calls the device driver routines (whose addresses were passed in the **BLK_DEV** structure) to access the device.

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

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 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**, **dosFsLib**, **rt11FsLib**, **ramDrv**, *VxWorks Programmer's Guide: I/O System, Local File Systems*

rebootLib

NAME **rebootLib** – reboot support library

SYNOPSIS *reboot***( )** – reset network devices and transfer control to boot ROMs
*rebootHookAdd***( )** – add a routine to be called at reboot

```
void reboot
    (int startType)

STATUS rebootHookAdd
    (FUNCPTR rebootHook)
```

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

NAME

remLib – remote command library

SYNOPSIS

*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

```
int rcmd
    (char *host, int remotePort, char *localUser, char *remoteUser,
    char *cmd, int *fd2p)

int rresvport
    (int *alport)

void remCurIdGet
    (char *user, char *passwd)

STATUS remCurIdSet
    (char *newUser, char *newPasswd)

STATUS iam
    (char *newUser, char *newPasswd)

void whoami
    (void)

STATUS bindresvport
    (int sd, struct sockaddr_in *sin)
```

DESCRIPTION

This library provides routines to support remote command functions. The *rcmd***( )** and *rresvport***( )** routines use protocols implemented in UNIX BSD 4.3; they support remote command execution, and the opening of a socket with a bound privileged port, respectively. Other routines in this library authorize network file access via **netDrv**.

INCLUDE FILES

remLib.h

SEE ALSO

inetLib, *VxWorks Programmer's Guide: Network*

rlogLib

NAME

rlogLib – remote login library

SYNOPSIS

*rlogInit***( )** – initialize the remote login facility
*rlogind***( )** – the VxWorks remote login daemon
*rlogin***( )** – log in to a remote host

```
STATUS rlogInit
    (void)

void rlogind
    (void)

STATUS rlogin
    (char *host)
```

DESCRIPTION

This library provides a remote login facility for VxWorks that uses the UNIX **rlogin** protocol (as implemented in UNIX BSD 4.3) to allow users at a VxWorks terminal to log in to remote systems via the network, and users at remote systems to log in to VxWorks via the network.

A VxWorks user may log in to any other remote VxWorks or UNIX system via the network by calling *rlogin***( )** from the shell.

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 in **configAll.h**. 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

SYNOPSIS

*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

```
RING_ID rngCreate
    (int nbytes)

void rngDelete
    (RING_ID ringId)

void rngFlush
    (RING_ID ringId)

int rngBufGet
    (RING_ID rngId, char *buffer, int maxbytes)

int rngBufPut
    (RING_ID rngId, char *buffer, int nbytes)

BOOL rngIsEmpty
    (RING_ID ringId)

BOOL rngIsFull
    (RING_ID ringId)

int rngFreeBytes
    (RING_ID ringId)

int rngNBytes
    (RING_ID ringId)

void rngPutAhead
    (RING_ID ringId, char byte, int offset)

void rngMoveAhead
    (RING_ID ringId, int n)
```

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**

---

routeLib

NAME **routeLib** – network route manipulation library

SYNOPSIS

*routeAdd***( )** – add a route
*routeNetAdd***( )** – add a route to a destination that is a network
*routeDelete***( )** – delete a route

```
STATUS routeAdd
    (char *destination, char *gateway)

STATUS routeNetAdd
    (char *destination, char *gateway)

STATUS routeDelete
    (char *destination, char *gateway)
```

DESCRIPTION

This library contains the routines *routeAdd***( )**, *routeNetAdd***( )**, and *routeDelete***( )** for changing and examining the network routing tables. Routines are provided for adding

and deleting routes that go through a passive gateway. The *routeShow*( ) routine in **netShow** displays the routing tables. VxWorks has no routing daemon; therefore, the tables must be maintained manually.

INCLUDE FILES **routeLib.h**

SEE ALSO **hostLib**, *VxWorks Programmer's Guide: Network*

rpcLib

NAME **rpcLib** – Remote Procedure Call (RPC) support library

SYNOPSIS *rpcInit*( ) – initialize the RPC package
rpcTaskInit( ) – initialize a task's access to the RPC package

```
STATUS rpcInit
    (void)

STATUS rpcTaskInit
    (void)
```

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 by defining **INCLUDE_RPC** in **configAll.h** or **config.h**.

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

SYNOPSIS

*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

```
RT_VOL_DESC *rt11FsDevInit
    (char *devName, BLK_DEV *pBlkDev, BOOL rt11Fmt, int nEntries,
    BOOL changeNoWarn)

STATUS rt11FsInit
    (int maxFiles)

RT_VOL_DESC *rt11FsMkfs
    (char *volName, BLK_DEV *pBlkDev)

void rt11FsDateSet
    (int year, int month, int day)

void rt11FsReadyChange
    (RT_VOL_DESC *vdptr)

void rt11FsModeChange
    (RT_VOL_DESC *vdptr, int newMode)
```

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***( )**.

To enable this initialization, define **INCLUDE_RT11FS** in **configAll.h**.

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 will have no knowledge that disk volumes have been changed, the *changeNoWarn* parameter should be set to **TRUE** when the device is defined with *rt11FsDevInit***( )**. This causes 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 call *ioctl***( )** with **FIODIRENTRY**.

HINTS

The RT-11 file system is much simpler than the 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 more 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 specified directory entries to a **REQ_DIR_ENTRY** structure 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, the structure contains the name, size, and creation date of the file in the first entry (0) of the directory:

```
REQ_DIR_ENTRY req;
req.entryNum = 0;
status = ioctl (fd, FIODIRENTRY, &req);
```

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)

SYNOPSIS

*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)

```
int sched_setparam
    (pid_t tid, const struct sched_param * param)

int sched_getparam
    (pid_t tid, struct sched_param * param)
```

```
int sched_setscheduler
    (pid_t tid, int policy, const struct sched_param * param)

int sched_getscheduler
    (pid_t tid)

int sched_yield
    (void)

int sched_get_priority_max
    (int policy)

int sched_get_priority_min
    (int policy)

int sched_rr_get_interval
    (pid_t tid, struct timespec * interval)
```

DESCRIPTION

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

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

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

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

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

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

INCLUDE FILES **sched.h**

SEE ALSO POSIX 1003.1b document, **taskLib**

scsi1Lib

NAME

scsi1Lib – Small Computer System Interface (SCSI) library (SCSI-1)

SYNOPSIS

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, /* lngth 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
: 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*

scsi2Lib

NAME **scsi2Lib** – Small Computer System Interface (SCSI) library (SCSI-2)

SYNOPSIS
*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
*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

```
void scsi2IfInit()

STATUS scsiTargetOptionsSet
    (SCSI_CTRL *pScsiCtrl, int devBusId, SCSI_OPTIONS *pOptions, UINT which)

STATUS scsiTargetOptionsGet
    (SCSI_CTRL *pScsiCtrl, int devBusId, SCSI_OPTIONS *pOptions)

void scsiPhysDevShow
    (SCSI_PHYS_DEV * pScsiPhysDev, BOOL showThreads, BOOL noHeader)

void scsiCacheSynchronize
    (SCSI_THREAD * pThread, SCSI_CACHE_ACTION action)

int scsiIdentMsgBuild
    (UINT8 * msg, SCSI_PHYS_DEV * pScsiPhysDev, SCSI_TAG_TYPE tagType,
    UINT tagNumber)

SCSI_IDENT_STATUS scsiIdentMsgParse
    (SCSI_CTRL * pScsiCtrl, UINT8 * msg, int msgLength,
    SCSI_PHYS_DEV ** ppScsiPhysDev, SCSI_TAG * pTagNum)

STATUS scsiMsgOutComplete
    (SCSI_CTRL *pScsiCtrl, SCSI_THREAD *pThread)

void scsiMsgOutReject
    (SCSI_CTRL *pScsiCtrl, SCSI_THREAD *pThread)

STATUS scsiMsgInComplete
    (SCSI_CTRL *pScsiCtrl, SCSI_THREAD *pThread)

void scsiSyncXferNegotiate
    (SCSI_CTRL *pScsiCtrl, SCSI_TARGET *pScsiTarget,
    SCSI_SYNC_XFER_EVENT eventType)

void scsiWideXferNegotiate
    (SCSI_CTRL *pScsiCtrl, SCSI_TARGET *pScsiTarget,
    SCSI_WIDE_XFER_EVENT eventType)

STATUS scsiThreadInit
    (SCSI_THREAD * pThread)

void scsiCacheSnoopEnable
    (SCSI_CTRL * pScsiCtrl)

void scsiCacheSnoopDisable
    (SCSI_CTRL * pScsiCtrl)
```

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 *scsi2Transact***( )** 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, /* lngth 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:

reqSenseLength
: If 0, issue a **REQUEST_SENSE** to determine a request sense length.

devType
: This parameter is ignored: an INQUIRY command is used to ascertain the device type. A value of NONE (-1) is the recommended placeholder.

numBlocks
: 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 that are not supported by these libraries (for example, writing software that needs to use an optional SCSI-2 command). Arbitrary commands are handled with the **FIOSCSICOMMAND** option to *scsiIoctl***( )**. The *arg* parameter for **FIOSCSICOMMAND** is a pointer to a valid **SCSI_TRANSACTION** structure. Typically, a call to *scsiIoctl***( )** is written as a subroutine of the form:

```
STATUS myScsiCommand
    (
    SCSI_PHYS_DEV *  pScsiPhysDev,  /* ptr to SCSI physical device     */
    char *           buffer,        /* ptr to data buffer              */
    int              bufLength,     /* length of buffer in bytes       */
    int              someParam      /* param. specifiable in cmd block */
    )
    {
    SCSI_COMMAND myScsiCmdBlock;        /* SCSI command byte array */
    SCSI_TRANSACTION myScsiXaction;     /* info on a SCSI transaction */
    /* fill in fields of SCSI_COMMAND structure */
    myScsiCmdBlock [0] = MY_COMMAND_OPCODE;     /* the required opcode */
```

```
.
myScsiCmdBlock [X] = (UINT8) someParam;     /* for example */
.
myScsiCmdBlock [N-1] = MY_CONTROL_BYTE;     /* typically == 0 */
/* fill in fields of SCSI_TRANSACTION structure */
myScsiXaction.cmdAddress    = myScsiCmdBlock;
myScsiXaction.cmdLength     = <# of valid bytes in myScsiCmdBlock>;
myScsiXaction.dataAddress   = (UINT8 *) buffer;
myScsiXaction.dataDirection = <O_RDONLY (0) or O_WRONLY (1)>;
myScsiXaction.dataLength    = bufLength;
myScsiXaction.addLengthByte = 0;            /* no longer used */
myScsiXaction.cmdTimeout    = <timeout in usec>;
myScsiXaction.tagType       = SCSI_TAG_{DEFAULT,UNTAGGED,
                                        SIMPLE,ORDERED,HEAD_OF_Q};
myScsiXaction.priority      = [ 0 (highest) to 255 (lowest) ];
if (scsiIoctl (pScsiPhysDev, FIOSCSICOMMAND, &myScsiXaction) == OK)
    return (OK);
else
    /* optionally perform retry or other action based on value of
     *  myScsiXaction.statusByte
     */
    return (ERROR);
}
```

INCLUDE FILES **scsiLib.h**, **scsi2Lib.h**

SEE ALSO **dosFsLib**, **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*

scsiCommonLib

NAME **scsiCommonLib** – SCSI library common commands for all devices (SCSI-2)

SYNOPSIS 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)

SYNOPSIS 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)

SYNOPSIS

*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

```
STATUS scsiStartStopUnit
    (SCSI_PHYS_DEV *pScsiPhysDev, BOOL start)

STATUS scsiReserve
    (SCSI_PHYS_DEV *pScsiPhysDev)

STATUS scsiRelease
    (SCSI_PHYS_DEV *pScsiPhysDev)
```

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*

1

scsiLib

NAME

scsiLib – Small Computer System Interface (SCSI) library

SYNOPSIS

*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

```
STATUS scsiPhysDevDelete
    (SCSI_PHYS_DEV *pScsiPhysDev)

SCSI_PHYS_DEV * scsiPhysDevCreate
    (SCSI_CTRL * pScsiCtrl, int devBusId, int devLUN, int reqSenseLength,
    int devType, BOOL removable, int numBlocks, int blockSize)

SCSI_PHYS_DEV * scsiPhysDevIdGet
    (SCSI_CTRL * pScsiCtrl, int devBusId, int devLUN)

STATUS scsiAutoConfig
    (SCSI_CTRL *pScsiCtrl)

STATUS scsiShow
    (SCSI_CTRL *pScsiCtrl)

BLK_DEV * scsiBlkDevCreate
    (SCSI_PHYS_DEV * pScsiPhysDev, int numBlocks, int blockOffset)

void scsiBlkDevInit
    (SCSI_BLK_DEV * pScsiBlkDev, int blksPerTrack, int nHeads)

void scsiBlkDevShow
    (SCSI_PHYS_DEV * pScsiPhysDev)
```

```
STATUS scsiBusReset
    (SCSI_CTRL * pScsiCtrl)

STATUS scsiIoctl
    (SCSI_PHYS_DEV * pScsiPhysDev, int function, int arg)

STATUS scsiFormatUnit
    (SCSI_PHYS_DEV * pScsiPhysDev, BOOL cmpDefectList, int defListFormat,
    int vendorUnique, int interleave, char * buffer, int bufLength)

STATUS scsiModeSelect
    (SCSI_PHYS_DEV * pScsiPhysDev, int pageFormat, int saveParams,
    char * buffer, int bufLength)

STATUS scsiModeSense
    (SCSI_PHYS_DEV * pScsiPhysDev, int pageControl, int pageCode,
    char * buffer, int bufLength)

STATUS scsiReadCapacity
    (SCSI_PHYS_DEV * pScsiPhysDev, int * pLastLBA, int * pBlkLength)

STATUS scsiRdSecs
    (SCSI_BLK_DEV * pScsiBlkDev, int sector, int numSecs, char * buffer)

STATUS scsiWrtSecs
    (SCSI_BLK_DEV * pScsiBlkDev, int sector, int numSecs, char * buffer)

STATUS scsiTestUnitRdy
    (SCSI_PHYS_DEV * pScsiPhysDev)

STATUS scsiInquiry
    (SCSI_PHYS_DEV * pScsiPhysDev, char * buffer, int bufLength)

STATUS scsiReqSense
    (SCSI_PHYS_DEV * pScsiPhysDev, char * buffer, int bufLength)
```

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)

SYNOPSIS

*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

```
STATUS scsiMgrEventNotify
    (SCSI_CTRL * pScsiCtrl, SCSI_EVENT * pEvent, int eventSize)

void scsiMgrBusReset
    (SCSI_CTRL * pScsiCtrl)

void scsiMgrCtrlEvent
    (SCSI_CTRL * pScsiCtrl, SCSI_EVENT_TYPE eventType)

void scsiMgrThreadEvent
    (SCSI_THREAD * pThread, SCSI_THREAD_EVENT_TYPE eventType)

void scsiMgrShow
    (SCSI_CTRL * pScsiCtrl, BOOL showPhysDevs, BOOL showThreads,
    BOOL showFreeThreads)
```

DESCRIPTION

This SCSI-2 library implements the SCSI manager. The purpose of the SCSI manager is to manage SCSI threads between requesting VxWorks tasks and the SCSI controller. The SCSI manager handles SCSI events and SCSI threads but allocation and de-allocation of SCSI threads is not the manager's responsiblity. 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 VxWorks task upon initialization of the SCSI interface within VxWorks. The entry point of the SCSI manager task is *scsiMgr***( )**. The SCSI manager task is usually spawned during initialization of the SCSI controller driver. The driver's *xxxCtrlCreateScsi2***( )** routine is typically responsible for such SCSI interface initializations.

Once the SCSI manager has been initialized, it is ready to handle SCSI requests from VxWorks tasks. The SCSI manager has the following resposibilities:

– 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 intialization 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)

SYNOPSIS
*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 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

```
SEQ_DEV *scsiSeqDevCreate
    (SCSI_PHYS_DEV *pScsiPhysDev)

STATUS scsiErase
    (SCSI_PHYS_DEV *pScsiPhysDev, BOOL longErase)
```

```
STATUS scsiTapeModeSelect
    (SCSI_PHYS_DEV *pScsiPhysDev, int pageFormat, int saveParams,
    char *buffer, int bufLength)

STATUS scsiTapeModeSense
    (SCSI_PHYS_DEV *pScsiPhysDev, int pageControl, int pageCode,
    char *buffer, int bufLength)

STATUS scsiSeqReadBlockLimits
    (SCSI_SEQ_DEV * pScsiSeqDev, int *pMaxBlockLength,
    UINT16 *pMinBlockLength)

STATUS scsiRdTape
    (SCSI_SEQ_DEV *pScsiSeqDev, int numBytes, char *buffer, BOOL fixedSize)

STATUS scsiWrtTape
    (SCSI_SEQ_DEV *pScsiSeqDev, int numBytes, char *buffer, BOOL fixedSize)

STATUS scsiRewind
    (SCSI_SEQ_DEV *pScsiSeqDev)

STATUS scsiReserveUnit
    (SCSI_SEQ_DEV *pScsiSeqDev)

STATUS scsiReleaseUnit
    (SCSI_SEQ_DEV *pScsiSeqDev)

STATUS scsiLoadUnit
    (SCSI_SEQ_DEV * pScsiSeqDev, BOOL load, BOOL reten, BOOL eot)

STATUS scsiWrtFileMarks
    (SCSI_SEQ_DEV * pScsiSeqDev, int numMarks, BOOL shortMark)

STATUS scsiSpace
    (SCSI_SEQ_DEV * pScsiSeqDev, int count, int spaceCode)

STATUS scsiSeqStatusCheck
    (SCSI_SEQ_DEV *pScsiSeqDev)

int scsiSeqIoctl
    (SCSI_SEQ_DEV * pScsiSeqDev, int function, int arg)
```

DESCRIPTION

This library contains commands common to all sequential-access SCSI devices. Sequential-access SCSI devices are usually SCSI tape devices. These routines are separated from **scsi2Lib** in order to create an additional layer for better support of all SCSI sequential devices.

SCSI commands in this library include:

Command	Op Code
ERASE	(0x19)
MODE SELECT (6)	(0x15)

Command	Op Code
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

SYNOPSIS

*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
*selWakeupListLen***( )** – get the number of nodes in a *select***( )** wake-up list
*selWakeupType***( )** – get the type of a *select***( )** wake-up node

```
void selectInit
    (void)
```

```
int select
    (int width, fd_set *pReadFds, fd_set *pWriteFds, fd_set *pExceptFds,
    struct timeval *pTimeOut)

void selWakeup
    (SEL_WAKEUP_NODE *pWakeupNode)

void selWakeupAll
    (SEL_WAKEUP_LIST *pWakeupList, SELECT_TYPE type)

STATUS selNodeAdd
    (SEL_WAKEUP_LIST *pWakeupList, SEL_WAKEUP_NODE *pWakeupNode)

STATUS selNodeDelete
    (SEL_WAKEUP_LIST *pWakeupList, SEL_WAKEUP_NODE *pWakeupNode)

void selWakeupListInit
    (SEL_WAKEUP_LIST *pWakeupList)

int selWakeupListLen
    (SEL_WAKEUP_LIST *pWakeupList)

SELECT_TYPE selWakeupType
    (SEL_WAKEUP_NODE *pWakeupNode)
```

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. The maximum number of file descriptors supported by **selectLib** is 256.

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.

INCLUDE FILES **selectLib.h**

SEE ALSO *VxWorks Programmer's Guide: I/O System*

semBLib

NAME **semBLib** – binary semaphore library

SYNOPSIS *semBCreate***( )** – create and initialize a binary semaphore

```
SEM_ID semBCreate
    (int options, SEM_B_STATE initialState)
```

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 preempts 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 are not given back, 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

SYNOPSIS *semCCreate***( )** – create and initialize a counting semaphore

```
SEM_ID semCCreate
    (int options, int initialCount)
```

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 are not given back, 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*

semLib

NAME **semLib** – general semaphore library

SYNOPSIS *semGive***( )** – give a semaphore
*semTake***( )** – take a semaphore
*semFlush***( )** – unblock every task pended on a semaphore
*semDelete***( )** – delete a semaphore

```
STATUS semGive
    (SEM_ID semId)

STATUS semTake
    (SEM_ID semId, int timeout)

STATUS semFlush
    (SEM_ID semId)

STATUS semDelete
    (SEM_ID semId)
```

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.

INCLUDE FILES

semLib.h

SEE ALSO

taskLib, **semBLib**, **semCLib**, **semMLib**, **semSmLib**, *VxWorks Programmer's Guide: Basic OS*

semMLib

NAME

semMLib – mutual-exclusion semaphore library

SYNOPSIS

*semMCreate***( )** – create and initialize a mutual-exclusion semaphore
*semMGiveForce***( )** – give a mutual-exclusion semaphore without restrictions

```
SEM_ID semMCreate
    (int options)

STATUS semMGiveForce
    (SEM_ID semId)
```

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 *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:

– Avoid overlapping critical regions.

– Adjust priorities of tasks so that there are no tasks at intermediate priority levels.

– Adjust priorities of tasks so that priority inheritance protocol is not needed.

– 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

SYNOPSIS *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

```
SEM_ID semCreate
    (void)

STATUS semInit
    (SEMAPHORE *pSemaphore)

STATUS semClear
    (SEM_ID semId)
```

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 reference entry for **semBLib**.

INCLUDE FILES **semLib.h**

SEE ALSO **semLib**, **semBLib**, *VxWorks Programmer's Guide: Basic OS*

semPxLib

NAME **semPxLib** – semaphore synchronization library (POSIX)

SYNOPSIS *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)

```
STATUS semPxLibInit
    (void)

int sem_init
    (sem_t * sem, int pshared, unsigned int value)

int sem_destroy
    (sem_t * sem)

sem_t * sem_open
    (const char * name, int oflag, ...)

int sem_close
    (sem_t * sem)

int sem_unlink
    (const char * name)

int sem_wait
    (sem_t * sem)

int sem_trywait
    (sem_t * sem)

int sem_post
    (sem_t * sem)

int sem_getvalue
    (sem_t * sem, int * sval)
```

DESCRIPTION

This library implements the POSIX 1003.1b semaphore interface. For alternative semaphore routines designed expressly for VxWorks, see the manual page for **semLib** and other semaphore libraries mentioned there. POSIX semaphores are counting semaphores; as such they are most similar to the **semCLib** VxWorks-specific semaphores.

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

SYNOPSIS *semPxShowInit***( )** – initialize the POSIX semaphore show facility

```
STATUS semPxShowInit
    (void)
```

DESCRIPTION This library provides a show routine for POSIX semaphore objects.

semShow

NAME **semShow** – semaphore show routines

SYNOPSIS *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

```
void semShowInit
    (void)

int semInfo
    (SEM_ID semId, int idList[], int maxTasks)

STATUS semShow
    (SEM_ID semId, int level)
```

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 **INCLUDE_SHOW_ROUTINES** is defined in **configAll.h**.

INCLUDE FILES **semLib.h**

SEE ALSO **semLib**, *VxWorks Programmer's Guide: Basic OS*

semSmLib

NAME **semSmLib** – shared memory semaphore library (VxMP Opt.)

SYNOPSIS *semBSmCreate***( )** – create and initialize a shared memory binary semaphore
*semCSmCreate***( )** – create and initialize a shared memory counting semaphore

```
SEM_ID semBSmCreate
    (int options, SEM_B_STATE initialState)

SEM_ID semCSmCreate
    (int options, int initialCount)
```

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 reference 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 reference entry for **semCLib**.

For information about the generic semaphore-handling routines, see the reference 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, defined by **SM_OBJ_MAX_SEM** in **configAll.h**.

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 the interrupt latency.

CONFIGURATION

Before routines in this library can be called, the shared memory object facility must be initialized by calling *usrSmObjInit***( )**, which is found in **src/config/usrSmObj.c**. This is done automatically from the root task, *usrRoot***( )**, in **usrConfig.c** if **INCLUDE_SM_OBJ** is defined in **configAll.h**.

AVAILABILITY

This module is provided with the unbundled shared memory objects support option, VxMP.

INCLUDE FILES

semSmLib.h

SEE ALSO

semLib, **semBLib**, **semCLib**, **smObjLib**, **semShow**, *usrSmObjInit***( )**, *VxWorks Programmer's Guide: Shared Memory Objects*, *Basic OS*

shellLib

NAME

shellLib – shell execution routines

SYNOPSIS

*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

```
STATUS shellInit
    (int stackSize, int arg)

void shell
    (BOOL interactive)

void shellScriptAbort
    (void)

void shellHistory
    (int size)

void shellPromptSet
    (char *newPrompt)

void shellOrigStdSet
    (int which, int fd)

BOOL shellLock
    (BOOL request)
```

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

SYNOPSIS

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
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

```
int sigInit
    (void)

int sigqueueInit
    (int nQueues)

int sigemptyset
    (sigset_t *pSet)

int sigfillset
    (sigset_t *pSet)

int sigaddset
    (sigset_t *pSet, int signo)

int sigdelset
    (sigset_t *pSet, int signo)

int sigismember
    (const sigset_t *pSet, int signo)
```

```
void
    (*signal (int signo, void (*pHandler) () ))()

int sigaction
    (int signo, const struct sigaction *pAct, struct sigaction *pOact)

int sigprocmask
    (int how, const sigset_t *pSet, sigset_t *pOset)

int sigpending
    (sigset_t *pSet)

int sigsuspend
    (const sigset_t *pSet)

int pause
    (void)

int sigtimedwait
    (const sigset_t *pSet, struct siginfo *pInfo,
    const struct timespec *pTimeout)

int sigwaitinfo
    (const sigset_t *pSet, struct siginfo *pInfo)

int sigvec
    (int sig, const struct sigvec *pVec, struct sigvec *pOvec)

int sigsetmask
    (int mask)

int sigblock
    (int mask)

int raise
    (int signo)

int kill
    (int tid, int signo)

int sigqueue
    (int tid, int signo, const union sigval value)
```

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*( )
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.

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.

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"

SPARC

Signal	Code	Exception
SIGBUS	**BUS_INSTR_ACCESS**	bus error on instruction fetch
SIGBUS	**BUS_ALIGN**	address error (bad alignment)
SIGBUS	**BUS_DATA_ACCESS**	bus error on data access
SIGILL	**ILL_ILLINSTR_FAULT**	illegal instruction

Signal	Code	Exception
SIGILL	**ILL_PRIVINSTR_FAULT**	privilege violation
SIGILL	**ILL_COPROC_DISABLED**	coprocessor disabled
SIGILL	**ILL_COPROC_EXCPTN**	coprocessor exception
SIGILL	**ILL_TRAP_FAULT(n)**	uninitialized user trap
SIGFPE	**FPE_FPA_ENABLE**	floating point disabled
SIGFPE	**FPE_FPA_ERROR**	floating point exception
SIGFPE	**FPE_INTDIV_TRAP**	zero divide
SIGEMT	**EMT_TAG**	tag overflow

Intel i960

Signal	Code	Exception
SIGBUS	**BUS_UNALIGNED**	address error (bad alignment)
SIGBUS	**BUS_BUSERR**	bus error
SIGILL	**ILL_INVALID_OPCODE**	invalid instruction
SIGILL	**ILL_UNIMPLEMENTED**	instr fetched from on-chip RAM
SIGILL	**ILL_INVALID_OPERAND**	invalid operand
SIGILL	**ILL_CONSTRAINT_RANGE**	constraint range failure
SIGILL	**ILL_PRIVILEGED**	privilege violation
SIGILL	**ILL_LENGTH**	bad index to sys procedure table
SIGILL	**ILL_TYPE_MISMATCH**	privilege violation
SIGTRAP	**TRAP_INSTRUCTION_TRACE**	instruction trace fault
SIGTRAP	**TRAP_BRANCH_TRACE**	branch trace fault
SIGTRAP	**TRAP_CALL_TRACE**	call trace fault
SIGTRAP	**TRAP_RETURN_TRACE**	return trace fault
SIGTRAP	**TRAP_PRERETURN_TRACE**	pre-return trace fault
SIGTRAP	**TRAP_SUPERVISOR_TRACE**	supervisor trace fault
SIGTRAP	**TRAP_BREAKPOINT_TRACE**	breakpoint trace fault
SIGFPE	**FPE_INTEGER_OVERFLOW**	integer overflow
SIGFPE	**FST_ZERO_DIVIDE**	integer zero divide
SIGFPE	**FPE_FLOATING_OVERFLOW**	floating point overflow
SIGFPE	**FPE_FLOATING_UNDERFLOW**	floating point underflow
SIGFPE	**FPE_FLOATING_INVALID_OPERATION**	invalid floating point operation
SIGFPE	**FPE_FLOATING_ZERO_DIVIDE**	floating point zero divide
SIGFPE	**FPE_FLOATING_INEXACT**	floating point inexact
SIGFPE	**FPE_FLOATING_RESERVED_ENCODING**	floating point reserved encoding

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
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 check
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**	instruction breakpoint (PPC603/4)
SIGTRAP	**_EXC_OFF_TRACE**	trace (PPC603/4, PPC860)
SIGBUS	**_EXC_OFF_CRTL**	critical interrupt (PPC403)
SIGILL	**_EXC_OFF_SYSCALL**	system call

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.)

SYNOPSIS

*memPartSmCreate***( )** – create a shared memory partition
*smMemAddToPool***( )** – add memory to the shared memory system partition
*smMemOptionsSet***( )** – set the debug options for the shared memory system partition
*smMemMalloc***( )** – allocate a block of memory from the shared memory system partition
*smMemCalloc***( )** – allocate memory for an array from the shared memory system partition
*smMemRealloc***( )** – reallocate a block of memory from the shared memory system partition
*smMemFree***( )** – free a shared memory system partition block of memory
*smMemFindMax***( )** – find the largest free block in the shared memory system partition

```
PART_ID memPartSmCreate
    (char * pPool, unsigned poolSize)

STATUS smMemAddToPool
    (char * pPool, unsigned poolSize)

STATUS smMemOptionsSet
    (unsigned options)
```

```
void * smMemMalloc
    (unsigned nBytes)

void * smMemCalloc
    (int elemNum, int elemSize)

void * smMemRealloc
    (void * pBlock, unsigned newSize)

STATUS smMemFree
    (void * ptr)

int smMemFindMax
    (void)
```

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 **SM_OBJ_MAX_MEM_PART**, defined in **configAll.h**.

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, 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 *usrSmObjInit***( )** in **src/config/usrSmObj.c**. This is done automatically from the root task, *usrRoot***( )** in **usrConfig.c** if **INCLUDE_SM_OBJ** is defined in **configAll.h**.

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 can be specified 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 option when a shared partition is created is **MEM_ALLOC_ERROR_LOG_MSG**.

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.)

SYNOPSIS *smMemShow***( )** – show the shared memory system partition blocks and statistics

```
void smMemShow
    (int type)
```

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 **INCLUDE_SM_OBJ** is defined in **configAll.h**.

AVAILABILITY This module is provided with 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.)

SYNOPSIS *smNameAdd***( )** – add a name to the shared memory name database
*smNameFind***( )** – look up a shared memory object by name
*smNameFindByValue***( )** – look up a shared memory object by value
*smNameRemove***( )** – remove an object from the shared memory objects name database

```
STATUS smNameAdd
    (char * name, void * value, int type)

STATUS smNameFind
    (char * name, void ** pValue, int * pType, int waitType)

STATUS smNameFindByValue
    (void * value, char * name, int * pType, int waitType)

STATUS smNameRemove
    (char * name)
```

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 **SM_OBJ_MAX_NAME** is defined in **configAll.h**. 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 **smNameShow**.

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 "stdio.h"
testSmSem1 (void)
    {
    SEM_ID smSemId;
    /* create a shared semaphore */
    if ((smSemId = semBSmCreate(SEM_EMPTY)) == NULL)
        {
        printf ("Shared semaphore creation error.");
        return (ERROR);
        }
    /*
     * make created semaphore Id available to all CPUs in
     * the system by entering its name in shared name database.
     */
```

```
    if (smNameAdd ("smSem", smSemId, T_SM_SEM_B) != OK )
        {
        printf ("Cannot add smSem into shared database.");
        return (ERROR);
        }
        ...
    /* now use the semaphore */
    semGive (smSemId);
        ...
    }
```

On CPU 2 :

```
#include "vxWorks.h"
#include "semLib.h"
#include "smNameLib.h"
#include "stdio.h"
testSmSem2 (void)
    {
    SEM_ID smSemId;
    /* get semaphore ID from name database */
    smNameFind ("smSem", &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***( )**, which is found in **src/config/usrSmObj.c**. This is done automatically from the root task, *usrRoot***( )**, in **usrConfig.c** if **INCLUDE_SM_OBJ** is defined in **configAll.h**.

AVAILABILITY This module is provided with 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.)

SYNOPSIS *smNameShow***( )** – show the contents of the shared memory objects name database

```
STATUS smNameShow
    (int level)
```

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 **smNameLib**.

CONFIGURATION The routines in this library are included by default if **INCLUDE_SM_OBJ** is defined in **configAll.h**.

AVAILABILITY This module is provided with 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 the shared memory network (backplane) driver

SYNOPSIS *smNetInit***( )** – initialize the shared memory network driver
*smNetAttach***( )** – attach the shared memory network interface
*smNetInetGet***( )** – get an address associated with a shared memory network interface

```
STATUS smNetInit
    (SM_ANCHOR * pAnchor, char * pMem, int memSize, BOOL tasType,
    int cpuMax, int maxPktBytes, u_long startAddr)

STATUS smNetAttach
    (int unit, SM_ANCHOR * pAnchor, int maxInputPkts, int intType,
    int intArg1, int intArg2, int intArg3)

STATUS smNetInetGet
    (char * smName, char * smInet, int cpuNum)
```

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 three user-callable routines: *smNetInit*( ), *smNetAttach*( ), and *smNetInetGet*( ).

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 in **usrConfig.c** when backplane parameters are specified in the boot line.

The *smNetInetGet*( ) routine gets the Internet address associated with a backplane interface.

INCLUDE FILES **smPktLib.h**, **smUtilLib.h**

SEE ALSO **ifLib**, **if_sm**, *VxWorks Programmer's Guide: Network*

smNetShow

NAME **smNetShow** – shared memory network driver show routines

SYNOPSIS *smNetShow*( ) – show information about a shared memory network

```
STATUS smNetShow
    (char * ifName, BOOL zero)
```

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 **smPktLib.h**

SEE ALSO *VxWorks Programmer's Guide: Network*

smObjLib

NAME

smObjLib – shared memory objects library (VxMP Opt.)

SYNOPSIS

*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***( )** – enable / disable logging of failed attempts to take a spin-lock

```
STATUS smObjLibInit
    (void)

STATUS smObjSetup
    (SM_OBJ_PARAMS * smObjParams)

void smObjInit
    (SM_OBJ_DESC * pSmObjDesc, SM_ANCHOR * anchorLocalAdrs,
    int ticksPerBeat, int smObjMaxTries, int intType, int intArg1,
    int intArg2, int intArg3)

STATUS smObjAttach
    (SM_OBJ_DESC * pSmObjDesc)

void * smObjLocalToGlobal
    (void * localAdrs)

void * smObjGlobalToLocal
    (void * globalAdrs)

void smObjTimeoutLogEnable
    (BOOL timeoutLogEnable)
```

DESCRIPTION

This library contains miscellaneous functions used by the shared memory objects facility. Shared memory objects provide high-speed synchronization and communication among tasks running on separate CPUs that have access to 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 interprocessor synchronization, mutual exclusion on multiprocessor shared data structures, and high-speed data exchange.

Routines for displaying shared memory objects statistics are provided by **smObjShow**.

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 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) which acts as a low-level mutual exclusion device. The spin-

lock mechanism is called with a system-wide parameter, **SM_OBJ_MAX_TRIES**, defined in **configAll.h**, which specifies the maximum number of RMW tries on a spin-lock location.

This parameter is set to 100 by default, but must be set to a higher value as the number of CPUs increases or when high-speed processors are used. 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 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 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 **INCLUDE_SM_OBJ** is defined in **configAll.h**, the init and setup routines in this library are called automatically by *usrSmObjInit***( )** from the root task, *usrRoot***( )**, in **usrConfig.c**.

AVAILABILITY

This module is provided with 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.)

SYNOPSIS *smObjShow*( ) – display the current status of shared memory objects

```
STATUS smObjShow()
```

DESCRIPTION This library provides routines to show shared memory object statistics, such as the current number of shared tasks, semaphores, message queues, etc.

CONFIGURATION These routines are included by default if **INCLUDE_SM_OBJ** is defined in **configAll.h**.

AVAILABILITY This module is provided with the unbundled shared memory objects support option, VxMP.

INCLUDE FILES **smObjLib.h**

SEE ALSO **smObjLib**, *VxWorks Programmer's Guide: Shared Memory Objects*

snmpAuxLib

NAME **snmpAuxLib** – utility routines for object identifiers

SYNOPSIS *ip_to_rlist*( ) – convert an IP address to an array of OID components
oidcmp( ) – compare two object identifiers
oidcmp2( ) – compare two object identifiers
oid_to_ip( ) – convert an object identifier to an IP address

```
int ip_to_rlist
    (UINT_32_T ip_address, OIDC_T * object_id);

int oidcmp
    (int length_1, OIDC_T * oid_1, int length_2, OIDC_T * oid_2);

int oidcmp2
    (int length_1, OIDC_T * oid_1, int length_2, OIDC_T * oid_2);

int oid_to_ip
    (int count, OIDC_T * object_id, UINT_32_T * addr);
```

DESCRIPTION This module defines the routines used to manipilate object identifiers.

INCLUDE FILES **snmpdefs.h**

snmpBindLib

NAME

snmpBindLib – routines for binding values to variables in SNMP packets

SYNOPSIS

*SNMP_Bind_Unsigned_Integer***( )** – bind an unsigned-integer variable
*SNMP_Bind_Integer***( )** – bind an integer variable
*SNMP_Bind_IP_Address***( )** – bind an IP address variable
*SNMP_Bind_Object_ID***( )** – bind an object-identifier variable
*SNMP_Bind_String***( )** – bind a string variable
*SNMP_Bind_64_Unsigned_Integer***( )** – bind a 64-bit unsigned-integer variable
*SNMP_Bind_Null***( )** – bind a null-valued variable

```
int SNMP_Bind_Unsigned_Integer
     (SNMP_PKT_T * pktp, int index, int compc, OIDC_T * compl,
     OCTET_T typeFlags, UINT_32_T value);

int SNMP_Bind_Integer
     (SNMP_PKT_T * pktp, int index, int compc, OIDC_T * compl,
     INT_32_T value);

int SNMP_Bind_IP_Address
     (SNMP_PKT_T * pktp, int index, int compc, OIDC_T * compl,
     OCTET_T * pIpAddr);

int SNMP_Bind_Object_ID
     (SNMP_PKT_T * pktp, int index, int compc, OIDC_T * compl, int valc,
     OIDC_T * vall);

int SNMP_Bind_String
     (SNMP_PKT_T * pktp, int index, int compc, OIDC_T * compl,
     OCTET_T typeFlags, int leng, OCTET_T * strp, int statflg);

int SNMP_Bind_64_Unsigned_Integer
     (SNMP_PKT_T * pktp, int index, int compc, OIDC_T * compl,
     OCTET_T typeFlags, UINT_32_T high, UINT_32_T low);

int SNMP_Bind_Null
     (SNMP_PKT_T * pktp, int index, int compc, OIDC_T * compl);
```

DESCRIPTION

This module defines the routines used to bind variables to their respective values in an SNMP packet.

INCLUDE FILES

snmpdefs.h

snmpdLib

NAME

snmpdLib – entry points to the SNMP v1/v2c agent

SYNOPSIS

snmpdPktProcess( ) – process a packet returned by the transport
snmpdLog( ) – log messgaes from the SNMP agent
snmpdViewEntrySet( ) – install an entry in the view table
snmpdViewEntryRemove( ) – remove an entry from the view table
snmpdTreeAdd( ) – dynamically add a subtree to the SNMP agent MIB tree
snmpdTreeRemove( ) – dynamically remove part of the SNMP agent MIB tree
snmpdTrapSend( ) – general interface to trap facilities
snmpdInitFinish( ) – complete the initialization of the agent
snmpdExit( ) – exit the SNMP agent
snmpdContinue( ) – continue processing of an SNMP packet
snmpdGroupByGetprocAndInstance( ) – gather set of similar variable bindings
snmpdVbRowExtract( ) – extract required pieces of a row for a **set** operation
snmpdVbExtractRowLoose( ) – incrementally extract pieces of a row for a set
snmpdPktLockGet( ) – lock an SNMP packet

```
void snmpdPktProcess
    (int pktSize, char * pBuf, void * pRemoteAddr, void * pLocalAddr,
    void * pSnmpEndpoint)

void snmpdLog
    (int level, char * string)

STATUS snmpdViewEntrySet
    (OIDC_T * pTreeOid, int treeOidLen, UINT_16_T index, uchar_t * pMask,
    int maskLen, int viewType)

void snmpdViewEntryRemove
    (OIDC_T * pTreeOid, int treeOidLen, UINT_16_T index)

STATUS snmpdTreeAdd
    (char * pTreeOidStr, MIBNODE_T * pTreeAddr)

void snmpdTreeRemove
    (char * pTreeOidStr)

void snmpdTrapSend
    (void * pSnmpEndpoint, int numDestn, void ** ppDestAddrTbl,
    void * pLocalAddr, int version, char * pTrapCmnty, OIDC_T * pMyOid,
    int myOidLen, u_long * pIpAddr, int trapType, int trapSpecific,
    int numVarBinds, FUNCPTR trapVarBindsRtn, void * pCookie)

void snmpdInitFinish
    (VOIDFUNCPTR pPrivRlse, FUNCPTR pSetPduVldt, FUNCPTR pPreSet,
    FUNCPTR pPostSet, FUNCPTR pSetFailed)
```

```
void snmpdExit
     (void)

void snmpdContinue
     (SNMP_PKT_T * pktp);

void snmpdGroupByGetprocAndInstance
     (SNMP_PKT_T * pktp, VB_T * firstVbp, int compc, OIDC_T * compl);

VB_T * snmpdVbRowExtract
     (SNMP_PKT_T * pktp, int start_index, int compc, OIDC_T * compl,
     int row_structure_length, struct create_row * row);

VB_T * snmpdVbExtractRowLoose
     (SNMP_PKT_T * pktp, int indx, MIBLEAF_T ** leaves, int compc,
     OIDC_T * compl);

STATUS snmpdPktLockGet
     (SNMP_PKT_T * pktp)
```

DESCRIPTION

This module implements the WindNet SNMPv1/v2c agent for VxWorks. This agent provides facilities for managing objects as defined by the MIB-II standard. The agent management information base can be extended to include additional user-defined MIBs. The agent also supports asynchronous method routines and dynamic loading of MIBs.

INCLUDE FILES

snmpdLib.h

SEE ALSO

The SNMP version 1 framework is defined by the following Request For Comments (RFCs): 1155, 1157, 1212. MIB-II is defined by RFC 1213. For more information about SNMP, refer to these documents. For more information about the VxWorks SNMP agent, see the *WindNet SNMP VxWorks Component Release Supplement.*

snmpEbufLib

NAME

snmpEbufLib – extended-buffer manipulation functions

SYNOPSIS

*EBufferClone***( )** – make a copy of an extended buffer
*EBufferClean***( )** – release dynamic memory in an extended buffer
*EBufferInitialize***( )** – place an extended buffer in a known state
*EBufferSetup***( )** – attach an empty memory buffer to an extended buffer
*EBufferPreLoad***( )** – attach a full memory buffer to an extended buffer
*EBufferNext***( )** – return a pointer to the next unused byte of the buffer memory
*EBufferStart***( )** – return a pointer to the first byte in the buffer memory
*EBufferUsed***( )** – return the number of used bytes in the buffer memory

EBufferReset( ) – reset the extended buffer
EBufferRemaining( ) – return the number of unused bytes remaining in buffer memory

```
int EBufferClone
    (EBUFFER_T * srcp, EBUFFER_T * dstp);

void EBufferClean
    (EBUFFER_T * ebuffp);

void EBufferInitialize
    (EBUFFER_T * ebuffp);

void EBufferSetup
    (unsigned int flags, EBUFFER_T * ebuffp, OCTET_T * datap,
    ALENGTH_T datal);

void EBufferPreLoad
    (unsigned int flags, EBUFFER_T * ebuffp, OCTET_T * datap,
    ALENGTH_T datal);

OCTET_T * EBufferNext
    (EBUFFER_T * ebuffp);

OCTET_T * EBufferStart
    (EBUFFER_T * ebuffp);

ALENGTH_T EBufferUsed
    (EBUFFER_T * ebuffp);

void EBufferReset
    (EBUFFER_T * ebuffp);

void EBufferRemaining
    (EBUFFER_T * ebuffp);
```

DESCRIPTION This module defines the routines used to manipulate extended buffers.

INCLUDE FILES **buffer.h**

snmpIoLib

NAME **snmpIoLib** – default transport routines for SNMP

SYNOPSIS *snmpIoInit*( ) – initialization routine for SNMP transport endpoint
snmpIoWrite( ) – write a packet to the transport
snmpIoClose( ) – close the transport endpoint
snmpIoMain( ) – main SNMP IO routine

snmpIoTrapSend( ) – send a standard SNMP or MIB-II trap
snmpIoCommunityValidate( ) – sample community validation routine
snmpdMemoryAlloc( ) – allocate memory for the SNMP agent
snmpdMemoryFree( ) – free memory allocated by the SNMP agent

```
STATUS snmpIoInit
    ()

void snmpIoWrite
    (void * pSocket, char * pBuf, int bufSize, void * remote, void * local)

void snmpIoClose
    (void)

void snmpIoMain
    ()

void snmpIoTrapSend
    (int trapType, int trapSpecific)

int snmpIoCommunityValidate
    (SNMP_PKT_T * pPkt, SNMPADDR_T * pRemoteAddr, SNMPADDR_T * pLocalAddr)

void * snmpdMemoryAlloc
    (size_t size)

void snmpdMemoryFree
    (void * pBuf)
```

DESCRIPTION This module implements the SNMP v1 / v2c transport transport-dependent routines.

INCLUDE FILES **snmpdIoLib.h**

snmpProcLib

NAME **snmpProcLib** – manipulate variable-bindings in an SNMP packet

SYNOPSIS
getproc_started( ) – indicate that a **getproc** operation has begun
getproc_good( ) – indicate successful completion of a **getproc** procedure
getproc_error( ) – indicate that a **getproc** operation encountered an error
nextproc_started( ) – indicate that a **nextproc** operation has begun
nextproc_good( ) – indicate successful completion of a **nextproc** procedure
nextproc_no_next( ) – indicate that there exists no next instance
nextproc_error( ) – indicate that a **nextproc** operation encountered an error
getproc_got_int32( ) – indicate retrieval of a 32-bit integer
getproc_got_uint32( ) – indicate retrieval of a 32-bit unsigned integer

getproc_got_ip_address( ) – indicate retrieval of an IP address
getproc_got_empty( ) – indicate retrieval of a null value
getproc_got_string( ) – indicate retrieval of a string
testproc_started( ) – indicate that a **testproc** operation has begun
testproc_good( ) – indicate successful completion of a **testproc** procedure
testproc_error( ) – indicate that a **testproc** operation encountered an error
setproc_started( ) – indicate that a **setproc** operation has begun
setproc_good( ) – indicates successful completion of a **setproc** procedure
setproc_error( ) – indicate that a **setproc** operation encountered an error
undoproc_started( ) – indicate that an **undoproc** operation has begun
undoproc_good( ) – indicates successful completion of an **undoproc** operation
undoproc_error( ) – indicate that an **undproc** operation encountered an error
getproc_got_uint64( ) – indicate retrieval of a 64-bit unsigned integer
getproc_got_uint64_high_low( ) – indicate retrieval of a 64-bit unsigned integer with high and low halves
getproc_nosuchins( ) – indicates that no such instance exists
getproc_got_object_id( ) – indicate retrieval of an object identifier
nextproc_next_instance( ) – install instance part of next instance

```
void getproc_started
     (SNMP_PKT_T * pPkt, VB_T * pVarBind)

void getproc_good
     (SNMP_PKT_T * pPkt, VB_T * pVarBind)

void getproc_error
     (SNMP_PKT_T * pPkt, VB_T * pVarBind, INT_32_T error)

void nextproc_started
     (SNMP_PKT_T * pPkt, VB_T * pVarBind)

void nextproc_good
     (SNMP_PKT_T * pPkt, VB_T * pVarBind)

void nextproc_no_next
     (SNMP_PKT_T * pPkt, VB_T * pVarBind)

void nextproc_error
     (SNMP_PKT_T * pPkt, VB_T * pVarBind, INT_32_T error)

void getproc_got_int32
     (SNMP_PKT_T * pPkt, VB_T * pVarBind, INT_32_T data)

void getproc_got_uint32
     (SNMP_PKT_T * pPkt, VB_T * pVarBind, UINT_32_T data, OCTET_T type)

void getproc_got_ip_address
     (SNMP_PKT_T * pPkt, VB_T * pVarBind, UINT_32_T addrData)

void getproc_got_empty
     (SNMP_PKT_T * pPkt, VB_T * pVarBind)
```

```
void getproc_got_string
     (SNMP_PKT_T * pPkt, VB_T * pVarBind, ALENGTH_T size, OCTET_T * data,
     int dynamicFlg, OCTET_T type)

void testproc_started
     (SNMP_PKT_T * pPkt, VB_T * pVarBind)

void testproc_good
     (SNMP_PKT_T * pPkt, VB_T * pVarBind)

void testproc_error
     (SNMP_PKT_T * pPkt, VB_T * pVarBind, INT_32_T error)

void setproc_started
     (SNMP_PKT_T * pPkt, VB_T * pVarBind)

void setproc_good
     (SNMP_PKT_T * pPkt, VB_T * pVarBind)

void setproc_error
     (SNMP_PKT_T * pPkt, VB_T * pVarBind, INT_32_T error)

void undoproc_started
     (SNMP_PKT_T * pPkt, VB_T * pVarBind)

void undoproc_good
     (SNMP_PKT_T * pPkt, VB_T * pVarBind)

void undoproc_error
     (SNMP_PKT_T * pPkt, VB_T * pVarBind, INT_32_T error)

void getproc_got_uint64
     (SNMP_PKT_T * pPkt, VB_T * pVarBind, UINT_64_T * data)

void getproc_got_uint64_high_low
     (SNMP_PKT_T * pPkt, VB_T * pVarBind, UINT_32_T high, UINT_32_T low)

void getproc_nosuchins
     (SNMP_PKT_T * pPkt, VB_T * pVarBind);

void getproc_got_object_id
     (SNMP_PKT_T * pPkt, VB_T * pVarBind, int length, OIDC_T * pOid,
     int flag);

void nextproc_next_instance
     (SNMP_PKT_T * pPkt, VB_T * pVarBind, int length, OIDC_T * pOid);
```

DESCRIPTION This module defines routines used to manipulate variable bindings in an SNMP packet. These are equivalents for macros defined in **snmpdefs.h**.

INCLUDE FILES **snmpdefs.h**

sockLib

NAME

sockLib – generic socket library

SYNOPSIS

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 a connection over a socket for 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

```
int socket
    (int domain, int type, int protocol)

STATUS bind
    (int s, struct sockaddr *name, int namelen)

STATUS listen
    (int s, int backlog)

int accept
    (int s, struct sockaddr *addr, int *addrlen)

STATUS connect
    (int s, struct sockaddr *name, int namelen)

STATUS connectWithTimeout
    (int sock, struct sockaddr *adrs, int adrsLen, struct timeval *timeVal)

int sendto
    (int s, caddr_t buf, int bufLen, int flags, struct sockaddr *to,
    int tolen)

int send
    (int s, char *buf, int bufLen, int flags)
```

```
int sendmsg
    (int sd, struct msghdr *mp, int flags)

int recvfrom
    (int s, char *buf, int bufLen, int flags, struct sockaddr *from,
    int *pFromLen)

int recv
    (int s, char *buf, int bufLen, int flags)

int recvmsg
    (int sd, struct msghdr *mp, int flags)

STATUS setsockopt
    (int s, int level, int optname, char *optval, int optlen)

STATUS getsockopt
    (int s, int level, int optname, char *optval, int *optlen)

STATUS getsockname
    (int s, struct sockaddr *name, int *namelen)

STATUS getpeername
    (int s, struct sockaddr *name, int *namelen)

STATUS shutdown
    (int s, int how)
```

DESCRIPTION

This library provides UNIX BSD 4.3 compatible socket calls. These calls may be used to open, close, read, and write sockets, either on the same CPU or over a network. The calling sequences of these routines are identical to UNIX BSD 4.3.

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**.

FIONBIO

Turns on / off non-blocking I / O.

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

FIONREAD

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

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

SIOCATMARK

Reports whether there is out-of-band data to be read on the socket. On the return of *ioctl***( )**, *atMark* will be TRUE (1) if there is out-of-band data, otherwise it will be FALSE (0).

```
status = ioctl (sFd, SIOCATMARK, &atMark);
```

INCLUDE FILES **types.h**, **mbuf.h**, **socket.h**, **socketvar.h**

SEE ALSO **netLib**, *VxWorks Programmer's Guide: Network*

spyLib

NAME **spyLib** – spy CPU activity library

SYNOPSIS *spyLibInit***( )** – initialize task CPU utilization tool package

```
void spyLibInit
    (void)
```

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**

---

sramDrv

NAME **sramDrv** – PCMCIA SRAM device driver

SYNOPSIS *sramDrv***( )** – install a PCMCIA SRAM memory driver
*sramMap***( )** – map PCMCIA memory onto a specified ISA address space
*sramDevCreate***( )** – create a PCMCIA memory disk device

```
STATUS sramDrv
    (int sock)

STATUS sramMap
    (int sock, int type, int start, int stop, int offset, int extraws)
```

```
BLK_DEV *sramDevCreate
    (int sock, int bytesPerBlk, int blksPerTrack, int nBlocks,
    int blkOffset)
```

DESCRIPTION This is a device driver for the SRAM PC card. The memory location and size are specified when the "disk" is created.

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: s*ramDrv***( )** to initialize the driver, and *sramDevCreate***( )** to create block devices. Additionally, the *sramMap***( )** routine is called directly to map the PCMCIA memory onto the ISA address space. Note that this routine does not use any mutual exclusion or synchronization mechanism; thus, special care must be taken in the multitasking environment.

Before using this driver, it must be initialized by calling s*ramDrv***( )**. This routine should be called only once, before any reads, writes, or calls to *sramDevCreate***( )** or *sramMap***( )**. It can be called from *usrRoot***( )** in **usrConfig.c** or at some later point.

SEE ALSO *VxWorks Programmer's Guide: I/O System*

straceLib

NAME **straceLib** – WindNet STREAMS message trace utility (STREAMS Opt.)

SYNOPSIS *strace***( )** – print STREAMS trace messages
*straceStop***( )** – stop the *strace***( )** task

```
strace
    (char *arg)

void straceStop
    (void)
```

DESCRIPTION This library consists of routines to support the **strace** utility. The **strace** utility is used to print to the trace file the trace output generated by STREAMS drivers and modules.

strerrLib

NAME

strerrLib – WindNet STREAMS error messages trace utility (STREAMS Opt.)

SYNOPSIS

*strerr***( )** – STREAMS error logger task
*strerrStop***( )** – stop the *strerr***( )** task

```
void strerr
    (void)

strerrStop
    (void)
```

DESCRIPTION

This library provides routines to support the **strerr** utility, which is used to print to the error file the error output generated by STREAMS drivers and modules.

strmLib

NAME

strmLib – driver for the WindNet STREAMS I/O system (STREAMS Opt.)

SYNOPSIS

*strmDriverAdd***( )** – add a STREAMS driver into the STREAMS subsystem
*strmModuleAdd***( )** – add a STREAMS module to the STREAMS subsystem
*strmTimeout***( )** – execute a routine in a specified length of time
*strmUntimeout***( )** – cancel previous *strmTimeout***( )** call
*strmSleep***( )** – suspend task execution pending occurrence of an event
*strmWakeup***( )** – resume suspended task execution
*strmSyncWriteAccess***( )** – access a shared data structure for synchronous writing
*strmWeld***( )** – connect the **q_next** pointers of arbitrary streams
*strmUnWeld***( )** – set the **q_next** pointers of streams queues to NULL
*strmPipe***( )** – create an intertask channel
*strmMkfifo***( )** – create a STREAMS FIFO

```
STATUS strmDriverAdd
    (char * pStrmDriverName, struct streamtab * pStreamtab, int cloneFlag,
    struct streamtab * pBuddyStreamtab, int flags, int sqlvl)

STATUS strmModuleAdd
    (char * pModuleName, struct streamtab * pStreamtab,
    struct streamtab * pBuddyStreamtab, int flags, int sqlvl)

int strmTimeout
    (void (*pFunc)(), caddr_t pArg, long ticks)
```

```
void strmUntimeout
     (int timeoutId)

int strmSleep
     (ulong_t event)

void strmWakeup
     (ulong_t event)

void strmSyncWriteAccess
     (queue_t *pQueue, mblk_t *pMsg, void (*pFuncToWrite)(PCELL, mblk_t *))

STATUS strmWeld
     (queue_t * pQdst1, queue_t * pQsrc1, queue_t * pQdst2,
     queue_t * pQsrc2, void (*pFunc)(), u32 arg0, u32 arg1)

STATUS strmUnWeld
     (queue_t * pQdst1, queue_t * pQdst2, void (*pFunc)(), u32 arg0,
     u32 arg1)

int strmPipe
     (int *fds)

intstrmMkfifo()
```

DESCRIPTION This library is a VxWorks device driver that provides the programmer interface to the WindNet STREAMS head, and therefore, to any STREAMS driver or module in the system. It provides the standard **open**, **close**, **read**, **write** and **ioctl** routines (including *select***( )**), as well as routines to load a driver or module, and the STREAMS *poll***( )** routine.

strmShow

NAME **strmShow** – library for STREAMS debugging (STREAMS Opt.)

SYNOPSIS
*strmDebugInit***( )** – include STREAMS debugging facility in VxWorks
*strmOpenStreamsShow***( )** – display all open streams in the STREAMS subsystem
*strmQueueShow***( )** – display all queues in a particular stream
*strmBandShow***( )** – display messages in a particular band
*strmMessageShow***( )** – display information about all messages in a stream
*strmQueueStatShow***( )** – display statistics about queues system-wide
*strmMsgStatShow***( )** – display statistics about system-wide usage of message blocks
*strmStatShow***( )** – display statistics about streams
*strmDriverModShow***( )** – list configuration information for modules and devices

```
STATUS strmDebugInit()

void strmOpenStreamsShow
     (char * msg)

int strmQueueShow
     (STHP sth, char * msg)

void strmBandShow
     (char * msg, queue_t * q, int pri)

void strmMessageShow
     (queue_t * q)

void strmQueueStatShow
     (void)

void strmMsgStatShow
     (void)

void strmStatShow
     (void)

void strmDriverModShow
     (int format)
```

DESCRIPTION This library consists of routines to facilitate debugging of STREAMS drivers developed under VxWorks. This library provides information about streams, queues, and messages. It supports the provision of system-wide statistics, as well as information about specific streams and queues.

strmSockLib

NAME **strmSockLib** – interface to STREAMS sockets (STREAMS Opt.)

SYNOPSIS *strmSockProtoAdd***( )** – add a new transport-protocol entry to STREAMS sockets
*strmSockProtoDelete***( )** – remove a protocol entry from the table
*strmSockDevNameGet***( )** – get the transport-provider device name

```
STATUS strmSockProtoAdd
     (int family, int type, char * devName)

STATUS strmSockProtoDelete
     (int family, int type)

char * strmSockDevNameGet
     (int family, int type)
```

DESCRIPTION This library provides facilities to add transport provider names to the list of transport providers. This list is used by the STREAMS socket call to open the appropriate transport provider. The library also provides facilities to delete the protocol entry from the list.

*strmSockLibInit***( )** initializes the table of function pointers with STREAMS sockets calls. This ensures the socket calls are configured for either BSD sockets or STREAMS sockets.

SEE ALSO *WindNet STREAMS Optional Component Supplement*

symLib

NAME **symLib** – symbol table subroutine library

SYNOPSIS

*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
*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

```
STATUS symLibInit
    (void)

SYMTAB_ID symTblCreate
    (int hashSizeLog2, BOOL sameNameOk, PART_ID symPartId)

STATUS symTblDelete
    (SYMTAB_ID symTblId)

STATUS symAdd
    (SYMTAB_ID symTblId, char *name, char *value, SYM_TYPE type,
    UINT16 group)

STATUS symRemove
    (SYMTAB_ID symTblId, char *name, SYM_TYPE type)

STATUS symFindByName
    (SYMTAB_ID symTblId, char *name, char **pValue, SYM_TYPE *pType)

STATUS symFindByNameAndType
    (SYMTAB_ID symTblId, char *name, char **pValue, SYM_TYPE *pType,
    SYM_TYPE sType, SYM_TYPE mask)
```

```
STATUS symFindByValue
    (SYMTAB_ID symTblId, UINT value, char * name, int * pValue,
    SYM_TYPE * pType)

STATUS symFindByValueAndType
    (SYMTAB_ID symTblId, UINT value, char * name, int * pValue,
    SYM_TYPE * pType, SYM_TYPE sType, SYM_TYPE mask)

SYMBOL *symEach
    (SYMTAB_ID symTblId, FUNCPTR routine, int routineArg)
```

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 character 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 **N_ABS**, **N_TEXT**, **N_DATA**, and **N_BSS** (defined in **a_out.h**); these are all even numbers, and any of them may be combined (via boolean or) with **N_EXT** (1). These values originate in the section names for a.out object code format, but the VxWorks system symbol table uses them as symbol types across all object formats. (The VxWorks system symbol table also occasionally includes additional types, in some object formats.)

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 of a specified name. The routine *symFindByValue***( )** finds the symbol with the value closest to a specified value. The routines *symFindByNameAndType***( )** and *symFindByValueAndType***( )** allow the symbol type to be used as an additional criterion in the searches.

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., *symFindByValue***( )**, 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 or not symbol tables can accept name clashes is set by a parameter when the symbol table is created with *symTblCreate***( )**. If name clashes are not allowed, *symAdd***( )** will return an error if there is an attempt to add a symbol with 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***( )**.

INCLUDE FILES **symLib.h**

SEE ALSO **loadLib**

symSyncLib

NAME **symSyncLib** – host / target symbol table synchronization

SYNOPSIS *symSyncLibInit***( )** – initialize host / target symbol table synchronization
*symSyncTimeoutSet***( )** – set WTX timeout

```
void symSyncLibInit()

UINT32 symSyncTimeoutSet
    (UINT32 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.

Synchronization is enabled by two actions: (1) the module is initialized by *symSyncLibInit***( )**, which is called automatically when **INCLUDE_SYM_TBL_SYNC** is defined in **configAll.h** or **config.h**; and (2) the target server is launched with the **-s** option.

If enabled, **symSyncLib** spawns a synchronization task, **tSymSync**, on the target. 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 also assures synchronization as new symbols are added either from the target or from host tools. The task waits for synchronization events on two channels: one for host events (via WTX event) and one for target events (via message queue).

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 will print a message indicating synchronization has been 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***( )**, *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 target are not synchronized; however, they are synchronized if they are loaded from the host).

SEE ALSO

tgtsvr

sysLib

NAME **sysLib** – system-dependent library

SYNOPSIS

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

```
STATUS sysClkConnect
    (FUNCPTR routine, int arg)
```

```
void sysClkDisable
     (void)

void sysClkEnable
     (void)

int sysClkRateGet
     (void)

STATUS sysClkRateSet
     (int ticksPerSecond)

STATUS sysAuxClkConnect
     (FUNCPTR routine, int arg)

void sysAuxClkDisable
     (void)

void sysAuxClkEnable
     (void)

int sysAuxClkRateGet
     (void)

STATUS sysAuxClkRateSet
     (int ticksPerSecond)

STATUS sysIntDisable
     (int intLevel)

STATUS sysIntEnable
     (int intLevel)

int sysBusIntAck
     (int intLevel)

STATUS sysBusIntGen
     (int intLevel, int vector)

STATUS sysMailboxConnect
     (FUNCPTR routine, int arg)

STATUS sysMailboxEnable
     (char *mailboxAdrs)

STATUS sysNvRamGet
     (char *string, int strLen, int offset)

STATUS sysNvRamSet
     (char *string, int strLen, int offset)

char *sysModel
     (void)
```

```
char * sysBspRev
     (void)

void sysHwInit
     (void)

char * sysPhysMemTop
     (void)

char *sysMemTop
     (void)

STATUS sysToMonitor
     (int startType)

int sysProcNumGet
     (void)

void sysProcNumSet
     (int procNum)

BOOL sysBusTas
     (char *adrs)

void sysScsiBusReset
     (WD_33C93_SCSI_CTRL *pSbic)

STATUS sysScsiInit
     (void)

STATUS sysScsiConfig
     (void)

STATUS sysLocalToBusAdrs
     (int adrsSpace, char *localAdrs, char **pBusAdrs)

STATUS sysBusToLocalAdrs
     (int adrsSpace, char *busAdrs, char **pLocalAdrs)

void sysSerialHwInit
     (void)

void sysSerialHwInit2
     (void)

void sysSerialReset
     (void)

SIO_CHAN * sysSerialChanGet
     (int channel)
```

DESCRIPTION This library provides board-specific routines.

NOTE: This is a generic man page 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 man page entry for **sysLib**. For example, the online UNIX man page for your BSP's version of **sysLib** can be accessed by viewing *bspName*_**sysLib**. See the *Tornado User's Guide: Getting Started* for information on accessing BSP-specific man pages.

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,* BSP-specific manual entry for **sysLib**

tapeFsLib

NAME

tapeFsLib – tape sequential device file system library

SYNOPSIS

*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

```
TAPE_VOL_DESC *tapeFsDevInit
    (char * volName, SEQ_DEV * pSeqDev, TAPE_CONFIG * pTapeConfig)

STATUS tapeFsInit()

STATUS tapeFsReadyChange
    (TAPE_VOL_DESC *pTapeVol)

STATUS tapeFsVolUnmount
    (TAPE_VOL_DESC *pTapeVol)
```

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

Specifies a block size value on the physical device and updates the value in the **SEQ_DEV** and **TAPE_VOL_DESC** structures, unless the 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; thus 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 permissable 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 preceeding 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 device status, thus setting appropriate **SEQ_DEV** fields.

MTRETEN
Re-tension 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*

taskArchLib

NAME **taskArchLib** – architecture-specific task management routines

SYNOPSIS *taskSRSet***( )** – set the task status register (MC680x0, MIPS, i386/i486)

```
STATUS taskSRSet
    (int tid, UINT16 sr)
```

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 SPARC.

INCLUDE FILES **regs.h**, **taskArchLib.h**

SEE ALSO **taskLib**

taskHookLib

NAME

taskHookLib – task hook library

SYNOPSIS

*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

```
void taskHookInit
    (void)

STATUS taskCreateHookAdd
    (FUNCPTR createHook)

STATUS taskCreateHookDelete
    (FUNCPTR createHook)

STATUS taskSwitchHookAdd
    (FUNCPTR switchHook)

STATUS taskSwitchHookDelete
    (FUNCPTR switchHook)

STATUS taskDeleteHookAdd
    (FUNCPTR deleteHook)

STATUS taskDeleteHookDelete
    (FUNCPTR deleteHook)
```

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

SYNOPSIS

*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

```
void taskHookShowInit
    (void)

void taskCreateHookShow
    (void)

void taskSwitchHookShow
    (void)

void taskDeleteHookShow
    (void)
```

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 **INCLUDE_SHOW_ROUTINES** is defined in **configAll.h**.

INCLUDE FILES **taskHookLib.h**

SEE ALSO **taskHookLib**, *VxWorks Programmer's Guide: Basic OS*

taskInfo

NAME **taskInfo** – task information library

SYNOPSIS

*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

```
STATUS taskOptionsSet
    (int tid, int mask, int newOptions)

STATUS taskOptionsGet
    (int tid, int *pOptions)

STATUS taskRegsGet
    (int tid, REG_SET *pRegs)

STATUS taskRegsSet
    (int tid, REG_SET *pRegs)

char *taskName
    (int tid)

int taskNameToId
    (char *name)
```

```
int taskIdDefault
    (int tid)

BOOL taskIsReady
    (int tid)

BOOL taskIsSuspended
    (int tid)

int taskIdListGet
    (int idList[], int maxTasks)
```

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

SYNOPSIS

*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

```
int taskSpawn
    (char *name, int priority, int options, int stackSize, FUNCPTR entryPt,
    int arg1, int arg2, int arg3, int arg4, int arg5, int arg6, int arg7,
    int arg8, int arg9, int arg10)

STATUS taskInit
    (WIND_TCB *pTcb, char *name, int priority, int options,
    char *pStackBase, int stackSize, FUNCPTR entryPt, int arg1, int arg2,
    int arg3, int arg4, int arg5, int arg6, int arg7, int arg8, int arg9,
    int arg10)

STATUS taskActivate
    (int tid)

void exit
    (int code)

STATUS taskDelete
    (int tid)

STATUS taskDeleteForce
    (int tid)

STATUS taskSuspend
    (int tid)

STATUS taskResume
    (int tid)

STATUS taskRestart
    (int tid)

STATUS taskPrioritySet
    (int tid, int newPriority)

STATUS taskPriorityGet
    (int tid, int *pPriority)
```

```
STATUS taskLock
    (void)

STATUS taskUnlock
    (void)

STATUS taskSafe
    (void)

STATUS taskUnsafe
    (void)

STATUS taskDelay
    (int ticks)

int taskIdSelf
    (void)

STATUS taskIdVerify
    (int tid)

WIND_TCB *taskTcb
    (int tid)
```

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 *taskDelete***( )** or *exit***( )**.

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

SYNOPSIS

*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

```
void taskShowInit
    (void)

STATUS taskInfoGet
    (int tid, TASK_DESC *pTaskDesc)

STATUS taskShow
    (int tid, int level)

void taskRegsShow
    (int tid)

STATUS taskStatusString
    (int tid, char *pString)
```

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 **INCLUDE_SHOW_ROUTINES** is defined in **configAll.h**.

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

SYNOPSIS

*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

```
STATUS taskVarInit
    (void)

STATUS taskVarAdd
    (int tid, int *pVar)

STATUS taskVarDelete
    (int tid, int *pVar)

int taskVarGet
    (int tid, int *pVar)

STATUS taskVarSet
    (int tid, int *pVar, int value)

int taskVarInfo
    (int tid, TASK_VAR varList[], int maxVars)
```

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*

tcic

NAME **tcic** – Databook TCIC/2 PCMCIA host bus adaptor chip driver

SYNOPSIS *tcicInit***( )** – initialize the TCIC chip

```
STATUS tcicInit
    (int ioBase, int intVec, int intLevel, FUNCPTR showRtn)
```

DESCRIPTION This library contains routines to manipulate the PCMCIA functions on the Databook DB86082 PCMCIA chip.

The initialization routine *tcicInit***( )** is the only global function and is included in the PCMCIA chip table **pcmciaAdapter**. If *tcicInit***( )** finds the TCIC chip, it registers all function pointers of the **PCMCIA_CHIP** structure.

tcicShow

NAME **tcicShow** – Databook TCIC/2 PCMCIA host bus adaptor chip show library

SYNOPSIS *tcicShow***( )** – show all configurations of the TCIC chip

```
void tcicShow
    (int sock)
```

DESCRIPTION This is a driver show routine for the Databook DB86082 PCMCIA chip. *tcicShow***( )** is the only global function and is installed in the PCMCIA chip table **pcmciaAdapter** in *pcmciaShowInit***( )**.

telnetLib

NAME **telnetLib** – telnet server library

SYNOPSIS *telnetInit***( )** – initialize the telnet daemon
*telnetd***( )** – VxWorks telnet daemon

```
void telnetInit
    (void)
```

```
void telnetd
    (void)
```

DESCRIPTION

This library provides a remote login facility for VxWorks. It uses the telnet protocol to enable users on remote systems to log in to VxWorks.

The telnet daemon, *telnetd***( )**, accepts remote telnet login requests and causes the shell's input and output to be redirected to the remote user. The telnet daemon is started by calling *telnetInit***( )**, which is called automatically when **INCLUDE_TELNET** is defined in **configAll.h**.

Internally, the telnet daemon provides a tty-like interface to the remote user through the use of the VxWorks pseudo-terminal driver, **ptyDrv**.

INCLUDE FILES **telnetLib.h**

SEE ALSO **ptyDrv**, **rlogLib**

---

tftpdLib

NAME **tftpdLib** – Trivial File Transfer Protocol server library

SYNOPSIS

*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

```
STATUS tftpdInit
    (int stackSize, int nDirectories, char **directoryNames,
    BOOL noControl, int maxConnections)

STATUS tftpdTask
    (int nDirectories, char **directoryNames, int maxConnections)

STATUS tftpdDirectoryAdd
    (char *fileName)

STATUS tftpdDirectoryRemove
    (char *fileName)
```

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."

INCLUDE FILES **tftpdLib.h**, **tftpLib.h**

SEE ALSO **tftpLib**, RFC 783 "TFTP Protocol", *VxWorks Programmer's Guide: Network*

tftpLib

NAME **tftpLib** – Trivial File Transfer Protocol (TFTP) client library

SYNOPSIS

*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

```
STATUS tftpXfer
    (char * pHost, int port, char * pFilename, char * pCommand,
    char * pMode, int * pDataDesc, int * pErrorDesc)

STATUS tftpCopy
    (char * pHost, int port, char * pFilename, char * pCommand,
    char * pMode, int fd)

TFTP_DESC * tftpInit
    (void)

STATUS tftpModeSet
    (TFTP_DESC * pTftpDesc, char * pMode)

STATUS tftpPeerSet
    (TFTP_DESC * pTftpDesc, char * pHostname, int port)
```

```
STATUS tftpPut
    (TFTP_DESC * pTftpDesc, char * pFilename, int fd, int clientOrServer)

STATUS tftpGet
    (TFTP_DESC * pTftpDesc, char * pFilename, int fd, int clientOrServer)

STATUS tftpInfoShow
    (TFTP_DESC * pTftpDesc)

STATUS tftpQuit
    (TFTP_DESC * pTftpDesc)

int tftpSend
    (TFTP_DESC * pTftpDesc, TFTP_MSG * pTftpMsg, int sizeMsg,
    TFTP_MSG * pTftpReply, int opReply, int blockReply, int * pPort)
```

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 (i.e., files to be read must be publicly readable, and files to be written must exist and be publicly writeable). 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);
```

INCLUDE FILES **tftpLib.h**

SEE ALSO **tftpdLib**, *VxWorks Programmer's Guide: Network*

tickLib

NAME

tickLib – clock tick support library

SYNOPSIS

*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

```
void tickAnnounce
    (void)

void tickSet
    (ULONG ticks)

ULONG tickGet
    (void)
```

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)

SYNOPSIS

*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)

```
int timer_cancel
    (timer_t timerid)

int timer_connect
    (timer_t timerid, VOIDFUNCPTR routine, int arg)

int timer_create
    (clockid_t clock_id, struct sigevent *evp, timer_t * pTimer)

int timer_delete
    (timer_t timerid)

int timer_gettime
    (timer_t timerid, struct itimerspec *value)

int timer_getoverrun
    (timer_t timerid)

int timer_settime
    (timer_t timerid, int flags, const struct itimerspec *value,
    struct itimerspec *ovalue)

int nanosleep
    (const struct timespec *rqtp, struct timespec *rmtp)
```

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

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

SYNOPSIS

*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

```
void timexInit
    (void)

void timexClear
    (void)

void timexFunc
    (int i, FUNCPTR func, int arg1, int arg2, int arg3, int arg4, int arg5,
    int arg6, int arg7, int arg8)
```

```
void timexHelp
    (void)

void timex
    (FUNCPTR func, int arg1, int arg2, int arg3, int arg4, int arg5,
    int arg6, int arg7, int arg8)

void timexN
    (FUNCPTR func, int arg1, int arg2, int arg3, int arg4, int arg5,
    int arg6, int arg7, int arg8)

void timexPost
    (int i, FUNCPTR func, int arg1, int arg2, int arg3, int arg4, int arg5,
    int arg6, int arg7, int arg8)

void timexPre
    (int i, FUNCPTR func, int arg1, int arg2, int arg3, int arg4, int arg5,
    int arg6, int arg7, int arg8)

void timexShow
    (void)
```

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**

---

ttyDrv

NAME **ttyDrv** – provide terminal device access to serial channels

SYNOPSIS

ttyDrv( ) – initialize the tty driver
ttyDevCreate( ) – create a VxWorks device for a serial channel

```
STATUS ttyDrv
    (void)

STATUS ttyDevCreate
    (char * name, SIO_CHAN * pSioChan, int rdBufSize, int wrtBufSize)
```

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

SYNOPSIS

*tyDevInit***( )** – initialize 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

```
STATUS tyDevInit
    (TY_DEV_ID pTyDev, int rdBufSize, int wrtBufSize, FUNCPTR txStartup)

void tyAbortFuncSet
    (FUNCPTR func)

void tyAbortSet
    (char ch)

void tyBackspaceSet
    (char ch)

void tyDeleteLineSet
    (char ch)

void tyEOFSet
    (char ch)
```

```
void tyMonitorTrapSet
    (char ch)

STATUS tyIoctl
    (TY_DEV_ID pTyDev, int request, int arg)

int tyWrite
    (TY_DEV_ID pTyDev, char *buffer, int nbytes)

int tyRead
    (TY_DEV_ID pTyDev, char *buffer, int maxbytes)

STATUS tyITx
    (TY_DEV_ID pTyDev, char *pChar)

STATUS tyIRd
    (TY_DEV_ID pTyDev, char inchar)
```

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 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*

unixDrv

NAME **unixDrv** – UNIX-file disk driver (VxSim)

SYNOPSIS
*unixDrv***( )** – install UNIX disk driver
*unixDiskDevCreate***( )** – create a UNIX disk device
*unixDiskInit***( )** – initialize a dosFs disk on top of UNIX

```
STATUS unixDrv
    (void)

BLK_DEV *unixDiskDevCreate
    (char *unixFile, int bytesPerBlk, int blksPerTrack, int nBlocks)

void unixDiskInit
    (char *unixFile, char *volName, int diskSize)
```

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 comand:

```
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

SYNOPSIS

*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

```
STATUS unld
    (void * nameOrId, int options)

STATUS unldByModuleId
    (MODULE_ID moduleId, int options)

STATUS unldByNameAndPath
    (char * name, char * path, int options)

STATUS unldByGroup
    (UINT16 group, int options)

MODULE_ID reld
    (void * nameOrId, int options)
```

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***( )**

INCLUDE FILES **unldLib.h**, **moduleLib.h**

SEE ALSO **loadLib**, **moduleLib**, *Tornado User's Guide: Cross-Development*

usrConfig

NAME **usrConfig** – user-defined system configuration library

SYNOPSIS *usrInit***( )** – user-defined system initialization routine
*usrRoot***( )** – the root task
*usrClock***( )** – user-defined system clock interrupt routine

```
void usrInit
    (int startType)

void usrRoot
    (char * pMemPoolStart, unsigned memPoolSize)

void usrClock()
```

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 **usrConfig.c**.

In an effort to simplify the configuration of VxWorks, this file is split into smaller files. These additional configuration source files are located in **../../src/config/usr***Xxx***.c** and are **#include**d into **usrConfig.c**.

The module **usrDepend.c** contains checks that guard against unsupported configurations suchas **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*

usrLib

NAME **usrLib** – user interface subroutine library

SYNOPSIS

*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
*cd***( )** – change the default directory
*pwd***( )** – print the current default directory
*copy***( )** – copy *in* (or stdin) to *out* (or stdout)
*copyStreams***( )** – copy from / to specified streams
*diskFormat***( )** – format a disk
*diskInit***( )** – initialize a file system on a block device
*squeeze***( )** – reclaim fragmented free space on an RT-11 volume
*ld***( )** – load an object module into memory
*ls***( )** – list the contents of a directory
*ll***( )** – do a long listing of directory contents
*lsOld***( )** – list the contents of an RT-11 directory

mkdir( ) – make a directory
rmdir( ) – remove a directory
rm( ) – remove a file
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

```
void help
    (void)

void netHelp
    (void)

void bootChange
    (void)

void periodRun
    (int secs, FUNCPTR func, int arg1, int arg2, int arg3, int arg4,
    int arg5, int arg6, int arg7, int arg8)

int period
    (int secs, FUNCPTR func, int arg1, int arg2, int arg3, int arg4,
    int arg5, int arg6, int arg7, int arg8)

void repeatRun
    (int n, FUNCPTR func, int arg1, int arg2, int arg3, int arg4, int arg5,
    int arg6, int arg7, int arg8)

int repeat
    (int n, FUNCPTR func, int arg1, int arg2, int arg3, int arg4, int arg5,
    int arg6, int arg7, int arg8)

int sp
    (FUNCPTR func, int arg1, int arg2, int arg3, int arg4, int arg5,
    int arg6, int arg7, int arg8, int arg9)
```

```
void checkStack
    (int taskNameOrId)

void i
    (int taskNameOrId)

void ti
    (int taskNameOrId)

void show
    (int objId, int level)

void ts
    (int taskNameOrId)

void tr
    (int taskNameOrId)

void td
    (int taskNameOrId)

void version
    (void)

void m
    (void *adrs, int width)

void d
    (void *adrs, int nunits, int width)

STATUS cd
    (char *name)

void pwd
    (void)

STATUS copy
    (char *in, char *out)

STATUS copyStreams
    (int inFd, int outFd)

STATUS diskFormat
    (char *devName)

STATUS diskInit
    (char *devName)

STATUS squeeze
    (char *devName)

MODULE_ID ld
    (int syms, BOOL noAbort, char *name)
```

```
STATUS ls
     (char *dirName, BOOL doLong)

STATUS ll
     (char *dirName)

STATUS lsOld
     (char *dirName)

STATUS mkdir
     (char *dirName)

STATUS rmdir
     (char *dirName)

STATUS rm
     (char *fileName)

void devs
     (void)

void lkup
     (char *substr)

void lkAddr
     (unsigned int addr)

STATUS mRegs
     (char *regName, int taskNameOrId)

int pc
     (int task)

void printErrno
     (int errNo)

void printLogo
     (void)

void logout
     (void)

void h
     (int size)

void spyReport
     (void)

void spyTask
     (int freq)

void spy
     (int freq, int ticksPerSec)
```

```
STATUS spyClkStart
    (int intsPerSec)

void spyClkStop
    (void)

void spyStop
    (void)

void spyHelp
    (void)
```

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 **spyLib**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

vmBaseLib

NAME

vmBaseLib – base virtual memory support library

SYNOPSIS

*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

```
STATUS vmBaseLibInit
    (int pageSize)

VM_CONTEXT_ID vmBaseGlobalMapInit
    (PHYS_MEM_DESC *pMemDescArray, int numDescArrayElements, BOOL enable)

STATUS vmBaseStateSet
    (VM_CONTEXT_ID context, void *pVirtual, int len, UINT stateMask,
    UINT state)

int vmBasePageSizeGet
    (void)
```

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

To include the bundled MMU support library in VxWorks, define **INCLUDE_MMU_BASIC** in **config.h**. Note that if **INCLUDE_MMU_FULL** is also defined in **config.h** and/or **configAll.h**, 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.)

SYNOPSIS

vmLibInit( ) – initialize the virtual memory support module
vmGlobalMapInit( ) – initialize global mapping
vmContextCreate( ) – create a new virtual memory context
vmContextDelete( ) – delete a virtual memory context
vmStateSet( ) – change the state of a block of virtual memory
vmStateGet( ) – get the state of a page of virtual memory
vmMap( ) – map physical space into virtual space
vmGlobalMap( ) – map physical pages to virtual space in shared global virtual memory
vmGlobalInfoGet( ) – get global virtual memory information
vmPageBlockSizeGet( ) – get the architecture-dependent page block size
vmTranslate( ) – translate a virtual address to a physical address
vmPageSizeGet( ) – return the page size
vmCurrentGet( ) – get the current virtual memory context
vmCurrentSet( ) – set the current virtual memory context
vmEnable( ) – enable or disable virtual memory
vmTextProtect( ) – write-protect a text segment

```
STATUS vmLibInit
    (int pageSize)

VM_CONTEXT_ID vmGlobalMapInit
    (PHYS_MEM_DESC *pMemDescArray, int numDescArrayElements, BOOL enable)

VM_CONTEXT_ID vmContextCreate
    (void)

STATUS vmContextDelete
    (VM_CONTEXT_ID context)

STATUS vmStateSet
    (VM_CONTEXT_ID context, void *pVirtual, int len, UINT stateMask,
    UINT state)

STATUS vmStateGet
    (VM_CONTEXT_ID context, void *pPageAddr, UINT *pState)

STATUS vmMap
    (VM_CONTEXT_ID context, void *virtualAddr, void *physicalAddr, UINT len)

STATUS vmGlobalMap
    (void *virtualAddr, void *physicalAddr, UINT len)

UINT8 *vmGlobalInfoGet
    (void)
```

```
int vmPageBlockSizeGet
    (void)

STATUS vmTranslate
    (VM_CONTEXT_ID context, void *virtualAddr, void **physicalAddr)

int vmPageSizeGet
    (void)

VM_CONTEXT_ID vmCurrentGet
    (void)

STATUS vmCurrentSet
    (VM_CONTEXT_ID context)

STATUS vmEnable
    (BOOL enable)

STATUS vmTextProtect
    (void)
```

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.

This library provides a facility *vmContextCreate***( )** for creating virtual memory contexts. 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

To include full MMU support (**vmLib**, and optionally, **vmShow**), define **INCLUDE_MMU_FULL** in **config.h**. Note that if **INCLUDE_MMU_BASIC** is also defined in **config.h** and / or **configAll.h**, the default is full MMU support.

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 interprocessor 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.)

SYNOPSIS

*vmShowInit***( )** – include virtual memory show facility
*vmContextShow***( )** – display the translation table for a context

```
void vmShowInit
    (void)

STATUS vmContextShow
    (VM_CONTEXT_ID context)
```

DESCRIPTION

This library contains virtual memory information display routines.

The routine *vmShowInit***( )** links this facility into the VxWorks system. It is called automatically if both **INCLUDE_MMU_FULL** and **INCLUDE_SHOW_ROUTINES** are defined in **configAll.h**.

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

SYNOPSIS

*vxTas***( )** – C-callable atomic test-and-set primitive
*vxMemProbe***( )** – probe an address for a bus error
*vxMemProbeAsi***( )** – probe address in ASI space for bus error (SPARC)
*vxSSEnable***( )** – enable the superscalar dispatch (MC68060)
*vxSSDisable***( )** – disable the superscalar dispatch (MC68060)
*vxPowerModeSet***( )** – set the power management mode (PowerPC)
*vxPowerModeGet***( )** – get the power management mode (PowerPC)
*vxPowerDown***( )** – place the processor in reduced-power mode (PowerPC)

```
BOOL vxTas
    (void * address)
```

```
STATUS vxMemProbe
    (char * adrs, int mode, int length, char * pVal)

STATUS vxMemProbeAsi
    (char * adrs, int mode, int length, char * pVal, int adrsAsi)

void vxSSEnable
    (void)

void vxSSDisable
    (void)

STATUS vxPowerModeSet
    (UINT32 mode)

UINT32 vxPowerModeGet
    (void)

STATUS vxPowerDown
    (void)
```

DESCRIPTION This module contains miscellaneous VxWorks support routines.

INCLUDE FILES **vxLib.h**

vxwLoadLib

NAME **vxwLoadLib** – object module class (WFC Opt.)

SYNOPSIS
*VXWModule::VXWModule***( )** – build module object from module ID
*VXWModule::VXWModule***( )** – load an object module at specified memory addresses
*VXWModule::VXWModule***( )** – load an object module into memory
*VXWModule::VXWModule***( )** – create and initialize an object module
*VXWModule::~VXWModule***( )** – unload an object module
*VXWModule::flags***( )** – get the flags associated with this module
*VXWModule::info***( )** – get information about object module
*VXWModule::name***( )** – get the name associated with module
*VXWModule::segFirst***( )** – find the first segment in module
*VXWModule::segGet***( )** – get (delete and return) the first segment from module
*VXWModule::segNext***( )** – find the next segment in module

```
VXWModule
    (MODULE_ID aModuleId)
```

```
VXWModule
    (int fd, int symFlag, char **ppText, char **ppData = 0,
    char **ppBss = 0)

VXWModule
    (int fd, int symFlag)

VXWModule
    (char *name, int format, int flags)

~VXWModule
    ()

int flags
    () const

STATUS info
    (MODULE_INFO * pModuleInfo) const

char * name
    () const

SEGMENT_ID segFirst
    () const

SEGMENT_ID segGet
    ()

SEGMENT_ID segNext
    (SEGMENT_ID segmentId) const
```

DESCRIPTION The **VXWModule** class provides a generic object-module loading facility. Any object files in a supported format 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. Modules may be loaded from any I/O stream.

INCLUDE FILE **vxwLoadLib.h**

SEE ALSO **usrLib**, **symLib**, **vxwMemPartLib**, *Tornado User's Guide: Cross-Development*

vxwLstLib

NAME

vxwLstLib – simple linked list class (WFC Opt.)

SYNOPSIS

VXWList::*VXWList*( ) – initialize a list
VXWList::*VXWList*( ) – initialize a list as a copy of another
VXWList::~*VXWList*( ) – free up a list
VXWList::*add*( ) – add a node to the end of list
VXWList::*concat*( ) – concatenate two lists
VXWList::*count*( ) – report the number of nodes in a list
VXWList::*extract*( ) – extract a sublist from list
VXWList::*find*( ) – find a node in list
VXWList::*first*( ) – find first node in list
VXWList::*get*( ) – delete and return the first node from list
VXWList::*insert*( ) – insert a node in list after a specified node
VXWList::*last*( ) – find the last node in list
VXWList::*next*( ) – find the next node in list
VXWList::*nStep*( ) – find a list node *nStep* steps away from a specified node
VXWList::*nth*( ) – find the Nth node in a list
VXWList::*previous*( ) – find the previous node in list
VXWList::*remove*( ) – delete a specified node from list

```
VXWList
     ()

VXWList
     (const VXWList &);

~VXWList
     ()

void add
     (NODE *pNode)

void concat
     (VXWList &aList)

int count
     () const

LIST extract
     (NODE *pStart, NODE *pEnd)

int find
     (NODE *pNode) const

NODE * first
     () const
```

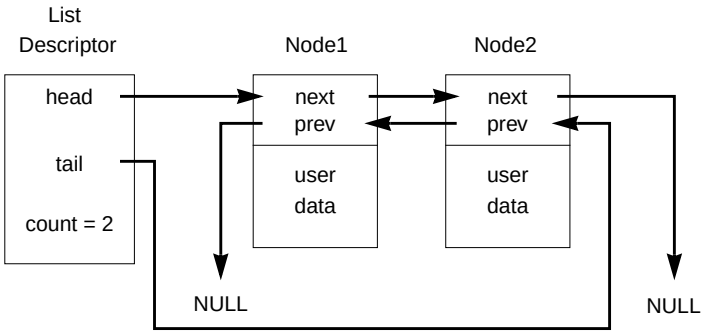
```
NODE * get
     ()

void insert
     (NODE *pPrev, NODE *pNode)

NODE * last
     () const

NODE * next
     (NODE *pNode) const

NODE * nStep
     (NODE *pNode, int nStep) const

NODE * nth
     (int nodeNum) const

NODE * previous
     (NODE *pNode) const

void remove
     (NODE *pNode)
```

DESCRIPTION

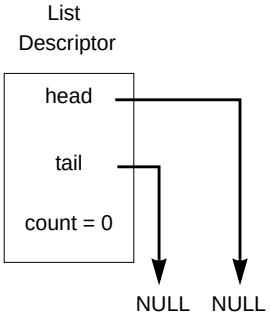
The VXWList class supports the creation and maintenance of a doubly linked list. The class contains pointers to the first and last nodes in the list, and a count of the number of nodes in the list. The nodes in the list are derived from the structure NODE, which provides two pointers: **NODE::next** and **NODE::previous**. Both the forward and backward chains are terminated with a NULL pointer.

The VXWList class 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 (such as a mutual-exclusion semaphore).

NON-EMPTY LIST



EMPTY LIST



WARNINGS Use only single inheritance! This class is an interface to the VxWorks library **lstLib**. More sophisticated alternatives are available in the **Tools.h**++ or the Booch components class libraries.

EXAMPLE The following example illustrates how to create a list by deriving elements from NODE and putting them on a VXWList.

```
class myListNode : public NODE
    {
  public:
    myListNode ()
     {
     }
  private:
    };
VXWList      myList;
myListNode   a, b, c;
NODE       * pEl = &c;
void useList ()
    {
    myList.add (&a);
    myList.insert (pEl, &b);
    }
```

INCLUDE FILES **vxwLstLib.h**

1

vxwMemPartLib

NAME

vxwMemPartLib – memory partition classes (WFC Opt.)

SYNOPSIS

*VXWMemPart::VXWMemPart***( )** – create a memory partition
*VXWMemPart::addToPool***( )** – add memory to a memory partition
*VXWMemPart::alignedAlloc***( )** – allocate aligned memory from partition
*VXWMemPart::alloc***( )** – allocate a block of memory from partition
*VXWMemPart::findMax***( )** – find the size of the largest available free block
*VXWMemPart::free***( )** – free a block of memory in partition
*VXWMemPart::info***( )** – get partition information
*VXWMemPart::options***( )** – set the debug options for memory partition
*VXWMemPart::realloc***( )** – reallocate a block of memory in partition
*VXWMemPart::show***( )** – show partition blocks and statistics

```
VXWMemPart
     (char *pool, unsigned poolSize)

STATUS addToPool
     (char *pool, unsigned poolSize)

void * alignedAlloc
     (unsigned nBytes, unsigned alignment)

void * alloc
     (unsigned nBytes)

int findMax
     () const

STATUS free
     (char *pBlock)

STATUS info
     (MEM_PART_STATS *pPartStats) const

STATUS options
     (unsigned options)

void * realloc
     (char *pBlock, int nBytes)

STATUS show
     (int type = 0) const
```

DESCRIPTION

The **VXWMemPart** class provides core facilities for managing the allocation of blocks of memory from ranges of memory called memory partitions.

The allocation of memory, using routines such as *VXWMemPart::alloc*( ), is done with a first-fit algorithm. Adjacent blocks of memory are coalesced when they are freed with *VXWMemPart::free*( ). There is also a routine provided for allocating memory aligned to a specified boundary from a specific memory partition, *VXWMemPart::alignedAlloc*( ).

CAVEATS

Architectures have various alignment constraints. To provide optimal performance, *VXWMemPart::alloc*( ) 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
68K	4	8
SPARC	8	12
MIPS	8	12
i960	16	16

INCLUDE FILES **vxwMemPartLib.h**

SEE ALSO **vxwSmLib**

vxwMsgQLib

NAME **vxwMsgQLib** – message queue classes (WFC Opt.)

SYNOPSIS

VXWMsgQ::VXWMsgQ( ) – create and initialize a message queue
VXWMsgQ::VXWMsgQ( ) – build message-queue object from ID
VXWMsgQ::~VXWMsgQ( ) – delete message queue
VXWMsgQ::send( ) – send a message to message queue
VXWMsgQ::receive( ) – receive a message from message queue
VXWMsgQ::numMsgs( ) – report the number of messages queued
VXWMsgQ::info( ) – get information about message queue
VXWMsgQ::show( ) – show information about a message queue

```
VXWMsgQ
    (int maxMsgs, int maxMsgLen, int opts)

VXWMsgQ
    (MSG_Q_ID id)

virtual ~VXWMsgQ
    ()
```

```
STATUS send
    (char * buffer, UINT nBytes, int timeout, int pri)

int receive
    (char * buffer, UINT nBytes, int timeout)

int numMsgs
    () const

STATUS info
    (MSG_Q_INFO *pInfo) const

STATUS show
    (int level) const
```

DESCRIPTION

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

CREATING AND USING MESSAGE QUEUES

The message-queue constructor takes parameters to specify the maximum number of messages that can be queued to that message queue and the maximum length in bytes of each message. Enough buffer space is pre-allocated to accommodate the specified number of messages of specified length.

A task or interrupt service routine sends a message to a message queue with *VXWMsgQ::send***( )**. 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 *VXWMsgQ::receive***( )**. If any messages are already available in the message queue's buffer, the first message is immediately dequeued and returned to the caller. If no messages are available, the calling task blocks and joins 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 *VXWMsgQ::send***( )** and *VXWMsgQ::receive***( )** 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 *VXWMsgQ::send***( )** routine allows the priority of a message to be specified as either normal (**MSG_PRI_NORMAL**) or urgent (**MSG_PRI_URGENT**). Normal priority messages are added to the tail of the list of queued messages, while urgent priority messages are added to the head of the list.

INCLUDE FILES **vxwMsgQLib.h**

SEE ALSO **pipeDrv**, **msgQSmLib**, *VxWorks Programmer's Guide: Basic OS*

vxwRngLib

NAME **vxwRngLib** – ring buffer class (WFC Opt.)

SYNOPSIS

*VXWRingBuf::VXWRingBuf***( )** – create an empty ring buffer
*VXWRingBuf::VXWRingBuf***( )** – build ring-buffer object from existing ID
*VXWRingBuf::~VXWRingBuf***( )** – delete ring buffer
*VXWRingBuf::get***( )** – get characters from ring buffer
*VXWRingBuf::put***( )** – put bytes into ring buffer
*VXWRingBuf::flush***( )** – make ring buffer empty
*VXWRingBuf::freeBytes***( )** – determine the number of free bytes in ring buffer
*VXWRingBuf::isEmpty***( )** – test whether ring buffer is empty
*VXWRingBuf::isFull***( )** – test whether ring buffer is full (no more room)
*VXWRingBuf::moveAhead***( )** – advance ring pointer by *n* bytes
*VXWRingBuf::nBytes***( )** – determine the number of bytes in ring buffer
*VXWRingBuf::putAhead***( )** – put a byte ahead in a ring buffer without moving ring pointers

```
VXWRingBuf
    (int nbytes)

VXWRingBuf
    (RING_ID aRingId)

~VXWRingBuf
    ()

int get
    (char * buffer, int maxbytes)

int put
    (char * buffer, int nBytes)
```

```
void flush
     ()

int freeBytes
     () const

BOOL isEmpty
     () const

BOOL isFull
     () const

void moveAhead
     (int n)

int nBytes
     () const

void putAhead
     (char byte, int offset)
```

DESCRIPTION

The **VXWRingBuf** class 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 *VXWRingBuf::get***( )**) and a writer task (invoking *VXWRingBuf::put***( )**) 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 (for example, a mutual-exclusion semaphore guarding a ring buffer).

INCLUDE FILES

vxwRngLib.h

vxwSemLib

NAME

vxwSemLib – semaphore classes (WFC Opt.)

SYNOPSIS

*VXWSem::VXWSem***( )** – build semaphore object from semaphore ID
*VXWSem::~VXWSem***( )** – delete a semaphore
*VXWSem::give***( )** – give a semaphore
*VXWSem::take***( )** – take a semaphore
*VXWSem::flush***( )** – unblock every task pended on a semaphore

*VXWSem::id***( )** – reveal underlying semaphore ID
*VXWSem::info***( )** – get a list of task IDs that are blocked on a semaphore
*VXWSem::show***( )** – show information about a semaphore
*VXWCSem::VXWCSem***( )** – create and initialize a counting semaphore
*VXWBSem::VXWBSem***( )** – create and initialize a binary semaphore
*VXWMSem::VXWMSem***( )** – create and initialize a mutual-exclusion semaphore
*VXWMSem::giveForce***( )** – give a mutual-exclusion semaphore without restrictions

```
VXWSem
    (SEM_ID id)

virtual ~VXWSem
    ()

STATUS give
    ()

STATUS take
    (int timeout)

STATUS flush
    ()

SEM_ID id
    () const

STATUS info
    (int idList [], int maxTasks) const

STATUS show
    (int level) const

VXWCSem
    (int opts, int count)

VXWBSem
    (int opts, SEM_B_STATE iState)

VXWMSem
    (int opts)

STATUS giveForce
    ()
```

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. The VXWSem class provides semaphore routines common to all VxWorks semaphore types. For all types, the two basic operations are *VXWSem::take***( )** and *VXWSem::give***( )**, the acquisition or relinquishing of a semaphore.

Semaphore creation and initialization is handled by the following classes, which inherit the basic operations from **VXWSem**:

VXWBSem – binary semaphores
VXWCSem – counting semaphores
VXWMSem – mutual exclusion semaphores

Two additional semaphore classes provide semaphores that operate over shared memory (with the optional product VxMP). These classes also inherit from **VXWSmNameLib**; they are described in **vxwSmLib**. The following are the class names for these shared-memory semaphores:

VXWSmBSem – shared-memory binary semaphores
VXWSmCSem – shared-memory counting semaphores

Binary semaphores offer the greatest speed and the broadest applicability.

The **VXWSem** class provides all other semaphore operations, including routines for semaphore control, deletion, and information.

SEMAPHORE CONTROL

The *VXWSem::take***( )** call acquires a specified semaphore, blocking the calling task or making the semaphore unavailable. All semaphore types support a timeout on the *VXWSem::take***( )** 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 *VXWSem::take***( )** times out, it returns ERROR. Refer to the library of the specific semaphore type for the exact behavior of this operation.

The *VXWSem::give***( )** 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 *VXWSem::flush***( )** call may be used to atomically unblock all tasks pended on a semaphore queue; that is, it unblocks all tasks 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 *VXWSem::flush***( )**; it is not analogous to *VXWSem::give***( )**.

SEMAPHORE DELETION

The *VXWSem::~VXWSem***( )** destructor 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 *VXWSem::info***( )** 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.

INCLUDE FILES **vxwSemLib.h**

SEE ALSO **vxwTaskLib**, **vxwSmLib**, *VxWorks Programmer's Guide: Basic OS*

vxwSmLib

NAME **vxwSmLib** – shared memory objects (WFC Opt.)

SYNOPSIS

*VXWSmBSem::VXWSmBSem***( )** – create and initialize binary shared-memory semaphore
*VXWSmBSem::VXWSmBSem***( )** – build a binary shared-memory semaphore object
*VXWSmCSem::VXWSmCSem***( )** – create and initialize a shared memory counting semaphore
*VXWSmCSem::VXWSmCSem***( )** – build a shared-memory counting semaphore object
*VXWSmMsgQ::VXWSmMsgQ***( )** – create and initialize a shared-memory message queue
*VXWSmMemPart::VXWSmMemPart***( )** – create a shared memory partition
*VXWSmMemBlock::VXWSmMemBlock***( )** – allocate a block of memory from the shared memory system partition
*VXWSmMemBlock::VXWSmMemBlock***( )** – allocate memory for an array from the shared memory system partition
*VXWSmMemBlock::~VXWSmMemBlock***( )** – free a shared memory system partition block of memory
*VXWSmMemBlock::baseAddress***( )** – address of shared-memory block

```
VXWSmBSem
    (int opts, SEM_B_STATE istate, char * name = 0)

VXWSmBSem
    (char * name, int waitType)

VXWSmCSem
    (int opts, int icount, char * name = 0)

VXWSmCSem
    (char * name, int waitType)

VXWSmMsgQ
    (int maxMsgs, int maxMsgLength, int options)
```

```
VXWSmMemPart
    (char *pool, unsigned poolSize)

VXWSmMemBlock
    (int nBytes)

VXWSmMemBlock
    (int nElems, int sizeOfElem)

virtual ~VXWSmMemBlock
    ()

void * baseAddress
    () const
```

DESCRIPTION This library defines wrapper classes for VxWorks shared-memory objects: **VXWSmBSem** (shared-memory binary semaphores), **VXWSmCSem** (shared-memory counting semaphores), **VXWSmMsgQ** (shared-memory message queues), **VXWSmMemPart** (shared memory partitions), and **VXWSmMemBlock** (shared memory blocks).

INHERITANCE All of the shared-memory object wrappers inherit the public members of **VXWSmNameLib**.

The **VXWSmBSem** and **VXWSmCSem** classes also inherit from **VXWSem**; **VXWSmMsgQ** also inherits from **VXWMsgQ**; **VXWSmMemPart** also inherits from **VXWMemPArt**.

AVAILABILITY This module depends on code that is distributed as a component of the unbundled shared memory objects support option, VxMP.

INCLUDE FILES **vxwSmLib.h**

SEE ALSO **vxwSmNameLib**, **vxwMsgQLib**, **vxwMemPartLib**, **vxwSemLib**, *VxWorks Programmer's Guide: Shared-Memory Objects*

vxwSmNameLib

NAME **vxwSmNameLib** – naming behavior common to all shared memory classes (WFC Opt.)

SYNOPSIS *VXWSmName::~VXWSmName***( )** – remove an object from the shared memory objects name database
*VXWSmName::nameSet***( )** – define a name string in the shared-memory name database
*VXWSmName::nameGet***( )** – get name and type of a shared memory object
*VXWSmName::nameGet***( )** – get name of a shared memory object

```
virtual ~VXWSmName
    ();

virtual STATUS nameSet
    (char * name) = 0;

STATUS nameGet
    (char * name, int * pType, int waitType)

STATUS nameGet
    (char * name, int waitType)
```

DESCRIPTION

This class library provides facilities for managing entries in 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 *VXWSmName::VXWSmName***( )**. They are removed by *VXWSmName::~VXWSmName***( )**.

Name database contents can be viewed using *smNameShow***( )**.

The maximum number of names to be entered in the database **SM_OBJ_MAX_NAME** is defined in **configAll.h**. 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 **smNameShow**.

CONFIGURATION

Before routines in this library can be called, the shared memory object facility must be initialized by calling *usrSmObjInit***( )**, which is found in **src/config/usrSmObj.c**. This is done automatically from the root task, *usrRoot***( )**, in **usrConfig** if **INCLUDE_SM_OBJ** is defined in **configAll.h**.

AVAILABILITY

This module depends on code that is distributed as a component of the unbundled shared memory objects support option, VxMP.

INCLUDE FILES

vxwSmNameLib.h

SEE ALSO

smNameLib, **smNameShow**, **vxwSmLib**, **smObjShow**, *usrSmObjInit***( )**, *VxWorks Programmer's Guide: Shared Memory Objects*

vxwSymLib

NAME

vxwSymLib – symbol table class (WFC Opt.)

SYNOPSIS

*VXWSymTab::VXWSymTab***( )** – create a symbol table
*VXWSymTab::VXWSymTab***( )** – create a symbol-table object
*VXWSymTab::~VXWSymTab***( )** – delete a symbol table
*VXWSymTab::add***( )** – create and add a symbol to a symbol table, including a group number
*VXWSymTab::each***( )** – call a routine to examine each entry in a symbol table
*VXWSymTab::findByName***( )** – look up a symbol by name
*VXWSymTab::findByNameAndType***( )** – look up a symbol by name and type
*VXWSymTab::findByValue***( )** – look up a symbol by value
*VXWSymTab::findByValueAndType***( )** – look up a symbol by value and type
*VXWSymTab::remove***( )** – remove a symbol from a symbol table

```
VXWSymTab
    (int hashSizeLog2, BOOL sameNameOk, PART_ID symPartId)

VXWSymTab
    (SYMTAB_ID aSymTabId)

~VXWSymTab
    ()

STATUS add
    (char *name, char *value, SYM_TYPE type, UINT16 group)

SYMBOL * each
    (FUNCPTR routine, int routineArg)

STATUS findByName
    (char *name, char **pValue, SYM_TYPE *pType) const

STATUS findByNameAndType
    (char *name, char **pValue, SYM_TYPE *pType, SYM_TYPE goalType,
    SYM_TYPE mask) const

STATUS findByValue
    (UINT value, char *name, int *pValue, SYM_TYPE *pType) const

STATUS findByValueAndType
    (UINT value, char *name, int *pValue, SYM_TYPE *pType,
    SYM_TYPE goalType, SYM_TYPE mask) const

STATUS remove
    (char *name, SYM_TYPE type)
```

DESCRIPTION

This class 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 character 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 VXWSymTab class 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 the VxWorks system symbol table is configured into your target system, you can use the VXWSymTab class to manipulate it based on its symbol-table ID, recorded in the global **sysSymTbl**; see *VXWSymTab::VXWSymTab***( )** to construct an object based on this global. In the VxWorks target-resident global symbol table, the values for **SYM_TYPE** are **N_ABS**, **N_TEXT**, **N_DATA**, and **N_BSS** (defined in **a_out.h**); these are all even numbers, and any of them may be combined (via boolean or) with **N_EXT** (1). These values originate in the section names for a.out object code format, but the VxWorks system symbol table uses them as symbol types across all object formats. (The VxWorks system symbol table also occasionally includes additional types, in some object formats.)

All operations on a symbol table are interlocked by means of a mutual-exclusion semaphore in the symbol table structure.

Symbols are added to a symbol table with *VXWSymTab::add***( )**. Each symbol in the symbol table has a name, a value, and a type. Symbols are removed from a symbol table with *VXWSymTab::remove***( )**.

Symbols can be accessed by either name or value. The routine *VXWSymTab::findByName***( )** searches the symbol table for a symbol of a specified name. The routine *VXWSymTab::findByValue***( )** finds the symbol with the value closest to a specified value. The routines *VXWSymTab::findByNameAndType***( )** and *VXWSymTab::findByValueAndType***( )** allow the symbol type to be used as an additional criterion in the searches.

Symbols in the symbol table are hashed by name into a hash table for fast look-up by name, for instance with *VXWSymTab::findByName***( )**. The size of the hash table is specified during the creation of a symbol table. Look-ups by value, such as with *VXWSymTab::findByValue***( )**, must search the table linearly; these look-ups can thus be much slower.

The routine *VXWSymTab::each***( )** 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 or not symbol tables can accept name clashes is set by a parameter when the symbol table is created with *VXWSymTab::VXWSymTab***( )**. If name clashes are not allowed, *VXWSymTab::add***( )** 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 is not an error. In such cases, *VXWSymTab::findByName***( )** returns the value most recently added, although all versions of the symbol can be found by *VXWSymTab::each***( )**.

INCLUDE FILES **vxwSymLib.h**

SEE ALSO **vxwLoadLib**

vxwTaskLib

NAME **vxwTaskLib** – task class (WFC Opt.)

SYNOPSIS

*VXWTask::VXWTask***( )** – initialize a task object
*VXWTask::VXWTask***( )** – create and spawn a task
*VXWTask::VXWTask***( )** – initialize a task with a specified stack
*VXWTask::~VXWTask***( )** – delete a task
*VXWTask::activate***( )** – activate a task
*VXWTask::deleteForce***( )** – delete a task without restriction
*VXWTask::envCreate***( )** – create a private environment
*VXWTask::errNo***( )** – retrieve error status value
*VXWTask::errNo***( )** – set error status value
*VXWTask::id***( )** – reveal task ID
*VXWTask::info***( )** – get information about a task
*VXWTask::isReady***( )** – check if task is ready to run
*VXWTask::isSuspended***( )** – check if task is suspended
*VXWTask::kill***( )** – send a signal to task
*VXWTask::name***( )** – get the name associated with a task ID
*VXWTask::options***( )** – examine task options
*VXWTask::options***( )** – change task options
*VXWTask::priority***( )** – examine the priority of task
*VXWTask::priority***( )** – change the priority of a task
*VXWTask::registers***( )** – set a task's registers
*VXWTask::registers***( )** – get task registers from the TCB
*VXWTask::restart***( )** – restart task
*VXWTask::resume***( )** – resume task
*VXWTask::show***( )** – display the contents of task registers
*VXWTask::show***( )** – display task information from TCBs
*VXWTask::sigqueue***( )** – send a queued signal to task
*VXWTask::SRSet***( )** – set the task status register (MC680x0, MIPS, i386/i486)
*VXWTask::statusString***( )** – get task status as a string
*VXWTask::suspend***( )** – suspend task
*VXWTask::tcb***( )** – get the task control block
*VXWTask::varAdd***( )** – add a task variable to task
*VXWTask::varDelete***( )** – remove a task variable from task
*VXWTask::varGet***( )** – get the value of a task variable
*VXWTask::varInfo***( )** – get a list of task variables
*VXWTask::varSet***( )** – set the value of a task variable

```
VXWTask
    (int tid)

VXWTask
    (char * name, int priority, int options, int stackSize,
    FUNCPTR entryPoint, int arg1 = 0, int arg2 = 0, int arg3 = 0,
    int arg4 = 0, int arg5 = 0, int arg6 = 0, int arg7 = 0, int arg8 = 0,
    int arg9 = 0, int arg10 = 0)

VXWTask
    (WIND_TCB * pTcb, char * name, int priority, int options,
    char * pStackBase, int stackSize, FUNCPTR entryPoint, int arg1 = 0,
    int arg2 = 0, int arg3 = 0, int arg4 = 0, int arg5 = 0, int arg6 = 0,
    int arg7 = 0, int arg8 = 0, int arg9 = 0, int arg10 = 0)

virtual ~VXWTask
    ()

STATUS activate
    ()

STATUS deleteForce
    ()

STATUS envCreate
    (int envSource)

int errNo
    () const

STATUS errNo
    (int errorValue)

int id
    () const

STATUS info
    (TASK_DESC *pTaskDesc) const

BOOL isReady
    () const

BOOL isSuspended
    () const

int kill
    (int signo)

char * name
    () const

STATUS options
    (int *pOptions) const
```

```
STATUS options
     (int mask, int newOptions)

STATUS priority
     (int *pPriority) const

STATUS priority
     (int newPriority)

STATUS registers
     (const REG_SET *pRegs)

STATUS registers
     (REG_SET *pRegs) const

STATUS restart
     ()

STATUS resume
     ()

void show
     () const

STATUS show
     (int level) const

int sigqueue
     (int signo, const union sigval value)

STATUS SRSet
     (UINT16 sr)

STATUS statusString
     (char *pString) const

STATUS suspend
     ()

WIND_TCB * tcb
     () const

STATUS varAdd
     (int * pVar)

STATUS varDelete
     (int * pVar)

int varGet
     (int * pVar) const

int varInfo
     (TASK_VAR varList [], int maxVars) const
```

```
STATUS varSet
    (int * pVar, int value)
```

DESCRIPTION This library provides the interface to the VxWorks task management facilities. This class library provides task control services, programmatic access to task information and debugging features, and higher-level task information display routines.

TASK CREATION Tasks are created with the constructor *VXWTask::VXWTask***( )**. 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 method *VXWTask::activate***( )**.

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.

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 deleted with the *exit***( )** routine, or explicitly with the *delete* operator, which arranges to call the class destructor *VXWTask::~VXWTask***( )**.

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.

A task can protect itself from deletion by taking a mutual-exclusion semaphore created with the **SEM_DELETE_SAFE** option (see **vxwSemLib** 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 The following methods control task state: *VXWTask::resume***( )**, *VXWTask::suspend***( )**, *VXWTask::restart***( )**, *VXWTask::priority***( )**, and *VXWTask::registers***( )**.

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 or examined using *VXWTask:priority***( )**.

INCLUDE FILES **taskLib.h**

SEE ALSO **taskLib**, **taskHookLib**, **vxwSemLib**, **kernelLib**, *VxWorks Programmer's Guide: Basic OS*

vxwWdLib

NAME

vxwWdLib – watchdog timer class (WFC Opt.)

SYNOPSIS

*VXWWd::VXWWd***( )** – construct a watchdog timer
*VXWWd::VXWWd***( )** – construct a watchdog timer
*VXWWd::~VXWWd***( )** – destroy a watchdog timer
*VXWWd::cancel***( )** – cancel a currently counting watchdog
*VXWWd::start***( )** – start a watchdog timer

```
VXWWd
    ()

VXWWd
    (WDOG_ID aWdId)

~VXWWd
    ()

STATUS cancel
    ()

STATUS start
    (int delay, FUNCPTR pRoutine, int parameter)
```

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, it can be started with *VXWWd::start***( )**. The *VXWWd::start***( )** 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 *VXWWd::cancel***( )** is called first to cancel the timer) the timeout routine is invoked with the parameter specified in the *VXWWd::start***( )** 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 *VXWWd::start***( )** invocation; there is no need to cancel a timer with *VXWWd::cancel***( )** 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.

```
VXWWd *pWd = new VXWWd;
pWd->start (60, logMsg, "Help, I've timed out!");
maybeSlowRoutine ();     /* user-supplied routine */
delete pWd;
```

INCLUDE FILES **vxwWdLib.h**

SEE ALSO **wdLib**, **logLib**, *VxWorks Programmer's Guide: Basic OS Tornado User's Guide: Cross-Development*

wd33c93Lib

NAME **wd33c93Lib** – WD33C93 SCSI-Bus Interface Controller (SBIC) library

SYNOPSIS *wd33c93CtrlInit***( )** – initialize the user-specified fields in an SBIC structure
*wd33c93Show***( )** – display the values of all readable WD33C93 chip registers

```
STATUS wd33c93CtrlInit
    (int *pSbic, int scsiCtrlBusId, UINT defaultSelTimeOut,
    int scsiPriority)

int wd33c93Show
    (int *pScsiCtrl)
```

DESCRIPTION This library contains the main interface routines to the Western Digital WD33C93 and WD33C93A SCSI-Bus Interface Controllers (SBIC). However, these routines simply switch the calls to either the SCSI-1 or SCSI-2 drivers, implemented in **wd33c93Lib1** and **wd33c93Lib2** respectively, as configued by the Board Support Package (BSP).

In order to configure the SCSI-1 driver, which depends upon **scsi1Lib**, the *wd33c93CtrlCreate***( )** routine, defined in **wd33c93Lib1**, must be invoked. Similarly, *wd33c93CtrlCreateScsi2***( )**, defined in **wd33c93Lib2** and dependent on **scsi2Lib**, must be called to configure and initialize the SCSI-2 driver.

INCLUDE FILES **wd33c93.h**, **wd33c93_1.h**, **wd33c93_2.h**

SEE ALSO **scsiLib**, **scsi1Lib**, **scsi2Lib**, **wd33c93Lib1**, **wd33c93Lib2**, *Western Digital WD33C92/93 SCSI-Bus Interface Controller, Western Digital WD33C92A/93A SCSI-Bus Interface Controller, VxWorks Programmer's Guide: I/O System*

wd33c93Lib1

NAME **wd33c93Lib1** – WD33C93 SCSI-Bus Interface Controller library (SCSI-1)

SYNOPSIS *wd33c93CtrlCreate***( )** – create and partially initialize a WD33C93 SBIC structure

```
WD_33C93_SCSI_CTRL *wd33c93CtrlCreate
    (UINT8 *sbicBaseAdrs, int regOffset, UINT clkPeriod, int devType,
    FUNCPTR sbicScsiReset, FUNCPTR sbicDmaBytesIn, FUNCPTR sbicDmaBytesOut)
```

DESCRIPTION This library contains part of the I/O driver for the Western Digital WD33C93 and WD33C93A SCSI-Bus Interface Controllers (SBIC). The driver routines in this library depend on the SCSI-1 version of the SCSI standard; for driver routines that do not depend on SCSI-1 or SCSI-2, and for overall SBIC driver documentation, see **wd33c93Lib**.

USER-CALLABLE ROUTINES Most of the routines in this driver are accessible only through the I/O system. The only exception in this portion of the driver is *wd33c93CtrlCreate***( )**, which creates a controller structure.

INCLUDE FILES **wd33c93.h**, **wd33c93_1.h**

SEE ALSO **scsiLib**, **scsi1Lib**, **wd33c93Lib**

wd33c93Lib2

NAME **wd33c93Lib2** – WD33C93 SCSI-Bus Interface Controller library (SCSI-2)

SYNOPSIS *wd33c93CtrlCreateScsi2***( )** – create and partially initialize an SBIC structure

```
WD_33C93_SCSI_CTRL *wd33c93CtrlCreateScsi2
    (UINT8 *sbicBaseAdrs, int regOffset, UINT clkPeriod,
    FUNCPTR sysScsiBusReset, int sysScsiResetArg, UINT sysScsiDmaMaxBytes,
    FUNCPTR sysScsiDmaStart, FUNCPTR sysScsiDmaAbort, int sysScsiDmaArg)
```

DESCRIPTION This library contains part of the I/O driver for the Western Digital WD33C93 family of SCSI-2 Bus Interface Controllers (SBIC). It is designed to work with **scsi2Lib**. The driver routines in this library depend on the SCSI-2 ANSI specification; for general driver routines and for overall SBIC documentation, see **wd33c93Lib**.

USER-CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. The only exception in this portion of the driver is *wd33c93CtrlCreateScsi2***( )**, which creates a controller structure.

INCLUDE FILES **wd33c93.h**, **wd33c93_2.h**

SEE ALSO **scsiLib**, **scsi2Lib**, **wd33c93Lib**, *VxWorks Programmer's Guide: I/O System*

wdbNetromPktDrv

NAME **wdbNetromPktDrv** – NETROM packet driver for the WDB agent

SYNOPSIS *wdbNetromPktDevInit***( )** – initialize a NETROM packet device for the WDB agent

```
void wdbNetromPktDevInit
    (WDB_NETROM_PKT_DEV *pPktDev, caddr_t dpBase, int width, int index,
    int numAccess, void (*stackRcv)(), int pollDelay)
```

DESCRIPTION This is a lightweight NETROM driver that interfaces with the WDB agent's UDP/IP interpreter. It allows the WDB agent to communicate with the host using the NETROM ROM emulator. It uses the emulator's read-only protocol for bi-directional communication. It requires that NetROM's udpsrcmode option is on.

wdbSlipPktDrv

NAME **wdbSlipPktDrv** – a serial line packetizer for the WDB agent

SYNOPSIS *wdbSlipPktDevInit***( )** – initialize a SLIP packet device for a WDB agent

```
void wdbSlipPktDevInit
    (WDB_SLIP_PKT_DEV *pPktDev, SIO_CHAN * pSioChan, void (*stackRcv)())
```

DESCRIPTION This is a lightweight SLIP driver that interfaces with the WDB agents UDP/IP interpreter. It is the lightweight equivalent of the VxWorks SLIP netif driver, and uses the same protocol to assemble serial characters into IP datagrams (namely the SLIP protocol). SLIP is a simple protocol that uses four token characters to delimit each packet:

– **FRAME_END** (0300)
– **FRAME_ESC** (0333)
– **FRAME_TRANS_END** (0334)
– **FRAME_TRANS_ESC** (0335)

The **END** character denotes the end of an IP packet. The **ESC** character is used with **TRANS_END** and **TRANS_ESC** to circumvent potential occurrences of **END** or **ESC** within a packet. If the **END** character is to be embedded, SLIP sends "**ESC TRANS_END**" to avoid confusion between a SLIP-specific **END** and actual data whose value is **END**. If the **ESC** character is to be embedded, then SLIP sends "**ESC TRANS_ESC**" to avoid confusion. (Note that the SLIP **ESC** is not the same as the ASCII ESC.)

On the receiving side of the connection, SLIP uses the opposite actions to decode the SLIP packets. Whenever an **END** character is received, SLIP assumes a full packet has been received and sends on.

This driver has an MTU of 1006 bytes. If the host is using a real SLIP driver with a smaller MTU, then you will need to lower the definition of **WDB_MTU** in **configAll.h** so that the host and target MTU match. If you are not using a SLIP driver on the host, but instead are using the target server's wdbserial backend to connect to the agent, then you do not need to worry about incompatabilities between the host and target MTUs.

wdbUlipPktDrv

NAME

wdbUlipPktDrv – WDB communication interface for the ULIP driver

SYNOPSIS

*wdbUlipPktDevInit***()** – initialize the WDB agent's communication functions for ULIP

```
void wdbUlipPktDevInit
    (WDB_ULIP_PKT_DEV * pDev, char * ulipDev, void (*stackRcv)())
```

DESCRIPTION

This is a lightweight ULIP driver that interfaces with the WDB agent's UDP/IP interpreter. It is the lightweight equivalent of the ULIP netif driver. This module provides a communication path which supports both a task mode and an external mode WDB agent.

wdbVioDrv

NAME

wdbVioDrv – virtual tty I/O driver for the WDB agent

SYNOPSIS

*wdbVioDrv***( )** – initialize the tty driver for a WDB agent

```
STATUS wdbVioDrv
    (char *name)
```

DESCRIPTION

This library provides a psuedo-tty driver for use with the WDB debug agent. I/O is performed on a virtual I/O device just like it is on a VxWorks serial device. The difference is that the data is not moved over a physical serial channel, but rather over a virtual channel created between the WDB debug agent and the Tornado host tools.

The driver is installed with *wdbVioDrv***( )**. Individual virtual I/O channels are created by opening the device (see **wdbVioDrv** for details). The virtual I/O channels are defined as follows:

Channel	Usage
0	Virtual console
1-0xffffff	Dynamically created on the host
>= 0x1000000	User defined

Once data is written to a virtual I/O channel on the target, it is sent to the host-based target server. The target server allows this data to be sent to another host tool, redirected to the "virtual console," or redirected to a file. For details see the *Tornado User's Guide.*

wdLib

NAME

wdLib – watchdog timer library

SYNOPSIS

*wdCreate***( )** – create a watchdog timer
*wdDelete***( )** – delete a watchdog timer
*wdStart***( )** – start a watchdog timer
*wdCancel***( )** – cancel a currently counting watchdog

```
WDOG_ID wdCreate
    (void)

STATUS wdDelete
    (WDOG_ID wdId)
```

```
STATUS wdStart
    (WDOG_ID wdId, int delay, FUNCPTR pRoutine, int parameter)

STATUS wdCancel
    (WDOG_ID wdId)
```

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

SYNOPSIS *wdShowInit***( )** – initialize the watchdog show facility
*wdShow***( )** – show information about a watchdog

```
void wdShowInit
    (void)

STATUS wdShow
    (WDOG_ID wdId)
```

DESCRIPTION This library provides routines to show watchdog statistics, such as watchdog activity, a watchdog routine, etc.

INCLUDE FILES **wdLib.h**

SEE ALSO **wdLib**, *VxWorks Programmer's Guide: Basic OS, Target Shell*, **windsh**, *Tornado User's Guide: Shell*

wvHostLib

NAME **wvHostLib** – host information library (WindView)

SYNOPSIS *wvHostInfoInit***( )** – initialize host connection information (WindView)
*wvHostInfoShow***( )** – show host connection information (WindView)

```
STATUS wvHostInfoInit
    (char * pIpAddress, ushort_t port)

STATUS wvHostInfoShow
    (void)
```

DESCRIPTION This library provides a means of communicating characteristics of the host to the target.

INCLUDE FILES **wvLib.h**

SEE ALSO **wvLib**, *WindView User's Guide*

1

wvLib

NAME

wvLib – event logging control library (WindView)

SYNOPSIS

*wvInstInit***( )** – initialize instrumentation (WindView)
*wvEvtLogEnable***( )** – start event logging (WindView)
*wvEvtLogDisable***( )** – stop event logging (WindView)
*wvObjInstModeSet***( )** – set mode for object instrumentation (WindView)
*wvObjInst***( )** – instrument objects (WindView)
*wvSigInst***( )** – instrument signals (WindView)
*wvEvent***( )** – log a user-defined event (WindView)
*wvEvtTaskInit***( )** – set parameters for the event task (WindView)
*wvOn***( )** – start event logging (WindView)
*wvOff***( )** – stop event logging (WindView)

```
void * wvInstInit
     (char * buffer, size_t bufferSize, int mode)

STATUS wvEvtLogEnable
     (int eventLevel)

STATUS wvEvtLogDisable
     (void)

STATUS wvObjInstModeSet
     (int mode)

STATUS wvObjInst
     (int objType, void * objId, int mode)

STATUS wvSigInst
     (int mode)

STATUS wvEvent
     (event_t usrEventId, char * buffer, size_t bufSize)

void wvEvtTaskInit
     (int stackSize, int priority)

void wvOn
     (int eventLevel)

void wvOff
     (void)
```

DESCRIPTION

This library provides initialization, event logging, and instrumentation routines for the instrumented kernel. If the event upload mode is set to **CONTINUOUS_MODE**, initialization consists of creating the event buffer and spawning the event buffer task,

tEvtTask. If the event upload mode is set to **POST_MORTEM_MODE**, the initialization consists of creating the event buffer.

With the *wvInstInit*( ) routine, the event buffer upload mode can be set to either continuous or post-mortem mode. In continuous mode, event logging is enabled and the contents of the event buffer are continuously transferred to the host for display by the GUI, or collection by **evtRecv( )**. In post-mortem mode, events are logged without the overhead of transferring the buffer contents to the host. Instead, the event buffer continuously overwrites itself until event logging is turned off, the system fails, or the system is rebooted.

The *wvInstInit*( ) routine also allows the user to specify the location and size of the event buffer. For example, it can be placed in memory that is not zero-cleared on reboot using post-mortem mode. The size of the buffer can be specified too.

Note that in continuous mode, the larger the buffer is, the less frequently **tEvtTask** runs (**tEvtTask** empties the event buffer), but the longer the data transfer takes. The most efficient size of the buffer depends on the type of application and the number of events it is generating. In post-mortem mode, the larger the buffer is, the more events can be stored.

The stack size and priority of **tEvtTask** can be modified by *wvEvtTaskInit*( ). Again, the best values depend on the type of application. If the priority of **tEvtTask** is too low or **tEvtTask** is dependent on any other task in the system (e.g., **tNetTask** if sockets are used to transfer the event stream between the target and the host), then the event buffer will not be emptied as quickly as it is being filled and event logging will turn itself off.

There are different modes of event logging that can be controlled by **wvLib** functionality. The user has the ability to turn on context switch events, task state transition events, and object status events. *wvEvtLogEnable*( ) provides a way to turn on event logging for any category. *wvEvtLogDisable*( ) is used to turn off event logging. User events can be generated with the *wvEvent*( ) routine in any mode.

The context switch event logging mode has the least amount of intrusion on the system, and provides context change information. The next event logging mode, task state transition events, allows the user to see the reason for a task state transition. For example, a *semTake*( ) that causes a task to pend on the semaphore will be logged, but one that does not cause a task status change will not be logged.

For objects that have been instrumented, object state event logging mode provides information about objects in the system (tasks, semaphores, message queues, and watchdog timers) when those objects have actions performed upon them, e.g., *semDelete*( ) and *wdCancel*( ).

Object status event logging is intrusive, so *wvObjInstModeSet*( ), *wvObjInst*( ), and *wvSigInst*( ) provide a way to choose what objects are to be instrumented.

Signals are instrumented at the object status level and can be controlled via the *wvSigInst*( ) routine. *wvObjInst*( ) provides the finest control of object status event logging. *wvObjInst*( ) allows all objects of a certain type or any particular object (for

example, sem1) to be instrumented. The instrumentation will appear when *wvEvtLogEnable***( )** is used to start object status event logging at some later time in the application's execution.

*wvObjInstModeSet***( )** provides a more general way to instrument objects at creation time. This helps the user to focus on what is of interest. The instrumented objects will appear when *wvEvtLogEnable***( )** is used to enable object status event logging at some later time in the application's execution.

INCLUDE FILES **wvLib.h**

SEE ALSO *WindView User's Guide*

---

wvTmrLib

NAME **wvTmrLib** – timer library (WindView)

SYNOPSIS *wvTmrRegister***( )** – register a timestamp timer (WindView)

```
void wvTmrRegister
     (UINTFUNCPTR wvTmrRtn, UINTFUNCPTR wvTmrLockRtn, FUNCPTR wvTmrEnable,
     FUNCPTR wvTmrDisable, FUNCPTR wvTmrConnect, UINTFUNCPTR wvTmrPeriod,
     UINTFUNCPTR wvTmrFreq)
```

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.

INCLUDE FILES **wvTmrLib**, *WindView User's Guide*

SEE ALSO **wvLib**

z8530Sio

NAME

z8530Sio – Z8530 SCC Serial Communications Controller driver

SYNOPSIS

*z8530DevInit***( )** – intialize a **Z8530_DUSART**
*z8530IntWr***( )** – handle a transmitter interrupt
*z8530IntRd***( )** – handle a reciever interrupt
*z8530IntEx***( )** – handle error interrupts
*z8530Int***( )** – handle all interrupts in one vector

```
void z8530DevInit
    (Z8530_DUSART * pDusart)

void z8530IntWr
    (Z8530_CHAN * pChan)

void z8530IntRd
    (Z8530_CHAN * pChan)

void z8530IntEx
    (Z8530_CHAN * pChan)

void z8530Int
    (Z8530_DUSART * pDusart)
```

DESCRIPTION

This is the driver for the Z8530 SCC (Serial Communications Controller). It uses the SCCs in asynchronous mode only.

USAGE

A **Z8530_DUSART** structure is used to describe the chip. This data structure contains two **Z8530_CHAN** structures which describe the chip's two serial channels. The BSP's *sysHwInit***( )** routine typically calls *sysSerialHwInit***( )** which initializes all the values in the **Z8530_DUSART** structure (except the **SIO_DRV_FUNCS**) before calling *z8530DevInit***( )**. The BSP's *sysHwInit2***( )** routine typically calls *sysSerialHwInit2***( )** which connects the chips interrupts via *intConnect***( )** (either the single interrupt z8530Int or the three interrupts z8530IntWr, z8530IntRd, and z8530IntEx).

INCLUDE FILES

drv/sio/z8530Sio.h

zbufLib

NAME

zbufLib – zbuf interface library

SYNOPSIS

*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

```
ZBUF_ID zbufCreate
    (void)

STATUS zbufDelete
    (ZBUF_ID zbufId)

ZBUF_SEG zbufInsert
    (ZBUF_ID zbufId1, ZBUF_SEG zbufSeg, int offset, ZBUF_ID zbufId2)

ZBUF_SEG zbufInsertBuf
    (ZBUF_ID zbufId, ZBUF_SEG zbufSeg, int offset, caddr_t buf, int len,
    VOIDFUNCPTR freeRtn, int freeArg)

ZBUF_SEG zbufInsertCopy
    (ZBUF_ID zbufId, ZBUF_SEG zbufSeg, int offset, caddr_t buf, int len)

int zbufExtractCopy
    (ZBUF_ID zbufId, ZBUF_SEG zbufSeg, int offset, caddr_t buf, int len)

ZBUF_SEG zbufCut
    (ZBUF_ID zbufId, ZBUF_SEG zbufSeg, int offset, int len)

ZBUF_ID zbufSplit
    (ZBUF_ID zbufId, ZBUF_SEG zbufSeg, int offset)

ZBUF_ID zbufDup
    (ZBUF_ID zbufId, ZBUF_SEG zbufSeg, int offset, int len)
```

```
int zbufLength
    (ZBUF_ID zbufId)

ZBUF_SEG zbufSegFind
    (ZBUF_ID zbufId, ZBUF_SEG zbufSeg, int * pOffset)

ZBUF_SEG zbufSegNext
    (ZBUF_ID zbufId, ZBUF_SEG zbufSeg)

ZBUF_SEG zbufSegPrev
    (ZBUF_ID zbufId, ZBUF_SEG zbufSeg)

caddr_t zbufSegData
    (ZBUF_ID zbufId, ZBUF_SEG zbufSeg)

int zbufSegLength
    (ZBUF_ID zbufId, ZBUF_SEG zbufSeg)
```

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, permitting the zbuf interface to be used with other buffering schemes eventually.

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*. *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 usually not the most efficient address formulation in themselves (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. With regard to the original zbuf data, therefore, 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_ID**s 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.

SEE ALSO

zbufSockLib, *VxWorks Programmer's Guide: Network*

zbufSockLib

NAME

zbufSockLib – zbuf socket interface library

SYNOPSIS

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

```
STATUS zbufSockLibInit
    (void)

int zbufSockSend
    (int s, ZBUF_ID zbufId, int zbufLen, int flags)

int zbufSockSendto
    (int s, ZBUF_ID zbufId, int zbufLen, int flags, struct sockaddr * to,
    int tolen)

int zbufSockBufSend
    (int s, char * buf, int bufLen, VOIDFUNCPTR freeRtn, int freeArg,
    int flags)

int zbufSockBufSendto
    (int s, char * buf, int bufLen, VOIDFUNCPTR freeRtn, int freeArg,
    int flags, struct sockaddr * to, int tolen)

ZBUF_ID zbufSockRecv
    (int s, int flags, int * pLen)

ZBUF_ID zbufSockRecvfrom
    (int s, int flags, int * pLen, struct sockaddr * from, int * pFromLen)
```

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.

SEE ALSO

zbufLib, **sockLib**, *VxWorks Programmer's Guide: Network*

2

Subroutines

a0( )

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

SYNOPSIS

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

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

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

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

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

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 parameter *addrlen* 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*.

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

acw( )

NAME *acw*( ) – return the contents of the **acw** register (i960)

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

DESCRIPTION This command extracts the contents of the **acw** register from the TCB of a specified task. If *taskId* is omitted or 0, the current default task is assumed.

RETURNS The contents of the **acw** register.

SEE ALSO **dbgArchLib**, *VxWorks Programmer's Guide: Target Shell*

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**

aioShow( )

NAME *aioShow*( ) – show AIO requests

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

DESCRIPTION This routine displays the outstanding AIO requests.

CAVEAT The *drvNum* parameter is not 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_cancel( )

NAME

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

SYNOPSIS

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

DESCRIPTION

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

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

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

RETURNS

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

ERRNO

EBADF

INCLUDE FILES

aio.h

SEE ALSO

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

aio_error( )

NAME

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

SYNOPSIS

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

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

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

ERRNO **EINVAL**

INCLUDE FILES **aio.h**

SEE ALSO **aioPxLib**

---

*aio_fsync***( )**

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

SYNOPSIS

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

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

The value of *op* is ignored. It currently has no meaning in VxWorks.

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

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

RETURNS OK if queued successfully, otherwise ERROR.

ERRNO **EINVAL**, **EBADF**

INCLUDE FILES **aio.h**

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

aio_read( )

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

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

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

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

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

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

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

RETURNS OK if the read queued successfully, otherwise ERROR.

ERRNO **EBADF**, **EINVAL**

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.

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*( )

arpAdd( )

NAME *arpAdd*( ) – add an entry to the system ARP table

SYNOPSIS
```
STATUS arpAdd
    (
    char *  host,   /* host name or IP address      */
    char *  eaddr,  /* Ethernet address             */
```

```
    int     flags   /* ARP flags                    */
    )
```


DESCRIPTION This routine adds a specified entry to the ARP table. *host* is a valid host name or Internet address. *eaddr* is the Ethernet address of the host and has the form "x:x:x:x:x:x" where x is a hexadecimal number between 0 and ff.

The *flags* parameter specifies the ARP flags for the entry; the following bits are settable:

ATF_PERM (0x04)
The **ATF_PERM** bit makes the ARP entry permanent. A permanent ARP entry does not time out as do normal ARP entries.

ATF_PUBL (0x08)
The **ATF_PUBL** bit causes the entry to be published (i.e., this system responds to ARP requests for this entry, even though it is not the host).

ATF_USETRAILERS (0x10)
The **ATF_USETRAILERS** bit indicates that trailer encapsulations can be sent to this host.

EXAMPLE The following call creates a permanent ARP table entry for the host with IP address 90.0.0.3 and Ethernet address 0:80:f9:1:2:3:

```
arpAdd ("90.0.0.3", "0:80:f9:1:2:3", 0x4)
```

The following call adds an entry to the ARP table for host "myHost", with an Ethernet address of 0:80:f9:1:2:4; no flags are set for this entry:

```
arpAdd ("myHost", "0:80:f9:1:2:4", 0)
```

RETURNS OK, or ERROR if unsuccessful.

ERRNO **S_arpLib_INVALID_ARGUMENT**

SEE ALSO **arpLib**

---

*arpDelete***( )**

NAME *arpDelete***( )** – delete an entry from the system ARP table

SYNOPSIS

```
STATUS arpDelete
    (
    char *  host  /* host name or IP address */
    )
```

DESCRIPTION This routine deletes an ARP table entry. *host* specifies the entry to delete and is a valid host name or Internet address.

EXAMPLE
```
arpDelete ("91.0.0.3")
arpDelete ("myHost")
```

RETURNS OK, or ERROR if unsuccessful.

ERRNO **S_arpLib_INVALID_ARGUMENT**

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**

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.

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
```

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*.

This routine is the POSIX re-entrant version of *asctime*( ).

INCLUDE FILES **time.h**

RETURNS The size of the created string.

SEE ALSO **ansiTime**

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**

---

ataDevCreate( )

NAME *ataDevCreate*( ) – create a device for a ATA / IDE disk

SYNOPSIS

```
BLK_DEV *ataDevCreate
    (
    int  ctrl,
    int  drive,
    int  nBlocks,
    int  blkOffset
    )
```

DESCRIPTION This routine creates a device for a specified ATA / IDE disk.

drive is a drive number for the hard drive; it must be 0 or 1.

The *nBlocks* parameter specifies the size of the device in blocks. If *nBlocks* is zero, the whole disk is used.

The *blkOffset* parameter specifies an offset, in blocks, from the start of the device to be used when writing or reading the hard disk. This offset is added to the block numbers passed by the file system during disk accesses. (VxWorks file systems always use block numbers beginning at zero for the start of a device.)

RETURNS A pointer to a block device structure (**BLK_DEV**) or NULL if memory cannot be allocated for the device structure.

SEE ALSO **ataDrv**, *dosFsMkfs*( ), *dosFsDevInit*( ), *rt11FsDevInit*( ), *rt11FsMkfs*( ), *rawFsDevInit*( )

ataDrv( )

NAME *ataDrv*( ) – initialize the ATA driver

SYNOPSIS

```
STATUS ataDrv
    (
    int   ctrl,        /* controller no.                        */
    int   drives,      /* number of drives                      */
    int   vector,      /* interrupt vector                      */
    int   level,       /* interrupt level                       */
    BOOL  configType,  /* configuration type                    */
    int   semTimeout,  /* timeout seconds for sync semaphore */
    int   wdgTimeout   /* timeout seconds for watch dog      */
    )
```

DESCRIPTION This routine initializes the ATA/IDE driver, sets up interrupt vectors, and initializes the ATA/IDE chip. It must be called exactly once, before any reads, writes, or calls to *ataDevCreate*( ). Normally, it is called by *usrRoot*( ) in **usrConfig.c**.

RETURNS OK, or ERROR if initialization fails.

SEE ALSO **ataDrv**, *ataDevCreate*( )

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**

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**

ataRawio( )

NAME *ataRawio*( ) – do raw I/O access

SYNOPSIS

```
STATUS ataRawio
    (
    int      ctrl,
    int      drive,
    ATA_RAW  *pAtaRaw
    )
```

DESCRIPTION This routine is called to perform raw I/O access.

drive is a drive number for the hard drive: it must be 0 or 1.

The *pAtaRaw* is a pointer to the structure **ATA_RAW** which is defined in **ataDrv.h**.

RETURNS OK, or ERROR if the parameters are not valid.

SEE ALSO **ataDrv**

ataShow( )

NAME *ataShow*( ) – show the ATA/IDE disk parameters

SYNOPSIS

```
STATUS ataShow
    (
    int  ctrl,
    int  drive
    )
```

DESCRIPTION This routine shows the ATA/IDE disk parameters. Its first argument is a controller number, 0 or 1; the second argument is a drive number, 0 or 1.

RETURNS OK, or ERROR if the parameters are invalid.

SEE ALSO **ataShow**

ataShowInit( )

NAME *ataShowInit*( ) – initialize the ATA / IDE disk driver show routine

SYNOPSIS
```
void ataShowInit (void)
```

DESCRIPTION This routine links the ATA / IDE disk driver show routine into the VxWorks system. The routine is included automatically by defining **INCLUDE_SHOW_ROUTINES** in **configAll.h**. No arguments are needed.

RETURNS N / A

SEE ALSO **ataShow**

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**

autopushAdd( )

NAME *autopushAdd*( ) – add a list of automatically pushed STREAMS modules (STREAMS Opt.)

SYNOPSIS

```
void autopushAdd
    (
    char *  arg  /* autopush information to be added for device */
    )
```

DESCRIPTION This routine sets up the autopush configuration for the device passed. The first parameter is the device name, and subsequent parameters are the modules to be pushed. If a clone device is given, the modules are pushed on all minor devices. Modules can be pushed on specific minor devices if a minor device is specified as the device parameter.

RETURNS N / A

SEE ALSO **autopushLib**

autopushDelete( )

NAME *autopushDelete*( ) – delete autopush information for a device (STREAMS Opt.)

SYNOPSIS

```
void autopushDelete
    (
    char  *deviceName  /* autopush information to be deleted for device */
    )
```

DESCRIPTION This routine removes autopush configuration information for the device passed. If the device name passed is a clone device, then autopush information for all minor devices is removed. If the device name is a minor device, then autopush information is removed only for that specific minor device.

RETURNS N / A

SEE ALSO **autopushLib**

autopushGet( )

NAME *autopushGet*( ) – get autopush information for a device (STREAMS Opt.)

SYNOPSIS

```
void autopushGet
    (
    char  *deviceName  /* device name to get autopush information */
    )
```

DESCRIPTION This routine displays the autopush configuration information for the device. The device name can be a clone device or a minor device. If the device is a clone device, autopush information is displayed for all minor devices. If the device is a minor device, autopush information is displayed only for the specific minor device.

RETURNS N / A

SEE ALSO **autopushLib**

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 breakpoint  */
                     /* 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*

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*( )

bcopyDoubles( )

NAME *bcopyDoubles*( ) – copy one buffer to another eight bytes at a time (SPARC)

SYNOPSIS

```
STATUS bcopyDoubles
    (
    void *  source,       /* 8-byte aligned source buffer      */
    void *  destination,  /* 8-byte aligned destination buffer */
    int     ndoubles      /* Number of 256-byte quantities     */
    )
```

DESCRIPTION This function copies the buffer *source* to the buffer *destination*, both of which must be 8-byte aligned. The copying is done eight bytes at a time. Note the count is the number of doubles, or the number of bytes divided by eight. The number of bytes copied will always be a multiple of 256.

RETURNS OK, if it runs to completion.

SEE ALSO **bALib**, *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: 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 *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***( )**

*bfillDoubles***( )**

NAME *bfillDoubles***( )** – fill a buffer with a specified eight-byte pattern (SPARC)

SYNOPSIS

```
STATUS bfillDoubles
    (
    void *  buffer,       /* 8-byte aligned buffer         */
    int     nbytes,       /* Multiple of 256 bytes         */
    ULONG   bits_63to32,  /* Upper 32 bits of fill pattern */
    ULONG   bits_31to0    /* Lower 32 bits of fill pattern */
    )
```

DESCRIPTION This routine copies a specified 8-byte pattern to the buffer, which must be 8-byte aligned. The filling is done eight bytes at a time. The number of bytes filled will be rounded up to a multiple of 256 bytes.

RETURNS OK, if it runs to completion.

SEE ALSO **bALib**, *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 dependent)       */
    int      task,    /* task for which to set breakpoint   */
                      /* 0 = set all tasks                  */
    int      count,   /* number of passes before hit        */
    BOOL     quiet    /* TRUE = don't print debugging info, */
                      /* FALSE = print debugging info       */
    )
```

SYNOPSIS (i386/i486)

```
STATUS bh
    (
    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        */
    int      type,    /* breakpoint type                    */
    INSTR *  addr0    /* 2nd addr for breakpoint registers  */
    )
```

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**

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**

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.

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**

bootpMsgSend( )

NAME *bootpMsgSend*( ) – send a BOOTP request message

SYNOPSIS
```
STATUS bootpMsgSend
    (
    char *            ifName,     /* network interface name       */
    struct in_addr *  pIpDest,    /* destination IP address       */
    int               port,       /* port number                  */
    BOOTP_MSG *       pBootpMsg,  /* pointer to BOOTP message     */
    u_int             timeOut     /* timeout in seconds           */
    )
```

DESCRIPTION

This routine sends a BOOTP message *pBootpMsg* through the network interface specified by *ifName*. *pIpDest* specifies the destination IP address, which, in most cases, will be the broadcast address (255.255.255.255); however, if it is desirable to send the message to a particular server, the server's IP address can be specified here. This server must reside on the same local network as the network interface *ifName*.

A non-zero value for *port* specifies an alternate BOOTP server port to use. A zero value means the default server port (67).

The calling application can fill in most of the fields in the BOOTP message for maximum flexibility. The only fields that the caller cannot specify are **bp_op**, **bp_xid**, and **bp_secs**. If **bp_hlen** is 0, *bootpMsgSend***( )** fills **bp_type** with 1 (Ethernet type), **bp_hlen** with 6, and **bp_chaddr** with the Ethernet address associated with *ifName*.

The *bootpMsgSend***( )** routine will retransmit the BOOTP message, if it gets no reply. The retransmission time increases exponentially but is bounded. *timeOut* specifies an ultimate timeout value in ticks. If no reply is received within this period, an error is returned. A value of zero specifies an infinite *timeOut* value.

NOTE: If **bp_ciaddr** is specified, the BOOTP server may assume that the client will respond to an ARP request.

RETURNS

OK, or ERROR if unsuccessful.

ERRNO

S_bootpLib_INVALID_ARGUMENT
S_bootpLib_NO_BROADCASTS
S_bootpLib_TIME_OUT

SEE ALSO

bootpLib, **bootLib**

*bootpParamsGet***( )**

NAME

*bootpParamsGet***( )** – retrieve boot parameters via BOOTP

SYNOPSIS

```
STATUS bootpParamsGet
    (
    char *  ifName,     /* network interface name            */
    int     port,       /* optional port number              */
    char *  pInetAddr,  /* client's IP address               */
    char *  pHostAddr,  /* host's IP address                 */
    char *  pBootFile,  /* boot file name                    */
    int *   pSizeFile,  /* size of boot file                 */
    int *   pSubnet,    /* optional pointer to subnet        */
    char *  pGateway,   /* optional pointer to subnet gateway */
```

```
    u_int   timeOut     /* timeout in ticks                    */
    )
```

DESCRIPTION

This routine transmits a BOOTP request message over the network interface associated with *ifName*. This interface must already be attached and initialized prior to calling this routine.

A non-zero value for *port* specifies an alternate BOOTP server port. A zero value means the default BOOTP server port (67).

pInetAddr is a string that holds the client's Internet address. On input, if *pInetAddr* contains a non-NULL string, it is interpreted as the client's Internet address and passed on to the BOOTP server in the *bp_ciaddr* field of the BOOTP message structure (**BOOT_MSG**). The server will use it as a lookup field into the BOOTP database. The client's Internet address is copied to *pInetAddr*, which should be of length **INET_ADDR_LEN** (18 bytes).

pHostAddr is a string that holds the host's IP address. On input, if *pHostAddr* contains a non-NULL string, it is interpreted as the host where the BOOTP message is to be sent. Note that this host must be local to the *pIf* network. On return, the host's IP address is copied to *pHostAddr*, which should be of length **INET_ADDR_LEN** (18 bytes).

On input, if *pBootFile* is a non-empty string, the contents are passed to the BOOTP server. On return, *pBootFile* returns the file name retrieved from BOOTP server. On input, *pSizeFile* specifies the buffer length of *pBootFile*. On output, *pSizeFile* contains the size of the copied file name (up to buffer maximum of 128 bytes).

pSubnet is a pointer to the subnet mask. If *pSubnet* is non-NULL, the library attempts to retrieve the subnet mask by sending a message with a vendor-type cookie, as described in RFC 1048. However, to obtain this parameter, the BOOTP server must support the vendor-specific options described in RFC 1048, and the subnet mask must be specified in the BOOTP server database. The subnet mask is returned in host byte order.

pGateway is a pointer to a gateway for the subnet. If *pGateway* is non-NULL, the library attempts to retrieve the subnet gateway by sending a message with a vendor-type cookie, as described in RFC 1048. However, to obtain this parameter, the BOOTP server must support the vendor-specific options described in RFC 1048, and the gateway must be specified in the BOOTP server database. If more than one gateway is specified, the first gateway listed will be returned, according to the assumed priority of RFC 1048. On return, the gateway address is copied to *pGateway*, which should be of length **INET_ADDR_LEN** (18 bytes).

timeOut specifies a timeout value in ticks. If no reply is received within this period, an error is returned. Specify zero for an infinite *timeout* value.

RETURNS

OK, or ERROR if unsuccessful.

SEE ALSO

bootpLib, **bootLib**, RFC 1048

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**

bpattach( )

NAME

bpattach( ) – publish the **bp** network interface and initialize the driver and device

SYNOPSIS

```
STATUS bpattach
    (
    int   unit,      /* backplane unit number              */
    char  *pAnchor,  /* bus pointer to bp anchor           */
    int   procNum,   /* processor number in backplane      */
    int   intType,   /* interrupt type: poll, bus, mailbox */
    int   intArg1,   /* as per interrupt type              */
    int   intArg2,   /* as per interrupt type              */
    int   intArg3    /* as per interrupt type              */
    )
```

DESCRIPTION

This routine attaches a **bp** interface to the network, if the interface exists. This routine makes the interface available by filling in the network interface record. The system will initialize the interface when it is ready to accept packets.

RETURN

OK or ERROR.

SEE ALSO

if_bp

bpInit( )

NAME

bpInit( ) – initialize the backplane anchor

SYNOPSIS

```
STATUS bpInit
    (
    char  *pAnchor,  /* backplane anchor address                        */
    char  *pMem,     /* start of backplane shared memory, NONE = alloc */
    int   memSize,   /* no. bytes in bp shared memory, 0 = 0x100000    */
    BOOL  tasOK      /* TRUE = hardware can do test-and-set            */
    )
```

DESCRIPTION

This routine initializes the backplane anchor. Typically, *pAnchor* and *pMem* both point to the same block of shared memory. If the first processor is dual-porting its memory, then, by convention, the anchor is at 0x600 (only 16 bytes are required) and the start of memory *pMem* is dynamically allocated by using the value NONE (-1). *memSize* should be at least 64 Kbytes. The *tasOK* parameter is provided for CPUs that do not support the test-and-set

instruction. If the system includes any test-and-set deficient CPUs, then all CPUs must use the software “test-and-set”.

RETURNS OK, or ERROR if data structures cannot be set up or memory is insufficient.

SEE ALSO **if_bp**

---

bpShow( )

NAME *bpShow*( ) – display information about the backplane network

SYNOPSIS

```
void bpShow
    (
    char  *bpName,  /* backplane interface name (NULL == "bp0") */
    BOOL  zero      /* TRUE = zap totals                        */
    )
```

DESCRIPTION This routine shows information about the different CPUs configured in the backplane network.

EXAMPLE

```
-> bpShow
Anchor at 0x800000
heartbeat = 705, header at 0x800010, free pkts = 237.
cpu int type    arg1       arg2       arg3    queued pkts rd index
--- -------- ---------- ---------- ---------- ----------- --------
 0  poll            0x0        0x0        0x0       0        27
 1  poll            0x0        0x0        0x0       0        11
 2  bus-int         0x3       0xc9        0x0       0        9
 3  mbox-2         0x2d     0x8000        0x0       0        1
input packets = 192     output packets = 164
output errors = 0       collisions = 0
value = 1 = 0x1
```

RETURNS N / A

SEE ALSO **if_bp**

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**

bzeroDoubles( )

NAME *bzeroDoubles*( ) – zero out a buffer eight bytes at a time (SPARC)

SYNOPSIS
```
STATUS bzeroDoubles
    (
    void *  buffer,  /* 8-byte aligned buffer */
    int     nbytes   /* multiple of 256 bytes */
    )
```

DESCRIPTION This routine fills the first *nbytes* characters of the specified buffer with 0, eight bytes at a time. The buffer address is assumed to be 8-byte aligned. The number of bytes will be rounded up to a multiple of 256 bytes.

RETURNS OK, if it runs to completion.

SEE ALSO **bALib**, *bzero*( )

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 */
    INSTR *  addr1  /* address for npc; 0 = instruction next to pc  */
    )
```

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

To execute, enter:

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

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

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

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

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

SEE ALSO **dbgLib**, *tr*( ), *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

cacheArchClearEntry( )

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

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 68K 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.

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

SEE ALSO **cacheArchLib**

---

*cacheArchLibInit***( )**

NAME *cacheArchLibInit***( )** – initialize the 68K 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 Motorola MC680x0 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.

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.

RETURNS OK.

SEE ALSO **cacheArchLib**

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**

*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**

cacheI960CxIC1kLoadNLock( )

NAME *cacheI960CxIC1kLoadNLock*( ) – load and lock I960Cx 1KB instruction cache (i960)

SYNOPSIS

```
void cacheI960CxIC1kLoadNLock (void *address)
```

DESCRIPTION This routine loads and locks the I960Cx 1KB instruction cache. The loaded address must be an address of a quad-word aligned block of memory. The instructions loaded into the cache can only be accessed by selected interrupts which vector to the addresses of these instructions. The load-and-lock mechanism selectively optimizes latency and throughput for interrupts.

RETURNS N / A

SEE ALSO **cacheI960CxALib**

cacheI960CxICDisable( )

NAME *cacheI960CxICDisable*( ) – disable the I960Cx instruction cache (i960)

SYNOPSIS
```
void cacheI960CxICDisable (void)
```

DESCRIPTION This routine disables the I960Cx instruction cache.

RETURNS N / A

SEE ALSO **cacheI960CxALib**

cacheI960CxICEnable( )

NAME *cacheI960CxICEnable*( ) – enable the I960Cx instruction cache (i960)

SYNOPSIS
```
void cacheI960CxICEnable ( void )
```

DESCRIPTION This routine enables the I960Cx instruction cache.

RETURNS N / A

SEE ALSO **cacheI960CxALib**

cacheI960CxICInvalidate( )

NAME *cacheI960CxICInvalidate*( ) – invalidate the I960Cx instruction cache (i960)

SYNOPSIS
```
void cacheI960CxICInvalidate ( void )
```

DESCRIPTION This routine invalidates the I960Cx instruction cache.

RETURNS N / A

SEE ALSO **cacheI960CxALib**

cacheI960CxICLoadNLock( )

NAME *cacheI960CxICLoadNLock*( ) – load and lock I960Cx 512-byte instruction cache (i960)

SYNOPSIS
```
void cacheI960CxICLoadNLock (void *address)
```

DESCRIPTION This routine loads and locks the I960Cx 512-byte instruction cache. The loaded address must be an address of a quad-word aligned block of memory. The instructions loaded into the cache can only be accessed by selected interrupts which vector to the addresses of these instructions. The load-and-lock mechanism selectively optimizes latency and throughput for interrupts.

RETURNS N / A

SEE ALSO **cacheI960CxALib**

cacheI960CxLibInit( )

NAME *cacheI960CxLibInit*( ) – initialize the I960Cx cache library (i960)

SYNOPSIS
```
STATUS cacheI960CxLibInit
    (
    CACHE_MODE  instMode,  /* instruction cache mode */
    CACHE_MODE  dataMode   /* data cache mode        */
    )
```

DESCRIPTION This routine initializes the function pointers for the I960Cx cache library. The board support package can select this cache library by calling this routine.

RETURNS OK.

SEE ALSO **cacheI960CxLib**

cacheI960JxDCCoherent( )

NAME *cacheI960JxDCCoherent*( ) – ensure data cache coherency (i960)

SYNOPSIS
```
void cacheI960JxDCCoherent ( void )
```

DESCRIPTION This routine ensures coherency by invalidating data cache on the I960Jx.

RETURNS N / A

SEE ALSO **cacheI960JxALib**

cacheI960JxDCDisable( )

NAME *cacheI960JxDCDisable*( ) – disable the I960Jx data cache (i960)

SYNOPSIS
```
void cacheI960JxDCDisable ( void )
```

DESCRIPTION This routine disables the I960Jx data cache.

RETURNS N / A

SEE ALSO **cacheI960JxALib**

cacheI960JxDCEnable( )

NAME *cacheI960JxDCEnable*( ) – enable the I960Jx data cache (i960)

SYNOPSIS
```
void cacheI960JxDCEnable ( void )
```

DESCRIPTION This routine enables the I960Jx data cache.

RETURNS N / A

SEE ALSO **cacheI960JxALib**

cacheI960JxDCFlush( )

NAME *cacheI960JxDCFlush*( ) – flush the I960Jx data cache (i960)

SYNOPSIS

```
void cacheI960JxDCFlush
    (
    sysDCDestAddr       /* destination address for cache sets */
    sysICSetsToStore    /* DC set #'s to be stored            */
    )
```

DESCRIPTION This routine flushes the I960Jx data cache.

RETURNS N / A

SEE ALSO **cacheI960JxALib**

cacheI960JxDCInvalidate( )

NAME *cacheI960JxDCInvalidate*( ) – invalidate the I960Jx data cache (i960)

SYNOPSIS

```
void cacheI960JxDCInvalidate ( void )
```

DESCRIPTION This routine invalidates the I960Jx data cache.

RETURNS N / A

SEE ALSO **cacheI960JxALib**

cacheI960JxDCStatusGet( )

NAME *cacheI960JxDCStatusGet*( ) – get the I960Jx data cache status (i960)

SYNOPSIS

```
void cacheI960JxDCStatusGet
    (
    sysDCStatusBuf    /* buffer to hold I-cache status */
    )
```

DESCRIPTION This routine gets the I960Jx data cache status.

RETURNS N / A

SEE ALSO **cacheI960JxALib**

cacheI960JxICDisable( )

NAME *cacheI960JxICDisable*( ) – disable the I960Jx instruction cache (i960)

SYNOPSIS
```
void cacheI960JxICDisable (void)
```

DESCRIPTION This routine disables the I960Jx instruction cache.

RETURNS N / A

SEE ALSO **cacheI960JxALib**

cacheI960JxICEnable( )

NAME *cacheI960JxICEnable*( ) – enable the I960Jx instruction cache (i960)

SYNOPSIS
```
void cacheI960JxICEnable ( void )
```

DESCRIPTION This routine enables the I960Jx instruction cache.

RETURNS N / A

SEE ALSO **cacheI960JxALib**

cacheI960JxICFlush( )

NAME *cacheI960JxICFlush*( ) – flush the I960Jx instruction cache (i960)

SYNOPSIS

```
void cacheI960JxICFlush
    (
    sysICDestAddr       /* destination address for cache sets */
    sysICSetsToStore    /* IC set #'s to be stored            */
    )
```

DESCRIPTION This routine flushes the I960Jx instruction cache.

RETURNS N / A

SEE ALSO **cacheI960JxALib**

cacheI960JxICInvalidate( )

NAME *cacheI960JxICInvalidate*( ) – invalidate the I960Jx instruction cache (i960)

SYNOPSIS

```
void cacheI960JxICInvalidate ( void )
```

DESCRIPTION This routine invalidates the I960Jx instruction cache.

RETURNS N / A

SEE ALSO **cacheI960JxALib**

cacheI960JxICLoadNLock( )

NAME *cacheI960JxICLoadNLock*( ) – load and lock the I960Jx instruction cache (i960)

SYNOPSIS

```
void cacheI960JxICLoadNLock
    (
    sysICCodeStart    /* starting address of code */
    sysICNoBlocks     /* # of blocks to lock      */
    )
```

DESCRIPTION This routine loads and locks the I960Jx instruction cache.

RETURNS N / A

SEE ALSO **cacheI960JxALib**

cacheI960JxICLockingStatusGet( )

NAME *cacheI960JxICLockingStatusGet*( ) – get the I960Jx I-cache locking status (i960)

SYNOPSIS
```
void cacheI960JxICLockingStatusGet
    (
    sysICStatusBuf    /* buffer to hold I-cache locking status */
    )
```

DESCRIPTION This routine gets the I960Jx instruction cache locking status.

RETURNS N / A

SEE ALSO **cacheI960JxALib**

cacheI960JxICStatusGet( )

NAME *cacheI960JxICStatusGet*( ) – get the I960Jx instruction cache status (i960)

SYNOPSIS
```
void cacheI960JxICStatusGet
    (
    sysICStatusBuf    /* buffer to hold I-cache status */
    )
```

DESCRIPTION This routine gets the I960Jx instruction cache status.

RETURNS N / A

SEE ALSO **cacheI960JxALib**

cacheI960JxLibInit( )

NAME *cacheI960JxLibInit*( ) – initialize the I960Jx cache library (i960)

SYNOPSIS

```
STATUS cacheI960JxLibInit
    (
    CACHE_MODE  instMode,  /* instruction cache mode */
    CACHE_MODE  dataMode   /* data cache mode        */
    )
```

DESCRIPTION This routine initializes the function pointers for the I960Jx cache library. The board support package can select this cache library by calling this routine.

RETURNS OK.

SEE ALSO **cacheI960JxLib**

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**

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**

cacheMicroSparcLibInit( )

NAME *cacheMicroSparcLibInit*( ) – initialize the microSPARC cache library

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

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

The only available cache mode is **CACHE_WRITETHROUGH**.

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

SEE ALSO **cacheMicroSparcLib**

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**

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**

cacheR3kDsize( )

NAME *cacheR3kDsize*( ) – return the size of the R3000 data cache

SYNOPSIS
```
ULONG cacheR3kDsize (void)
```

DESCRIPTION This routine returns the size of the R3000 data cache. Generally, this value should be placed into the value *cacheDCacheSize* for use by other routines.

RETURNS The size of the data cache in bytes.

SEE ALSO **cacheR3kALib**

cacheR3kIsize( )

NAME *cacheR3kIsize*( ) – return the size of the R3000 instruction cache

SYNOPSIS
```
ULONG cacheR3kIsize (void)
```

DESCRIPTION This routine returns the size of the R3000 instruction cache. Generally, this value should be placed into the value *cacheDCacheSize* for use by other routines.

RETURNS The size of the instruction cache in bytes.

SEE ALSO **cacheR3kALib**

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**

cacheR4kLibInit( )

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

SYNOPSIS
```
STATUS cacheR4kLibInit
    (
    CACHE_MODE  instMode,  /* instruction cache mode */
    CACHE_MODE  dataMode   /* data cache mode        */
    )
```

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**

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 Virtual address that maps to the physical address bits [31:0] argument, or NULL if it fails.

RETURNS N / A

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**

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**)

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
    (
    char  *name  /* new directory name */
    )
```

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

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

usrLib, *pwd***( )**, *VxWorks Programmer's Guide: Target Shell*

---

*cd2400HrdInit***( )**

NAME

*cd2400HrdInit***( )** – initialize the chip

SYNOPSIS

```
void cd2400HrdInit
    (
    CD2400_QUSART *  pQusart  /* chip to reset */
    )
```

DESCRIPTION

This routine initializes the chip and the four channels.

SEE ALSO

cd2400Sio

cd2400Int( )

NAME *cd2400Int*( ) – handle special status interrupts

SYNOPSIS
```
void cd2400Int
    (
    CD2400_CHAN *  pChan
    )
```

DESCRIPTION This routine handles special status interrupts from the MPCC.

SEE ALSO **cd2400Sio**

cd2400IntRx( )

NAME *cd2400IntRx*( ) – handle receiver interrupts

SYNOPSIS
```
void cd2400IntRx
    (
    CD2400_CHAN *  pChan
    )
```

DESCRIPTION This routine handles the interrupts for all channels for a Receive Data Interrupt.

SEE ALSO **cd2400Sio**

cd2400IntTx( )

NAME *cd2400IntTx*( ) – handle transmitter interrupts

SYNOPSIS
```
void cd2400IntTx
    (
    CD2400_CHAN *  pChan
    )
```

DESCRIPTION This routine handles transmitter interrupts from the MPCC.

SEE ALSO **cd2400Sio**

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). 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*

cisConfigregGet( )

NAME

cisConfigregGet( ) – get the PCMCIA configuration register

SYNOPSIS

```
STATUS cisConfigregGet
    (
    int  sock,    /* socket no.                */
    int  reg,     /* configuration register no. */
    int  *pValue  /* content of the register    */
    )
```

DESCRIPTION This routine gets that PCMCIA configuration register.

RETURNS OK, or ERROR if it cannot set a value on the PCMCIA chip.

SEE ALSO **cisLib**

cisConfigregSet( )

NAME *cisConfigregSet*( ) – set the PCMCIA configuration register

SYNOPSIS
```
STATUS cisConfigregSet
    (
    int  sock,  /* socket no.              */
    int  reg,   /* register no.            */
    int  value  /* content of the register */
    )
```

DESCRIPTION This routine sets the PCMCIA configuration register.

RETURNS OK, or ERROR if it cannot set a value on the PCMCIA chip.

SEE ALSO **cisLib**

cisFree( )

NAME *cisFree*( ) – free tuples from the linked list

SYNOPSIS
```
void cisFree
    (
    int  sock  /* socket no. */
    )
```

DESCRIPTION This routine free tuples from the linked list.

RETURNS N / A

SEE ALSO **cisLib**

cisGet( )

NAME *cisGet*( ) – get information from a PC card's CIS

SYNOPSIS

```
STATUS cisGet
    (
    int  sock  /* socket no. */
    )
```

DESCRIPTION This routine gets information from a PC card's CIS, configures the PC card, and allocates resources for the PC card.

RETURNS OK, or ERROR if it cannot get the CIS information, configure the PC card, or allocate resources.

SEE ALSO **cisLib**

cisShow( )

NAME *cisShow*( ) – show CIS information

SYNOPSIS

```
void cisShow
    (
    int  sock  /* socket no. */
    )
```

DESCRIPTION This routine shows CIS information.

RETURNS N / A

SEE ALSO **cisShow**

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 interupt. 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.

INCLUDE FILES **time.h**

RETURNS ERROR (-1).

SEE ALSO **ansiTime**

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.

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 sets the clock resolution in the POSIX timers data structures. It does not affect the system clock or auxiliary clocks. It should be called to inform POSIX timers of the new clock resolution if *sysClkRateSet*( ) has been called after this library has been initialized.

If *res* is non-NULL, the resolution to be set is stored in the location pointed to; otherwise, this routine has no effect.

NOTE: Non-POSIX.

RETURNS 0 (OK), or -1 (ERROR) if *clock_id* is invalid or the resolution is greater than 1 second.

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.

RETURNS OK, or ERROR if the call fails.

SEE ALSO **sockLib**

connectWithTimeout( )

NAME *connectWithTimeout*( ) – attempt a connection over a socket for 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 This routine performs the same function as *connect*( ); however, in addition, users can specify how long to continue attempting the new connection.

If the *timeVal* is a NULL pointer, this routine performs exactly like *connect*( ). If *timeVal* is not NULL, it will attempt to establish a new connection for the duration of the time specified in *timeVal*, after which it will report a time-out error if the connection is not established.

RETURNS OK, or ERROR if a connection cannot be established.

SEE ALSO **sockLib**, *connect*( )

connRtnSet( )

NAME *connRtnSet*( ) – set up connection routines for target-host communication (WindView)

SYNOPSIS

```
void connRtnSet
    (
    FUNCPTR      initRtn,     /* connection initialization routine */
    VOIDFUNCPTR  closeRtn,    /* connection close routine          */
    VOIDFUNCPTR  errorRtn,    /* connection error handling routine */
    VOIDFUNCPTR  dataXferRtn  /* connection data transfer routine  */
    )
```

DESCRIPTION This routine establishes four target-host communication routines; by default they are TCP socket routines. Users can replace these routines with their own communication routines.

Four routines are required: An initialization routine, a close connection routine, an error handler, and a routine that transfers the data from the event buffer to another location.

The data transfer routine must complete its job before the next data transfer cycle. If it fails to do so, a bandwidth exceeded condition occurs and event logging stops.

FUNCPTR and **VOIDFUNCPTR** are defined as follows,

```
typedef int                (*FUNCPTR) (...);
typedef void               (*VOIDFUNCPTR) (...);
```

RETURNS N / A

SEE ALSO **connLib**, **wvLib**

---

copy( )

NAME *copy*( ) – copy *in* (or stdin) to *out* (or stdout)

SYNOPSIS

```
STATUS copy
    (
    char  *in,  /* name of file to read  (if NULL assume stdin)  */
    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 an error occurs copying *in* to *out*.

SEE ALSO **usrLib**, *copyStreams*( ), *tyEOFSet*( ), *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 **usrLib**, *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**

cplusCallNewHandler( )

NAME *cplusCallNewHandler*( ) – call the allocation exception 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
```

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	Behavior
off	0	The function name is not demangled
terse	1	Only the function name is printed. The class name and parameter type list are omitted.
complete	2	When C++ function names are printed, the class name (if any) is prefixed and the function's parameter type list is appended.

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**

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
```

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 **INCLUDE_CPLUS** is defined in **configAll.h**, *cplusLibInit*( ) is called automatically from the root task, *usrRoot*( ), in **usrConfig.c**.

RETURNS OK or ERROR.

SEE ALSO **cplusLib**

cplusLibMinInit( )

NAME *cplusLibMinInit*( ) – initialize the minimal C++ library (C++)

SYNOPSIS
```
extern "C" STATUS cplusLibMinInit (void)
```

DESCRIPTION This routine initializes the C++ library without forcing unused C++ run-time support to be linked with the bootable VxWorks image. If **INCLUDE_CPLUS_MIN** is defined in **configAll.h**, *cplusLibMinInit*( ) 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 0.

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

cpmattach( )

NAME

cpmattach( ) – publish the **cpm** network interface and initialize the driver

SYNOPSIS

```
STATUS cpmattach
    (
    int             unit,       /* unit number                              */
    SCC *           pScc,       /* address of SCC parameter RAM             */
    SCC_REG *       pSccReg,    /* address of SCC registers                 */
    VOIDFUNCPTR *   ivec,       /* interrupt vector offset                  */
    SCC_BUF *       txBdBase,   /* transmit buffer descriptor base address  */
    SCC_BUF *       rxBdBase,   /* receive buffer descriptor base address   */
    int             txBdNum,    /* number of transmit buffer descriptors    */
    int             rxBdNum,    /* number of receive buffer descriptors     */
    UINT8 *         bufBase     /* address of memory pool; NONE = malloc it */
    )
```

DESCRIPTION

The routine publishes the **cpm** interface by filling in a network Interface Data Record (IDR) and adding this record to the system's interface list.

The SCC shares a region of memory with the driver. The caller of this routine can specify the address of a shared, non-cacheable memory region with *bufBase*. If this parameter is **NONE**, the driver obtains this memory region by calling *cacheDmaMalloc*( ). Non-cacheable memory space is important for cases where the SCC is operating with a processor that has a data cache.

Once non-cacheable memory is obtained, this routine divides up the memory between the various buffer descriptors (BDs). The number of BDs can be specified by *txBdNum* and *rxBdNum*, or if NULL, a default value of 32 BDs will be used. Additional buffers are reserved as receive loaner buffers. The number of loaner buffers is the lesser of *rxBdNum* and a default value of 16.

The user must specify the location of the transmit and receive BDs in the CPU's dual-ported RAM. *txBdBase* and *rxBdBase* give the base address of the BD rings. Each BD uses 8 bytes. Care must be taken so that the specified locations for Ethernet BDs do not conflict with other dual-ported RAM structures.

Up to four individual device units are supported by this driver. Device units may reside on different processor chips, or may be on different SCCs within a single CPU.

Before this routine returns, it calls *cpmReset*( ) to put the Ethernet controller in a quiescent state, and connects up the interrupt vector *ivec*.

RETURNS

OK or ERROR.

SEE ALSO

if_cpm, **ifLib**, *Motorola MC68360 User's Manual*, *Motorola MPC821 & MPC860 User's Manual*

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*

---

*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));
```

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**) (MC680x0)

SYNOPSIS
```
int d0
    (
    int  taskId  /* task ID, 0 means default task */
    )
```

DESCRIPTION This command extracts the contents of register **d0** from the TCB of a specified task. If *taskId* is omitted or zero, the last task referenced is assumed.

Similar routines are provided for all data registers (**d0** – **d7**): *d0***( )** – *d7***( )**.

RETURNS The contents of register **d0** (or the requested register).

SEE ALSO **dbgArchLib**, *VxWorks Programmer's Guide: Target Shell*

*dbgBpTypeBind***( )**

NAME *dbgBpTypeBind***( )** – bind a breakpoint handler to a breakpoint type (MIPS R3000, R4000)

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 function *dbgBreakpoint***( )** (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 the 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[,task[,eventNo[,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. It also enables the shell abort function, **CTRL+C**.

NOTE The debugging package should be initialized before any debugging routines are used. If **INCLUDE_DEBUG** is defined in **configAll.h**, *dbgInit*( ) is called by the root task, *usrRoot*( ), in **usrConfig.c**.

RETURNS OK, always.

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

dcattach( )

NAME *dcattach*( ) – publish the **dc** network interface

SYNOPSIS

```
STATUS dcattach
    (
    int     unit,        /* unit number                                  */
    ULONG   devAdrs,     /* LANCE I/O address                            */
    int     ivec,        /* interrupt vector                             */
    int     ilevel,      /* interrupt level                              */
    char *  memAdrs,     /* address of memory pool (-1 = malloc it)      */
    ULONG   memSize,     /* only used if memory pool is NOT malloc()'d   */
    int     memWidth,    /* byte-width of data (-1 = any width)          */
    ULONG   pciMemBase,  /* main memory base as seen from PCI bus        */
    int     dcOpMode     /* mode of operation                            */
    )
```

DESCRIPTION This routine publishes the **dc** interface by filling in a network interface record and adding this record to the system list. This routine also initializes the driver and the device to the operational state. Parameters:

unit	device unit to initialize.
devAdrs	I/O address base of the device.
ivec	interrupt vector associated with the device interrupt.
ilevel	level of the interrupt which the device will use.
memAdrs	location of memory that will be shared between the driver and device. The value **NONE** is used to indicate that the driver should obtain the memory.
memSize	valid only if the *memAdrs* parameter is not set to **NONE**, in which case *memSize* indicates the size of the provided memory region.
memWidth	memory-pool data-port width (in bytes); if **NONE**, any data width is used.
pciMemBase	main memory base as seen from the PCI bus.
dcOpMode	mode in which the device will operate.

BUGS To zero out LANCE data structures, this routine uses *bzero*( ), which ignores the *memWidth* specification and uses any size data access to write to memory.

RETURNS OK or ERROR.

SEE ALSO **if_dc**

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*

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**

diskFormat( )

NAME *diskFormat*( ) – format a disk

SYNOPSIS
```
STATUS diskFormat
    (
    char  *devName  /* 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*( ) or *rt11FsDevInit*( ).

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 **usrLib**, **dosFsLib**, **rt11FsLib**, *VxWorks Programmer's Guide: Target Shell*

diskInit( )

NAME *diskInit*( ) – initialize a file system on a block device

SYNOPSIS
```
STATUS diskInit
    (
    char  *devName  /* name of the device to initialize */
    )
```

DESCRIPTION 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 *dosFsDevInit*( ) or *rt11FsDevInit*( ).

This command calls *ioctl*( ) to perform the **FIODISKINIT** function.

EXAMPLE
```
-> diskInit "/fd0/"
```

RETURNS OK, or ERROR if the device cannot be opened or initialized.

SEE ALSO **usrLib**, **dosFsLib**, **rt11FsLib**, *VxWorks Programmer's Guide: Target Shell*

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*.

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**

dlpiInit( )

NAME *dlpiInit*( ) – initialize the DLPI driver

SYNOPSIS
```
STATUS dlpiInit (void)
```

DESCRIPTION This routine installs the STREAMS DLPI driver into the VxWorks I/O subsystem.

RETURNS N/A

SEE ALSO **dlpiLib**

dosFsConfigGet( )

NAME *dosFsConfigGet*( ) – obtain dosFs volume configuration values

SYNOPSIS
```
STATUS dosFsConfigGet
    (
    DOS_VOL_DESC    *vdptr,   /* ptr to volume descriptor        */
    DOS_VOL_CONFIG  *pConfig  /* ptr to config structure to fill */
    )
```

DESCRIPTION This routine obtains the current configuration values for a dosFs disk volume. The data is obtained from the dosFs volume descriptor specified by *vdptr*. No physical I/O to the device takes place.

The configuration data is placed into a **DOS_VOL_CONFIG** structure, whose address is *pConfig*. This structure must be allocated before calling *dosFsConfigGet*( ).

One use for this routine is to obtain the configuration data from a known good disk, to be used to initialize a new disk (using *dosFsDevInit*( )).

The volume is not locked while the data is being read from the volume descriptor, so it is conceivable that another task may modify the configuration information while this routine is executing.

RETURNS OK or ERROR.

SEE ALSO **dosFsLib**

dosFsConfigInit( )

NAME *dosFsConfigInit*( ) – initialize dosFs volume configuration structure

SYNOPSIS

```
STATUS dosFsConfigInit
    (
    DOS_VOL_CONFIG  *pConfig,     /* pointer to volume config structure */
    char            mediaByte,    /* media descriptor byte              */
    UINT8           secPerClust,  /* sectors per cluster                */
    short           nResrvd,      /* number of reserved sectors         */
    char            nFats,        /* number of FAT copies               */
    UINT16          secPerFat,    /* number of sectors per FAT copy     */
    short           maxRootEnts,  /* max number of entries in root dir  */
    UINT            nHidden,      /* number of hidden sectors           */
    UINT            options       /* volume options                     */
    )
```

DESCRIPTION This routine initializes a dosFs volume configuration structure (**DOS_VOL_CONFIG**). This structure is used by the *dosFsDevInit*( ) routine to specify the file system configuration for the disk.

The **DOS_VOL_CONFIG** structure must have been allocated prior to calling this routine. Its address is specified by *pConfig*. The specified configuration variables are placed into their respective fields in the structure.

This routine is provided only to allow convenient initialization of the **DOS_VOL_CONFIG** structure (particularly from the VxWorks shell). A structure which is properly initialized by other means may be used equally well by *dosFsDevInit*( ).

RETURNS OK, or ERROR if there is an invalid parameter or *pConfig* is NULL.

SEE ALSO **dosFsLib**, *dosFsDevInit*( )

dosFsConfigShow( )

NAME *dosFsConfigShow*( ) – display dosFs volume configuration data

SYNOPSIS

```
STATUS dosFsConfigShow
    (
    char  *devName  /* name of device */
    )
```

DESCRIPTION This routine obtains the dosFs volume configuration for the named device, formats the data, and displays it on the standard output. The information which is displayed is that which is contained in a **DOS_VOL_CONFIG** structure, along with other configuration values (for example, from the **BLK_DEV** structure which describes the device).

If no device name is specified, the current default device is described.

RETURNS OK or ERROR.

SEE ALSO **dosFsLib**

---

*dosFsDateSet***( )**

NAME *dosFsDateSet***( )** – set the dosFs file system date

SYNOPSIS

```
STATUS dosFsDateSet
    (
    int  year,   /* year (1980...2099) */
    int  month,  /* month (1...12)     */
    int  day     /* day (1...31)       */
    )
```

DESCRIPTION This routine sets the date for the dosFs file system, which remains in effect until changed. All files created or modified are assigned this date in their directory entries.

NOTE: No automatic incrementing of the date is performed; each new date must be set with a call to this routine.

RETURNS OK, or ERROR if the date is invalid.

SEE ALSO **dosFsLib**, *dosFsTimeSet***( )**, *dosFsDateTimeInstall***( )**

dosFsDateTimeInstall( )

NAME

dosFsDateTimeInstall( ) – install a user-supplied date / time function

SYNOPSIS

```
void dosFsDateTimeInstall
    (
    FUNCPTR  pDateTimeFunc  /* pointer to user-supplied function */
    )
```

DESCRIPTION

This routine installs a user-supplied function to provide the current date and time. Once such a function is installed, **dosFsLib** will call it when necessary to obtain the date and time. Otherwise, the date and time most recently set by *dosFsDateSet*( ) and *dosFsTimeSet*( ) are used.

The user-supplied routine must take exactly one input parameter, the address of a **DOS_DATE_TIME** structure (defined in **dosFsLib.h**). The user routine should update the necessary fields in this structure and then return. Any fields which are not changed by the user routine will retain their previous value.

RETURNS

N / A

SEE ALSO

dosFsLib

dosFsDevInit( )

NAME

dosFsDevInit( ) – associate a block device with dosFs file system functions

SYNOPSIS

```
DOS_VOL_DESC *dosFsDevInit
    (
    char            *devName,  /* device name                    */
    BLK_DEV         *pBlkDev,  /* pointer to block device struct */
    DOS_VOL_CONFIG  *pConfig   /* pointer to volume config data  */
    )
```

DESCRIPTION

This routine takes a block device structure (**BLK_DEV**) created by a device driver 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 *pBlkDev* parameter is the address of the **BLK_DEV** structure which describes this device.

This routine associates the name *devName* with the 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 dosFs library during *dosFsInit***( )**. (The driver number is placed in the global variable **dosFsDrvNum**.)

The **BLK_DEV** structure contains configuration data describing the device and the addresses of five routines which will be called to read sectors, write sectors, reset the device, check device status, and perform other control functions (*ioctl***( )**). These routines will not be called until they are required by subsequent I/O operations.

The *pConfig* parameter is the address of a **DOS_VOL_CONFIG** structure. This structure must have been previously initialized with the specific dosFs configuration data to be used for this volume. This structure may be easily initialized using *dosFsConfigInit***( )**.

If the device being initialized already has a valid dosFs (MS-DOS) file system on it, the *pConfig* parameter may be NULL. In this case, the volume will be mounted and the configuration data will be read from the boot sector of the disk. (If *pConfig* is NULL, both change-no-warn and auto-sync options are initially disabled. These can be enabled using the *dosFsVolOptionsSet***( )** routine.)

This routine allocates and initializes a volume descriptor (**DOS_VOL_DESC**) for the device. It returns a pointer to **DOS_VOL_DESC**.

RETURNS A pointer to the volume descriptor **DOS_VOL_DESC**, or NULL if there is an error.

SEE ALSO **dosFsLib**, *dosFsMkfs***( )**

---

*dosFsDevInitOptionsSet***( )**

NAME *dosFsDevInitOptionsSet***( )** – specify volume options for *dosFsDevInit***( )**

SYNOPSIS

```
STATUS dosFsDevInitOptionsSet
    (
    UINT  options  /* options for future dosFsDevInit() calls */
    )
```

DESCRIPTION This routine allows volume options to be set that will be enabled by subsequent calls to *dosFsDevInit***( )** that do not explicitly supply configuration information in a **DOS_VOL_CONFIG** structure. This is normally done when mounting a disk which has already been initialized with file system data. The value of *options* will be used for all volumes that are initialized by *dosFsDevInit***( )**, unless a specific configuration is given.

The only volume options which may be specified in this call are those which are not tied to the actual data on the disk. Specifically, you may not specify the long file name option in this call; if a disk using that option is mounted, that will be automatically detected. If you specify such an unsettable option during this call it will be ignored; all valid option bits will still be accepted and applied during subsequent *dosFsDevInit***( )** calls.

For example, to use *dosFsDevInit***( )** to initialize a volume with the auto-sync and filesystem export options, do the following:

```
status = dosFsDevInitOptionsSet (DOS_OPT_AUTOSYNC | DOS_OPT_EXPORT);
if (status != OK)
    return (ERROR);
vdptr = dosFsDevInit ("DEV1:", pBlkDev, NULL);
                          /* note NULL pointer for DOS_VOL_CONFIG */
```

RETURNS OK, or ERROR if *options* is invalid.

SEE ALSO **dosFsLib**, *dosFsDevInit***( )**, *dosFsVolOptionsSet***( )**

*dosFsInit***( )**

NAME *dosFsInit***( )** – prepare to use the dosFs library

SYNOPSIS

```
STATUS dosFsInit
    (
    int  maxFiles  /* max no. of simultaneously open files */
    )
```

DESCRIPTION This routine initializes the dosFs library. It must be called exactly once, before any other routine in the library. The argument specifies the number of dosFs files that may be open at once. This routine installs **dosFsLib** as a driver in the I/O system driver table, allocates and sets up the necessary memory structures, and initializes semaphores. The driver number assigned to **dosFsLib** is placed in the global variable **dosFsDrvNum**.

To enable this initialization, define **INCLUDE_DOSFS** in **configAll.h**; *dosFsInit***( )** will then be called from the root task, *usrRoot***( )**, in **usrConfig.c**.

RETURNS OK or ERROR.

SEE ALSO **dosFsLib**

dosFsMkfs( )

NAME *dosFsMkfs*( ) – initialize a device and create a dosFs file system

SYNOPSIS

```
DOS_VOL_DESC *dosFsMkfs
    (
    char     *volName,  /* volume name to use             */
    BLK_DEV  *pBlkDev   /* pointer to block device struct */
    )
```

DESCRIPTION This routine provides a quick method of creating a dosFs file system on a device. It is used instead of the two-step procedure of calling *dosFsDevInit*( ) followed by an *ioctl*( ) call with an **FIODISKINIT** function code.

This call uses default values for various dosFs configuration parameters (i.e., those found in the volume configuration structure, **DOS_VOL_CONFIG**). The values used are:

2	sectors per cluster (see below)
1	reserved sector
2	FAT copies
112	root directory entries
0xF0	media byte value
0	hidden sectors

The volume options (auto-sync mode, change-no-warn mode, and long filenames) that are enabled by this routine can be set in advance using *dosFsMkfsOptionsSet*( ). By default, none of these options is enabled for disks initialized by *dosFsMkfs*( ).

If initializing a large disk, it is quite possible that the entire disk area cannot be described by the maximum 64K clusters if only two sectors are contained in each cluster. In such a situation, *dosFsMkfs*( ) will automatically increase the number of sectors per cluster to a number which will allow the entire disk area to be described in 64K clusters.

The number of sectors per FAT copy is set to the minimum number of sectors which will contain sufficient FAT entries for the entire block device.

RETURNS A pointer to a dosFs volume descriptor, or NULL if there is an error.

SEE ALSO **dosFsLib**, *dosFsDevInit*( )

dosFsMkfsOptionsSet( )

NAME

dosFsMkfsOptionsSet( ) – specify volume options for *dosFsMkfs*( )

SYNOPSIS

```
STATUS dosFsMkfsOptionsSet
    (
    UINT  options  /* options for future dosFsMkfs() calls */
    )
```

DESCRIPTION

This routine allows volume options to be set that will be enabled by subsequent calls to *dosFsMkfs*( ). The value of *options* will be used for all volumes initialized by *dosFsMkfs*( ).

For example, to use *dosFsMkfs*( ) to initialize a volume with the auto-sync and long filename options, do the following:

```
status = dosFsMkfsOptionsSet (DOS_OPT_AUTOSYNC | DOS_OPT_LONGNAMES);
if (status != OK)
    return (ERROR);
vdptr = dosFsMkfs ("DEV1:", pBlkDev);
```

RETURNS

OK, or ERROR if *options* is invalid.

SEE ALSO

dosFsLib, *dosFsMkfs*( ), *dosFsVolOptionsSet*( )

dosFsModeChange( )

NAME

dosFsModeChange( ) – modify the mode of a dosFs volume

SYNOPSIS

```
void dosFsModeChange
    (
    DOS_VOL_DESC  *vdptr,  /* pointer to volume descriptor    */
    int           newMode  /* O_RDONLY/O_WRONLY/O_RDWR (both) */
    )
```

DESCRIPTION

This routine sets the volume's mode to *newMode*. The mode is actually kept in **bd_mode** fields of the the **BLK_DEV** structure, so that it may also be used by the device driver. Changing that field directly has the same result as calling this routine. The mode field should be updated whenever the read and write capabilities are determined, usually after a ready change. See the manual entry for *dosFsReadyChange*( ).

The driver's device initialization routine should initially set the mode field to **O_RDWR** (i.e., both **O_RDONLY** and **O_WRONLY**).

RETURNS N / A

SEE ALSO **dosFsLib**, *dosFsReadyChange***( )**

*dosFsReadyChange***( )**

NAME *dosFsReadyChange***( )** – notify dosFs of a change in ready status

SYNOPSIS

```
void dosFsReadyChange
    (
    DOS_VOL_DESC  *vdptr  /* pointer to volume descriptor */
    )
```

DESCRIPTION This routine sets the volume descriptor's state to **DOS_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 is called, the next attempt to use the volume will result in an attempted remount.

This routine may also be invoked by calling *ioctl***( )** with the function **FIODISKCHANGE**.

Setting the **bd_readyChanged** field to TRUE in the **BLK_DEV** structure that describes this device will have the same result as calling this routine.

RETURNS N / A

SEE ALSO **dosFsLib**

*dosFsTimeSet***( )**

NAME *dosFsTimeSet***( )** – set the dosFs file system time

SYNOPSIS

```
STATUS dosFsTimeSet
    (
    int  hour,    /* 0 to 23 */
    int  minute,  /* 0 to 59 */
    int  second   /* 0 to 59 */
    )
```

DESCRIPTION This routine sets the time for the dosFs file system, which remains in effect until changed. All files created or modified are assigned this time in their directory entries.

NOTE: No automatic incrementing of the time is performed; each new time must be set with a call to this routine.

RETURNS OK, or ERROR if the time is invalid.

SEE ALSO **dosFsLib**, *dosFsDateSet***( )**, *dosFsDateTimeInstall***( )**

*dosFsVolOptionsGet***( )**

NAME *dosFsVolOptionsGet***( )** – get current dosFs volume options

SYNOPSIS

```
STATUS dosFsVolOptionsGet
    (
    DOS_VOL_DESC *  vdptr,    /* ptr to volume descriptor           */
    UINT *          pOptions  /* where to put current options value */
    )
```

DESCRIPTION This routine obtains the current options for a specified dosFs volume and stores them in the field pointed to by *pOptions*.

RETURNS OK, always.

SEE ALSO **dosFsLib**, *dosFsVolOptionsSet***( )**

*dosFsVolOptionsSet***( )**

NAME *dosFsVolOptionsSet***( )** – set dosFs volume options

SYNOPSIS

```
STATUS dosFsVolOptionsSet
    (
    DOS_VOL_DESC *  vdptr,   /* ptr to volume descriptor */
    UINT            options  /* new options for volume   */
    )
```

DESCRIPTION This routine sets the volume options for an already-initialized dosFs device. Only the following options can be changed (enabled or disabled) dynamically:

DOS_OPT_CHANGENOWARN (0x1)
DOS_OPT_AUTOSYNC (0x2)

The **DOS_OPT_CHANGENOWARN** option may be enabled only for removable volumes (i.e., the **bd_removable** field in the **BLK_DEV** structure for the device must be set to TRUE). If specified for a non-removable volume, it is ignored. When successfully set, the **DOS_OPT_CHANGENOWARN** option also enables the **DOS_OPT_AUTOSYNC** option.

It is recommended that the current volume options be obtained by calling *dosFsVolOptionsGet***( )**, the desired option bits modified, and then the options set using *dosFsVolOptionsSet***( )**.

RETURNS OK, or ERROR if *options* is invalid or an attempt is made to change an option that is not dynamically changeable.

SEE ALSO **dosFsLib**, *dosFsDevInitOptionsSet***( )**, *dosFsMkfsOptionsSet***( )**, *dosFsVolOptionsGet***( )**

---

*dosFsVolUnmount***( )**

NAME *dosFsVolUnmount***( )** – unmount a dosFs volume

SYNOPSIS

```
STATUS dosFsVolUnmount
    (
    DOS_VOL_DESC  *vdptr  /* pointer to volume descriptor */
    )
```

DESCRIPTION This routine is called when I/O operations on a volume are to be discontinued. This is the preferred action prior to changing a removable disk.

All buffered data for the volume is written to the device (if possible, with no error returned if data cannot be written), any open file descriptors are marked as obsolete, and the volume is marked as not currently mounted. When a subsequent I/O operation is initiated on the disk (e.g., during the next *open***( )**), the volume will be remounted automatically.

Once file descriptors have been marked as obsolete, any attempt to use them for file operations will return an error. (An obsolete file descriptor may be freed by using *close***( )**. The call to *close***( )** will return an error, but the descriptor will in fact be freed.) File descriptors obtained by opening the entire volume (in raw mode) are not marked as obsolete.

This routine may also be invoked by calling *ioctl***( )** with the **FIOUNMOUNT** function code.

This routine must not be called from interrupt level.

RETURNS OK, or ERROR if the volume was not mounted.

SEE ALSO **dosFsLib**, *dosFsReadyChange***( )**

e( )

NAME e( ) – set or display eventpoints (WindView)

SYNOPSIS

```
STATUS e
    (
    INSTR *  addr,          /* where to set eventpoint, or          */
                            /* 0 means display all eventpoints      */
    event_t  eventId,       /* event ID                             */
    int      taskNameOrId,  /* task affected; 0 means all tasks     */
    FUNCPTR  evtRtn,        /* function to be invoked;              */
                            /* NULL means no function is invoked    */
    int      arg            /* argument to be passed to <evtRtn>    */
    )
```

DESCRIPTION This routine sets “eventpoints"—that is, breakpoint-like instrumentation markers that can be inserted in code to generate and log an event for use with WindView. Event logging must be enabled with *wvEvtLogEnable*( ) for the eventpoint to be logged.

eventId selects the evenpoint number that will be logged: it is in the user event ID range (0-25536).

If *addr* is NULL, then all eventpoints and breakpoints are displayed. If *taskNameOrId* is 0, then this event is logged in all tasks. The *evtRtn* routine is called when this eventpoint is hit. If *evtRtn* returns OK, then the eventpoint is logged; otherwise, it is ignored. If *evtRtn* is a NULL pointer, then the eventpoint is always logged.

Eventpoints are exactly like breakpoints (which are set with the *b*( ) command) except in how the system responds when the eventpoint is hit. An eventpoint typically records an event and continues immediately (if *evtRtn* is supplied, this behavior may be different). Eventpoints cannot be used at interrupt level and cannot be called from an unbreakable task.

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*( )

EBufferClean( )

NAME *EBufferClean*( ) – release dynamic memory in an extended buffer

SYNOPSIS

```
void    EBufferClean
    (
    EBUFFER_T *  ebuffp  /* extended buffer */
    );
```

DESCRIPTION This routine releases any dynamic memory attached to *ebuffp*

RETURNS N / A

SEE ALSO **snmpEbufLib**

EBufferClone( )

NAME *EBufferClone*( ) – make a copy of an extended buffer

SYNOPSIS

```
int    EBufferClone
    (
    EBUFFER_T *  srcp,  /* source buffer      */
    EBUFFER_T *  dstp   /* destination buffer */
    );
```

DESCRIPTION This routine creates a copy of an extended buffer, allocating space from the dynamic pool. Parameter *srcp* is the old buffer, and *dstp* the new.

RETURNS 0 if successful, otherwise -1.

SEE ALSO **snmpEbufLib**

EBufferInitialize( )

NAME *EBufferInitialize*( ) – place an extended buffer in a known state

SYNOPSIS
```
void    EBufferInitialize
    (
    EBUFFER_T *  ebuffp  /* extended buffer */
    );
```

DESCRIPTION This routine places the buffer-control block in a known state. The buffer is not ready to accept data, but may be safely handled by the extended-buffer routines.

RETURNS N / A

SEE ALSO **snmpEbufLib**

EBufferNext( )

NAME *EBufferNext*( ) – return a pointer to the next unused byte of the buffer memory

SYNOPSIS
```
OCTET_T *   EBufferNext
    (
    EBUFFER_T *  ebuffp  /* extended buffer */
    );
```

DESCRIPTION This routine returns a pointer to the next unused byte in the buffer memory. The pointer is valid only if there are unused bytes remaining in the buffer.

RETURNS a pointer to the first unused byte.

SEE ALSO **snmpEbufLib**

EBufferPreLoad( )

NAME *EBufferPreLoad*( ) – attach a full memory buffer to an extended buffer

SYNOPSIS

```
void EBufferPreLoad
    (
    unsigned int  flags,   /*  storage type flags */
    EBUFFER_T *   ebuffp,  /* extended buffer     */
    OCTET_T *     datap,   /* data pointer        */
    ALENGTH_T     datal    /* data length         */
    );
```

DESCRIPTION This routine attaches a full memory buffer to a buffer-control block. Use this routine when constructing a parameter for a procedure which requires buffers in the extended-buffer format.

flags should be set to the manifest constant **BFL_IS_DYNAMIC** if the buffer has been allocated from the dynamic pool. Otherwise, *flags* should be set to **BFL_IS_STATIC**. The location and length of the buffer are described by *datap* and *datal*, respectively.

RETURNS N / A

SEE ALSO **snmpEbufLib**

EBufferRemaining( )

NAME *EBufferRemaining*( ) – return the number of unused bytes remaining in buffer memory

SYNOPSIS

```
void    EBufferRemaining
    (
    EBUFFER_T *  ebuffp  /*  extended buffer */
    );
```

DESCRIPTION This routine returns the number of unused bytes remaining in the extended buffer memory.

RETURNS the number of unused bytes.

SEE ALSO **snmpEbufLib**

EBufferReset( )

NAME *EBufferReset*( ) – reset the extended buffer

SYNOPSIS

```
void    EBufferReset
    (
    EBUFFER_T *  ebuffp  /*  extended buffer */
    );
```

DESCRIPTION This routine sets the various pointers and counters so that the buffer is exactly as it would be after a call to *EBufferSetup*( ). The memory buffer itself is left unchanged.

RETURNS N / A

SEE ALSO **snmpEbufLib**

EBufferSetup( )

NAME *EBufferSetup*( ) – attach an empty memory buffer to an extended buffer

SYNOPSIS

```
void EBufferSetup
    (
    unsigned int  flags,   /*  storage type flags */
    EBUFFER_T *   ebuffp,  /* extended buffer     */
    OCTET_T *     datap,   /* data pointer        */
    ALENGTH_T     datal    /* data length         */
    );
```

DESCRIPTION This routine attaches an empty memory buffer to a buffer-control block.

flags should be set to the manifest constant **BFL_IS_DYNAMIC** if the buffer has been allocated from the dynamic pool. Otherwise, *flags* should be set to **BFL_IS_STATIC**. The location and length of the buffer are described by *datap* and *datal*, respectively.

RETURNS N / A

SEE ALSO **snmpEbufLib**

EBufferStart( )

NAME *EBufferStart*( ) – return a pointer to the first byte in the buffer memory

SYNOPSIS

```
OCTET_T *    EBufferStart
    (
    EBUFFER_T *  ebuffp  /* extended buffer */
    );
```

DESCRIPTION This routine returns a pointer to the first byte in the buffer memory.

RETURNS a pointer to the first memory byte.

SEE ALSO **snmpEbufLib**

EBufferUsed( )

NAME *EBufferUsed*( ) – return the number of used bytes in the buffer memory

SYNOPSIS

```
ALENGTH_T    EBufferUsed
    (
    EBUFFER_T *  ebuffp  /* extended buffer */
    );
```

DESCRIPTION This routine returns the number of used bytes currently in the buffer.

RETURNS the number of used bytes.

SEE ALSO **snmpEbufLib**

edi( )

NAME *edi*( ) – return the contents of register **edi** (also **esi** – **eax**) (i386/i486)

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: Target Shell*

eexattach( )

NAME *eexattach*( ) – publish the **eex** network interface and initialize the driver and device

SYNOPSIS
```
STATUS eexattach
    (
    int  unit,       /* unit number                      */
    int  port,       /* base I/O address                 */
    int  ivec,       /* interrupt vector number          */
    int  ilevel,     /* interrupt level                  */
    int  nTfds,      /* # of transmit frames (0=default) */
    int  attachment  /* 0=default, 1=AUI, 2=BNC, 3=TPE   */
    )
```

DESCRIPTION The routine publishes the **eex** interface by filling in a network interface record and adding this record to the system list. This routine also initializes the driver and the device to the operational state.

RETURNS OK or ERROR.

SEE ALSO **if_eex**, **ifLib**

eflags( )

NAME

eflags( ) – return the contents of the status register (i386/i486)

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: Target Shell*

eiattach( )

NAME

eiattach( ) – publish the **ei** network interface and initialize the driver and device

SYNOPSIS

```
STATUS eiattach
    (
    int     unit,     /* unit number                          */
    int     ivec,     /* interrupt vector number              */
    UINT8   sysbus,   /* sysbus field of SCP                  */
    char *  memBase,  /* address of memory pool or NONE       */
    int     nTfds,    /* no. of transmit frames (0 = default) */
    int     nRfds     /* no. of receive frames (0 = default)  */
    )
```

DESCRIPTION

This routine publishes the **ei** interface by filling in a network interface record and adding this record to the system list. This routine also initializes the driver and the device to the operational state.

The 82596 shares a region of memory with the driver. The caller of this routine can specify the address of this memory region, or can specify that the driver must obtain this memory region from the system resources.

The *sysbus* parameter accepts values as described in the Intel manual for the 82596. A default number of transmit/receive frames of 32 can be selected by passing zero in *nTfds* and *nRfds*. In other cases, the number of frames selected should be greater than two.

The *memBase* parameter is used to inform the driver about the shared memory region. If this parameter is set to the constant "NONE," then this routine will attempt to allocate the shared memory from the system. Any other value for this parameter is interpreted by this routine as the address of the shared memory region to be used.

If the caller provides the shared memory region, then the driver assumes that this region does not require cache coherency operations, nor does it require conversions between virtual and physical addresses.

If the caller indicates that this routine must allocate the shared memory region, then this routine will use *cacheDmaMalloc***( )** to obtain some non-cacheable memory. The attributes of this memory will be checked, and if the memory is not both read and write coherent, this routine will abort and return ERROR.

RETURNS OK or ERROR.

SEE ALSO **if_ei**, **ifLib**, *Intel 82596 User's Manual*

---

*eitpattach***( )**

NAME *eitpattach***( )** – publish the **ei** network interface for the TP41V and initialize the driver and device

SYNOPSIS

```
STATUS eitpattach
    (
    int     unit,     /* unit number                          */
    int     ivec,     /* interrupt vector number              */
    UINT8   sysbus,   /* sysbus field of SCP                  */
    char *  memBase,  /* address of memory pool or NONE       */
    int     nTfds,    /* no. of transmit frames (0 = default) */
    int     nRfds     /* no. of receive frames (0 = default)  */
    )
```

DESCRIPTION The routine publishes the **ei** interface by filling in a network interface record and adding this record to the system list. It also initializes the driver and the device.

The 82596 shares a region of memory with the driver. The caller of this routine can specify the address of this memory region, or can specify that the driver must obtain this memory region from the system resources.

The *sysbus* parameter accepts values as described in the Intel manual for the 82596. A default number of transmit / receive frames of 32 can be selected by passing zero in the parameters *nTfds* and *nRfds*. In other cases, the number of frames selected should be greater than two.

The *memBase* parameter is used to inform the driver about the shared memory region. If this parameter is set to the constant **NONE**, then *eitpattach***( )** will attempt to allocate the shared memory from the system. Any other value for this parameter is interpreted as the address of the shared memory region to be used.

If the caller provides the shared memory region, then the driver assumes that this region does not require cache coherency operations, nor does it require conversions between virtual and physical addresses.

If the caller indicates that this routine must allocate the shared memory region, then this routine will use the routine *cacheDmaMalloc***( )** to obtain some non-cacheable memory. The attributes of this memory will be checked, and if the memory is not both read and write coherent, this routine will abort and return ERROR.

RETURNS OK or ERROR.

SEE ALSO **if_eitp**, **if_ei**, **ifLib**, *Intel 82596 User's Manual*

---

*elcattach***( )**

NAME *elcattach***( )** – publish the **elc** network interface and initialize the driver and device

SYNOPSIS

```
STATUS elcattach
    (
    int  unit,     /* unit number                              */
    int  ioAddr,   /* address of elc's shared memory           */
    int  ivec,     /* interrupt vector to connect to           */
    int  ilevel,   /* interrupt level                          */
    int  memAddr,  /* address of elc's shared memory           */
    int  memSize,  /* size of elc's shared memory              */
    int  config    /* 0: RJ45 + AUI(Thick)  1: RJ45 + BNC(Thin) */
    )
```

DESCRIPTION This routine attaches an **elc** Ethernet interface to the network if the device exists. It makes the interface available by filling in the network interface record. The system will initialize the interface when it is ready to accept packets.

RETURNS OK or ERROR.

SEE ALSO **if_elc**, **ifLib**, **netShow**

elcShow( )

NAME *elcShow*( ) – display statistics for the SMC 8013WC **elc** network interface

SYNOPSIS

```
void elcShow
    (
    int   unit,  /* interface unit  */
    BOOL  zap    /* 1 = zero totals */
    )
```

DESCRIPTION This routine displays statistics about the **elc** Ethernet interface. It has two parameters:

unit interface unit; should be 0.

zap if 1, all collected statistics are cleared to zero.

RETURNS N / A

SEE ALSO **if_elc**

eltattach( )

NAME *eltattach*( ) – publish the **elt** interface and initialize the driver and device

SYNOPSIS

```
STATUS eltattach
    (
    int   unit,        /* unit number                     */
    int   port,        /* base I/O address                */
    int   ivec,        /* interrupt vector number         */
    int   intLevel,    /* interrupt level                 */
    int   nRxFrames,   /* # of receive frames (0=default) */
    int   attachment,  /* Ethernet connector to use       */
    char  *ifName      /* interface name                  */
    )
```

DESCRIPTION The routine publishes the **elt** interface by filling in a network interface record and adding the record to the system list. It also initializes the driver and the device to the operational state.

RETURNS OK or ERROR.

SEE ALSO **if_elt**, **ifLib**

eltShow( )

NAME *eltShow*( ) – display statistics for the 3C509 **elt** network interface

SYNOPSIS

```
void eltShow
    (
    int   unit,  /* interface unit  */
    BOOL  zap    /* 1 = zero totals */
    )
```

DESCRIPTION This routine displays statistics about the **elt** Ethernet network interface. It has two parameters:

unit interface unit; should be 0.

zap if 1, all collected statistics are cleared to zero.

RETURNS N / A

SEE ALSO **if_elt**

eneattach( )

NAME *eneattach*( ) – publish the **ene** network interface and initialize the driver and device

SYNOPSIS

```
STATUS eneattach
    (
    int  unit,    /* unit number                    */
    int  ioAddr,  /* address of ene's shared memory */
    int  ivec,    /* interrupt vector to connect to */
    int  ilevel   /* interrupt level                */
    )
```

DESCRIPTION This routine attaches an **ene** Ethernet interface to the network if the device exists. It makes the interface available by filling in the network interface record. The system will initialize the interface when it is ready to accept packets.

RETURNS OK or ERROR.

SEE ALSO **if_ene**, **ifLib**, **netShow**

eneShow( )

NAME *eneShow*( ) – display statistics for the NE2000 **ene** network interface

SYNOPSIS

```
void eneShow
    (
    int   unit,  /* interface unit  */
    BOOL  zap    /* 1 = zero totals */
    )
```

DESCRIPTION This routine displays statistics about the **ene** Ethernet network interface. It has two parameters:

unit interface unit; should be 0.

zap if 1, all collected statistics are cleared to zero.

RETURNS N / A

SEE ALSO **if_ene**

enpattach( )

NAME *enpattach*( ) – publish the **enp** network interface and initialize the driver and device

SYNOPSIS

```
STATUS enpattach
    (
    int   unit,      /* unit number                      */
    char  *addr,     /* address of enp's shared memory   */
    int   ivec,      /* interrupt vector to connect to   */
    int   ilevel,    /* interrupt level                  */
    int   enpAddrAm  /* address VME address modifier     */
    )
```

DESCRIPTION This routine publishes the **enp** interface by filling in a network interface record and adding this record to the system list. It also initializes the driver and the device to the operational state.

RETURNS OK or ERROR.

SEE ALSO **if_enp**

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*( )

etherAddrResolve( )

NAME

etherAddrResolve( ) – resolve an Ethernet address for a specified Internet address

SYNOPSIS

```
STATUS etherAddrResolve
    (
    struct ifnet  *pIf,         /* interface on which to send ARP req */
    char          *targetAddr,  /* name or Internet address of target */
    char          *eHdr,        /* where to return the Ethernet addr  */
    int           numTries,     /* number of times to try ARPing      */
    int           numTicks      /* number of ticks between ARPing     */
    )
```

DESCRIPTION

This routine uses the Address Resolution Protocol (ARP) and internal ARP cache to resolve the Ethernet address of a machine that owns the Internet address given in *targetAddr*.

The first argument *pIf* is a pointer to a variable of type **struct ifnet** which identifies the network interface through which the ARP request messages are to be sent out. The routine *ifunit*( ) is used to retrieve this pointer from the system in the following way:

```
struct ifnet *pIf;
...
pIf = ifunit ("ln0");
```

If *ifunit*( ) returns a non-NULL pointer, it is a valid pointer to the named network interface device structure of type **struct ifnet**. In the above example, *pIf* will be pointing to the data structure that describes the first LANCE network interface device if *ifunit*( ) is successful.

The six-byte Ethernet address is copied to *eHdr*, if the resolution of *targetAddr* is successful. *eHdr* must point to a buffer of at least six bytes.

RETURNS

OK if the address is resolved successfully, or ERROR if *eHdr* is NULL, *targetAddr* is invalid, or address resolution is unsuccessful.

SEE ALSO

etherLib, *etherOutput*( )

etherInputHookAdd( )

NAME *etherInputHookAdd*( ) – add a routine to receive all Ethernet input packets

SYNOPSIS

```
STATUS etherInputHookAdd
    (
    FUNCPTR  inputHook  /* routine to receive Ethernet input */
    )
```

DESCRIPTION This routine adds a hook routine that will be called for every Ethernet packet received. The calling sequence of the input hook routine is:

```
BOOL inputHook
     (
     struct ifnet *pIf,    /* interface packet was received on */
     char         *buffer, /* received packet                  */
     int          length   /* length of received packet        */
     )
```

The hook routine should return TRUE if it has handled the input packet and no further action should be taken with it. It should return FALSE if it has not handled the input packet and normal processing (e.g., Internet) should take place.

The packet is in a temporary buffer when the hook routine is called. This buffer will be reused upon return from the hook. If the hook routine needs to retain the input packet, it should copy it elsewhere.

IMPLEMENTATION A call to the function pointed to by the global function pointer **etherInputHookRtn** should be invoked in the receive routine of every network driver providing this service. For example:

```
...
#include "etherLib.h"
...
xxxRecv ()
...
/* call input hook if any */
    if ((etherInputHookRtn != NULL) &&
        (* etherInputHookRtn) (&ls->ls_if, (char *)eh, len))
        {
        return; /* input hook has already processed this packet */
        }
```

RETURNS OK, always.

SEE ALSO **etherLib**

etherInputHookDelete( )

NAME *etherInputHookDelete*( ) – delete a network interface input hook routine

SYNOPSIS
```
void etherInputHookDelete (void)
```

DESCRIPTION This routine deletes a network interface input hook.

RETURNS N / A

SEE ALSO **etherLib**

etherOutput( )

NAME *etherOutput*( ) – send a packet on an Ethernet interface

SYNOPSIS
```
STATUS etherOutput
    (
    struct ifnet         *pIf,           /* interface on which to send */
    struct ether_header  *pEtherHeader,  /* Ethernet header to send    */
    char                 *pData,         /* data to send               */
    int                  dataLength      /* # of bytes of data to send */
    )
```

DESCRIPTION This routine sends a packet on the specified Ethernet interface by calling the interface's output routine directly.

The first argument *pIf* is a pointer to a variable of type **struct ifnet** which contains some useful information about the network interface. A routine named *ifunit*( ) can retrieve this pointer from the system in the following way:

```
struct ifnet *pIf;
...
pIf = ifunit ("ln0");
```

If *ifunit*( ) returns a non-NULL pointer, it is a valid pointer to the named network interface device structure of type **struct ifnet**. In the above example, *pIf* points to the data structure that describes the first LANCE network interface device if *ifunit*( ) is successful.

The second argument *pEtherHeader* should contain a valid Ethernet address of the machine for which the message contained in the argument *pData* is intended. If the Ethernet address of this machine is fixed and well-known to the user, filling in the

structure **ether_header** can be accomplished by using *bcopy***( )** to copy the six-byte Ethernet address into the **ether_dhost** field of the structure **ether_header**. Alternatively, users can make use of the routine *etherAddrResolve***( )** which will use ARP (Address Resolution Protocol) to resolve the Ethernet address for a specified Internet address.

RETURNS OK, or ERROR if the routine runs out of mbufs.

SEE ALSO **etherLib**, *etherAddrResolve***( )**

*etherOutputHookAdd***( )**

NAME *etherOutputHookAdd***( )** – add a routine to receive all Ethernet output packets

SYNOPSIS

```
STATUS etherOutputHookAdd
    (
    FUNCPTR  outputHook  /* routine to receive Ethernet output */
    )
```

DESCRIPTION This routine adds a hook routine that will be called for every Ethernet packet that is transmitted.

The calling sequence of the output hook routine is:

```
BOOL outputHook
    (
    struct ifnet *pIf,    /* interface packet will be sent on */
    char         *buffer, /* packet to transmit               */
    int          length   /* length of packet to transmit     */
    )
```

The hook is called immediately before transmission. The hook routine should return TRUE if it has handled the output packet and no further action should be taken with it. It should return FALSE if it has not handled the output packet and normal transmission should take place.

The Ethernet packet data is in a temporary buffer when the hook routine is called. This buffer will be reused upon return from the hook. If the hook routine needs to retain the output packet, it should be copied elsewhere.

IMPLEMENTATION A call to the function pointed to be the global function pointer **etherOutputHookRtn** should be invoked in the transmit routine of every network driver providing this service. For example:

```
...
#include "etherLib.h"
```

```
...
xxxStartOutput ()
/* call output hook if any */
if ((etherOutputHookRtn != NULL) &&
     (* etherOutputHookRtn) (&ls->ls_if, buf0, len))
    {
    /* output hook has already processed this packet */
    }
else
...
```

RETURNS OK, always.

SEE ALSO **etherLib**

---

*etherOutputHookDelete***( )**

NAME *etherOutputHookDelete***( )** – delete a network interface output hook routine

SYNOPSIS

```
void etherOutputHookDelete (void)
```

DESCRIPTION This routine deletes a network interface output hook.

RETURNS N / A

SEE ALSO **etherLib**

---

*evbNs16550HrdInit***( )**

NAME *evbNs16550HrdInit***( )** – initialize the NS 16550 chip

SYNOPSIS

```
void evbNs16550HrdInit
    (
    EVBNS16550_CHAN  *pChan
    )
```

DESCRIPTION This routine is called to reset the NS 16550 chip to a quiescent state.

SEE ALSO **evbNs16550Sio**

evbNs16550Int( )

NAME *evbNs16550Int*( ) – handle a receiver / transmitter interrupt for the NS 16550 chip

SYNOPSIS
```
void evbNs16550Int
    (
    EVBNS16550_CHAN  *pChan
    )
```

DESCRIPTION This routine is called to handle interrupts. If there is another character to be transmitted, it sends it. If the interrupt handler is called erroneously (for example, if a device has never been created for the channel), it disables the interrupt.

SEE ALSO **evbNs16550Sio**

evtBufferAddress( )

NAME *evtBufferAddress*( ) – return the address of the event buffer (WindView)

SYNOPSIS
```
char * evtBufferAddress (void)
```

DESCRIPTION This routine returns the address of the WindView event buffer. This can be useful when using WindView in post-mortem mode.

RETURN Address of the event buffer, otherwise NULL.

SEE ALSO **evtBufferLib**, **wvLib**

evtBufferIsEmpty( )

NAME *evtBufferIsEmpty*( ) – check whether the event buffer is empty (WindView)

SYNOPSIS
```
BOOL evtBufferIsEmpty (void)
```

DESCRIPTION This routine returns a boolean indicating an empty or non-empty event buffer. This can be useful in post-mortem mode. The WindView event buffer is empty only if event logging was not enabled or the buffer was erased (e.g., on reboot).

RETURN TRUE if empty, otherwise FALSE.

SEE ALSO **evtBufferLib**, **wvLib**

evtBufferToFile( )

NAME *evtBufferToFile*( ) – transfer the contents of the event buffer to a file (WindView)

SYNOPSIS

```
STATUS evtBufferToFile
    (
    char *  filename  /* file name to hold event data */
    )
```

DESCRIPTION This routine transfers the contents of the WindView event buffer to a host file, specified by *filename*. This can be useful when working with WindView in post-mortem mode.

The data transferred to *filename* is a raw event log; use the WindView **Analyze** command to examine the file.

RETURN OK, or ERROR if the event buffer is not successfully transferred to the specified file.

SEE ALSO **evtBufferLib**, **wvLib**

evtBufferUpLoad( )

NAME *evtBufferUpLoad*( ) – upload the contents of the event buffer to the host (WindView)

SYNOPSIS

```
void evtBufferUpLoad (void)
```

DESCRIPTION This routine uploads the contents of the WindView event buffer to the host. This can be useful when using WindView in post-mortem mode. The uploading is carried out by the data transfer routine specified in *connRtnSet*( ).

If the default data transfer routine is used, WindView or **evtRecv** must be running on the host to collect the event data.

SEE ALSO **evtBufferLib**, **connLib**, **wvLib**, **evtRecv**

exattach( )

NAME *exattach*( ) – publish the **ex** network interface and initialize the driver and device

SYNOPSIS

```
STATUS exattach
    (
    int   unit,     /* logical number of this interface       */
    char  *pDev,    /* bus address of EXOS device ports       */
    int   ivec,     /* interrupt vector                       */
    int   ilevel,   /* interrupt level                        */
    int   exDmaAm,  /* VME addr modifier to access CPU memory */
    int   exAdrsAm  /* VME addr modifier to access EXOS ports */
    )
```

DESCRIPTION This routine publishes the **ex** interface by filling in a network interface record and adding this record to the system list. It also initializes the driver and the device to the operational state.

RETURNS OK or ERROR.

SEE ALSO **if_ex**

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 describe below:

```
void excHandler (ESFPPC *);
```

When the C routine returns, the exception stub restores the registers saved in the ESF and continues the current task execution.

RETURNS OK always.

SEE ALSO **excArchLib**, *excIntConnect***( )**, *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 ()
```

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)

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 (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.

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.

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.

RETURNS OK always.

SEE ALSO **excArchLib**, *excConnect***( )**, *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)

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.

SPARC:
All vectors from 0 (offset 0x000) through 255 (offset 0xff0) are initialized.

i960:
The i960 fault table is filled with a default fault handler, and all non-reserved vectors in the i960 interrupt table are filled with a default interrupt handler.

MIPS:
All MIPS exception, trap, and interrupt vectors are set to default handlers.

i386 / i486:
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.

NOTE: This routine is usually called from the system start-up routine, *usrInit*( ), in **usrConfig.c**. It must be called before interrupts are enabled. (SPARC: It must also be called when the system runs with the on-chip windows (no stack)).

RETURNS OK, always.

SEE ALSO **excArchLib**, **excLib**

excVecSet( )

NAME *excVecSet*( ) – set a CPU exception vector (PowerPC)

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.

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.

ERRNOS 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*.

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**

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.

SEE ALSO **ansiStdio**, *fflush*( )

fdDevCreate( )

NAME *fdDevCreate*( ) – create a device for a floppy disk

SYNOPSIS

```
BLK_DEV *fdDevCreate
    (
    int  drive,     /* driver number of floppy disk (0 - 3)   */
    int  fdType,    /* type of floppy disk                    */
    int  nBlocks,   /* device size in blocks (0 = whole disk) */
    int  blkOffset  /* offset from start of device            */
    )
```

DESCRIPTION This routine creates a device for a specified floppy disk.

The *drive* parameter is the drive number of the floppy disk; valid values are 0 to 3.

The *fdType* parameter specifies the type of diskette, which is described in the structure table **fdTypes[]** in **sysLib.c**. *fdType* is an index to the table. Currently the table contains two diskette types:

– An *fdType* of 0 indicates the first entry in the table (3.5” 2HD, 1.44MB);

– An *fdType* of 1 indicates the second entry in the table (5.25” 2HD, 1.2MB).

Members of the **fdTypes[]** structure are:

```
int  sectors;       /* no of sectors */
int  sectorsTrack;  /* sectors per track */
int  heads;         /* no of heads */
int  cylinders;     /* no of cylinders */
int  secSize;       /* bytes per sector, 128 << secSize */
char gap1;          /* gap1 size for read, write */
char gap2;          /* gap2 size for format */
char dataRate;      /* data transfer rate */
char stepRate;      /* stepping rate */
char headUnload;    /* head unload time */
char headLoad;      /* head load time */
char mfm;           /* MFM bit for read, write, format */
char sk;            /* SK bit for read */
char *name;         /* name */
```

The *nBlocks* parameter specifies the size of the device, in blocks. If *nBlocks* is zero, the whole disk is used.

The *blkOffset* parameter specifies an offset, in blocks, from the start of the device to be used when writing or reading the floppy 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.) Normally, *blkOffset* is 0.

RETURNS A pointer to a block device structure (**BLK_DEV**) or NULL if memory cannot be allocated for the device structure.

SEE ALSO **nec765Fd**, *fdDrv***( )**, *fdRawio***( )**, *dosFsMkfs***( )**, *dosFsDevInit***( )**, *rt11FsDevInit***( )**, *rt11FsMkfs***( )**, *rawFsDevInit***( )**

---

*fdDrv***( )**

NAME *fdDrv***( )** – initialize the floppy disk driver

SYNOPSIS

```
STATUS fdDrv
    (
    int  vector,  /* interrupt vector */
    int  level    /* interrupt level  */
    )
```

DESCRIPTION This routine initializes the floppy driver, sets up interrupt vectors, and performs hardware initialization of the floppy chip.

This routine should be called exactly once, before any reads, writes, or calls to *fdDevCreate***( )**. Normally, it is called by *usrRoot***( )** in **usrConfig.c**.

RETURNS OK.

SEE ALSO **nec765Fd**, *fdDevCreate***( )**, *fdRawio***( )**

*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.

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***( )**

---

*fdRawio***( )**

NAME *fdRawio***( )** – provide raw I/O access

SYNOPSIS

```
STATUS fdRawio
    (
    int     drive,   /* drive number of floppy disk (0 - 3) */
    int     fdType,  /* type of floppy disk                 */
    FD_RAW  *pFdRaw  /* pointer to FD_RAW structure         */
    )
```

DESCRIPTION This routine is called when the raw I/O access is necessary.

The *drive* parameter is the drive number of the floppy disk; valid values are 0 to 3.

The *fdType* parameter specifies the type of diskette, which is described in the structure table **fdTypes[]** in **sysLib.c**. *fdType* is an index to the table. Currently the table contains two diskette types:

– An *fdType* of 0 indicates the first entry in the table (3.5” 2HD, 1.44MB);

– An *fdType* of 1 indicates the second entry in the table (5.25” 2HD, 1.2MB).

The *pFdRaw* is a pointer to the structure **FD_RAW**, defined in **nec765Fd.h**

RETURNS OK or ERROR.

SEE ALSO **nec765Fd**, *fdDrv***( )**, *fdDevCreate***( )**

feiattach( )

NAME

feiattach( ) – publish the **fei** network interface

SYNOPSIS

```
STATUS feiattach
    (
    int     unit,     /* unit number                                */
    char *  memBase,  /* address of shared memory (NONE = malloc) */
    int     nCFD,     /* command frames (0 = default)              */
    int     nRFD,     /* receive frames (0 = default)              */
    int     nRFDLoan  /* loanable rx frames (0 = default, -1 = 0) */
    )
```

DESCRIPTION

This routine publishes the **fei** interface by filling in a network interface record and adding the record to the system list.

The 82557 shares a region of main memory with the CPU. The caller of this routine can specify the address of this shared memory region through the *memBase* parameter; if *memBase* is set to the constant **NONE**, the driver will allocate the shared memory region.

If the caller provides the shared memory region, the driver assumes that this region does not require cache coherency operations.

If the caller indicates that *feiattach*( ) must allocate the shared memory region, *feiattach*( ) will use *cacheDmaMalloc*( ) to obtain a block of non-cacheable memory. The attributes of this memory will be checked, and if the memory is not both read and write coherent, *feiattach*( ) will abort and return ERROR.

A default number of 32 command (transmit) and 32 receive frames can be selected by passing zero in the parameters *nCFD* and *nRFD*, respectively. If *nCFD* or *nRFD* is used to select the number of frames, the values should be greater than two.

A default number of 8 loanable receive frames can be selected by passing zero in the parameters *nRFDLoan*, else set *nRFDLoan* to the desired number of loanable receive frames. If *nRFDLoan* is set to -1, no loanable receive frames will be allocated / used.

RETURNS

OK, or ERROR if the driver could not be published and initialized.

SEE ALSO

if_fei, **ifLib**, *Intel 82557 User's Manual*

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*( )

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.

CAVEAT 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.

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*

*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 **INCLUDE_FLOATING_POINT** has been defined in **configAll.h**, 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.

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**

fnattach( )

NAME *fnattach*( ) – publish the **fn** network interface and initialize the driver and device

SYNOPSIS

```
STATUS fnattach
    (
    int  unit  /* unit number */
    )
```

DESCRIPTION The routine publishes the **fn** interface by filling in a network interface record and adding this record to the system list. This routine also initializes the driver and the device to the operational state.

RETURNS OK or ERROR.

SEE ALSO **if_fn**

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***( )**

---

*fp***( )**

NAME *fp***( )** – return the contents of register **fp** (i960)

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

DESCRIPTION This command extracts the contents of register **fp**, the frame pointer, from the TCB of a specified task. If *taskId* is omitted or 0, the current default task is assumed.

RETURNS The contents of the **fp** register.

SEE ALSO **dbgArchLib**, *VxWorks Programmer's Guide: Target Shell*

---

*fp0***( )**

NAME *fp0***( )** – return the contents of register **fp0** (also **fp1** – **fp3**) (i960KB, i960SB)

SYNOPSIS
```
double fp0
    (
    volatile int  taskId  /* task ID, 0 means default task */
    )
```

DESCRIPTION This command extracts the contents of the floating-point register **fp0** from the TCB of a specified task. If *taskId* is omitted or 0, the current default task is assumed.

Routines are provided for the floating-point registers **fp0** – **fp3**: *fp0***( )** – *fp3***( )**.

RETURNS The contents of the **fp0** register (or the requested register).

SEE ALSO **dbgArchLib**, *VxWorks Programmer's Guide: Target Shell*

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 **INCLUDE_HW_FP** is defined in **configAll.h**.

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, SPARC, i386 / i486:

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.

i960:

This routine merely indicates whether VxWorks was compiled with the flag **-DCPU=I960KB**.

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.

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.)

SPARC: registers **fsr** and **fpq**
registers **f0** – **f31**

i960: registers **fp0** – **fp3**

MIPS: register **fpcsr**
registers **fp0** – **fp31**

i386/i486: control word, status word, tag word, IP offset, CS selector, data operand offset, and operand selector (4 bytes each)
registers **st0** – **st7** (8 bytes each)

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 */
    )
```

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.)
SPARC:	registers **fsr** and **fpq** registers **f0** – **f31**
i960:	registers **fp0** – **fp3**
MIPS:	register **fpcsr** registers **fp0** – **fp31**
i386/i486:	control word, status word, tag word, IP offset, CS selector, data operand offset, and operand selector (4 bytes each) registers **st0** – **st7** (8 bytes each)

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. The facility is included automatically when **INCLUDE_SHOW_ROUTINES** is defined in **configAll.h**.

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 *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.

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 as follows:

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 plus a terminating null character, added automatically. The conversion specifier includes all subsequent characters 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 is in the scanlist and the next right bracket 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 there is no field width). 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.

SEE ALSO **ansiStdio**, *ftell***( )**

---

*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***( )**

fsrShow( )

NAME

fsrShow( ) – display the meaning of a specified fsr value, symbolically (SPARC)

SYNOPSIS

```
void fsrShow
    (
    UINT  fsrValue  /* fsr value to show */
    )
```

DESCRIPTION

This routine displays the meaning of all the fields in a specified **fsr** value, symbolically.

Extracted from **reg.h**:

```
Definition of bits in the Sun-4 FSR (Floating-point Status Register)
  ------------------------------------------------------------
 |  RD |  RP | TEM |  res | FTT | QNE | PR | FCC | AEXC | CEXC |
 |-----|---- |-----|------|-----|-----|----|-----|------|------|
  31 30 29 28 27 23 22  17 16 14   13   12  11 10 9    5 4    0
```

For compatibility with future revisions, reserved bits are defined to be initialized to zero and, if written, must be preserved.

EXAMPLE

```
-> fsrShow 0x12345678
Rounding Direction: nearest or even if tie.
Rounding Precision: single.
Trap Enable Mask:
   underflow.
Floating-point Trap Type: IEEE exception.
Queue Not Empty: FALSE;
Partial Remainder: TRUE;
Condition Codes: less than.
Accumulated exceptions:
   inexact divide-by-zero invalid.
Current exceptions:
   overflow invalid
```

RETURNS

N / A

SEE ALSO

dbgArchLib, *SPARC Architecture Manual*

fstat( )

NAME *fstat*( ) – get file status information (POSIX)

SYNOPSIS

```
STATUS fstat
    (
    int          fd,     /* file descriptor for file to check */
    struct stat  *pStat  /* pointer to stat structure         */
    )
```

DESCRIPTION This routine obtains various characteristics of a file (or directory). The file must already have been opened using *open*( ) or *creat*( ). The *fd* parameter is the file descriptor returned by *open*( ) or *creat*( ).

The *pStat* parameter is a pointer to a **stat** structure (defined in **stat.h**). This structure must be allocated before *fstat*( ) is called.

Upon return, the fields in the **stat** structure are updated to reflect the characteristics of the file.

RETURNS OK or ERROR.

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 **stat** structure (defined in **stat.h**). This structure must be allocated before *fstat*( ) is called.

On return, the fields in the **statfs** structure are updated to reflect 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 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***( )**

---

*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

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. 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*( ).)

RETURNS The file descriptor of the data socket created, or ERROR.

SEE ALSO **ftpLib**, *ftpHookup*( ), *ftpLogin*( ), *ftpCommand*( ), *ftpXfer*( )

ftpdDelete( )

NAME *ftpdDelete*( ) – clean up and finalize the FTP server task

SYNOPSIS
```
void ftpdDelete (void)
```

DESCRIPTION This routine finalizes and deletes the main FTP server task, *ftpdTask*( ). It cleans up all active sessions which it has spawned, and reclaims all resources used by each active slot in the active session list. All sockets associated with FTP services will be closed, and all memory dynamically allocated will be freed.

RETURNS N / A

SEE ALSO **ftpdLib**

ftpdInit( )

NAME *ftpdInit*( ) – initialize the FTP server task

SYNOPSIS

```
STATUS ftpdInit
    (
    int  stackSize  /* stack size for the ftpdTask */
    )
```

DESCRIPTION This routine will spawn a new FTP server task, if one does not already exist. If an existing FTP server task is running already, *ftpdInit*( ) returns without creating a new task. It reports whether a new FTP task was successfully spawned. The argument *stackSize* can be specified to change the default stack size for the FTP server task. The default size is set in the global variable **ftpdWorkTaskStackSize**.

RETURNS OK, or ERROR if a new FTP task cannot be successfully created.

SEE ALSO **ftpdLib**

ftpdTask( )

NAME *ftpdTask*( ) – FTP server daemon task

SYNOPSIS

```
STATUS ftpdTask (void)
```

DESCRIPTION This routine processes incoming FTP client requests by spawning a new FTP work task for each connection that is set up.

RETURNS OK, or ERROR when the task terminates with errors due to socket related I/O problems, linked-list management, or task creation.

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*( )

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*( )

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 gets a command reply on the specified control socket. All the lines of a reply are read (multi-line replies are indicated with the continuation character "-" as the fourth character of all but the last line).

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 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 on standard error.

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

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 (i.e., file delete or rename), *pDataSock* should be NULL, in which case no data connection will be established.

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.

```
int ctrlSock;
int dataSock;
char buf [512];
int nBytes;
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) != 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.

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

g0( )

NAME

g0( ) – return the contents of register **g0**, also **g1** – **g7** (SPARC) and **g1** – **g14** (i960)

SYNOPSIS

```
int g0
    (
    int  taskId  /* task ID, 0 means default task */
    )
```

DESCRIPTION

This command extracts the contents of global register **g0** from the TCB of a specified task. If *taskId* is omitted or 0, the current default task is assumed.

Routines are provided for all global registers:

SPARC:	*g0*( ) – *g7*( )	(**g0** – **g7**)
i960:	*g0*( ) – *g14*( )	(**g0** – **g14**)

RETURNS The contents of register **g0** (or the requested register).

SEE ALSO **dbgArchLib**, *VxWorks Programmer's Guide: Target Shell*

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 0 or -1.

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*. *namelen* 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**

getproc_error( )

NAME *getproc_error*( ) – indicate that a **getproc** operation encountered an error

SYNOPSIS

```
void getproc_error
    (
    SNMP_PKT_T *  pPkt,      /* internal representation of the snmp packet */
    VB_T *        pVarBind,  /* var bind being processed                   */
    INT_32_T      error      /* error code                                 */
    )
```

DESCRIPTION This routine indicates that **getproc** encountered an error and cannot retrieve the requested value.

RETURNS N / A

SEE ALSO **snmpProcLib**

getproc_good( )

NAME *getproc_good*( ) – indicate successful completion of a **getproc** procedure

SYNOPSIS

```
void getproc_good
    (
    SNMP_PKT_T *  pPkt,     /* internal representation of the snmp packet */
    VB_T *        pVarBind  /* var bind being processed                   */
    )
```

DESCRIPTION This routine is called when **getproc** successfully retrieves the value for an SNMP variable binding.

RETURNS N / A

SEE ALSO **snmpProcLib**

getproc_got_empty( )

NAME *getproc_got_empty*( ) – indicate retrieval of a null value

SYNOPSIS

```
void getproc_got_empty
    (
    SNMP_PKT_T *  pPkt,     /* internal representation of the snmp packet */
    VB_T *        pVarBind  /* var bind being processed                   */
    )
```

DESCRIPTION This routine is called from **getproc** or **nextproc** when a null value is retrieved for a variable binding.

RETURNS N / A

SEE ALSO **snmpProcLib**

getproc_got_int32( )

NAME *getproc_got_int32*( ) – indicate retrieval of a 32-bit integer

SYNOPSIS

```
void getproc_got_int32
    (
    SNMP_PKT_T *  pPkt,      /* internal representation of the snmp packet */
    VB_T *        pVarBind,  /* var bind being processed                   */
    INT_32_T      data       /* 32 bit integer value for varbind           */
    )
```

DESCRIPTION This routine is called from **getproc** or **nextproc** when a 32-bit integer value is retreived for a variable binding.

RETURNS N / A

SEE ALSO **snmpProcLib**

getproc_got_ip_address( )

NAME *getproc_got_ip_address*( ) – indicate retrieval of an IP address

SYNOPSIS

```
void getproc_got_ip_address
    (
    SNMP_PKT_T *  pPkt,      /* internal representation of the snmp packet */
    VB_T *        pVarBind,  /* var bind being processed                   */
    UINT_32_T     addrData   /* ip address value                           */
    )
```

DESCRIPTION This routine is called from **getproc** or **nextproc** when an IP address is retrieved for a variable binding.

RETURNS N / A

SEE ALSO **snmpProcLib**

getproc_got_object_id( )

NAME *getproc_got_object_id*( ) – indicate retrieval of an object identifier

SYNOPSIS

```
void getproc_got_object_id
    (
    SNMP_PKT_T *  pPkt,      /* internal representation of the snmp packet */
    VB_T *        pVarBind,  /* var bind being processed                   */
    int           length,    /* The length of the oid                      */
    OIDC_T *      pOid,      /* The oid to attach                          */
    int           flag       /* The dynamic flag static (0), dynamic (1)   */
    );
```

DESCRIPTION This routine is called from **getproc** or **nextproc** when an object identifier is retrieved for a variable binding. If *flag* is 1, then *pOid* is assumed to point to dynamic memory which will be later freed by the agent.

RETURNS N / A

SEE ALSO **snmpProcLib**

*getproc_got_string***( )**

NAME *getproc_got_string***( )** – indicate retrieval of a string

SYNOPSIS

```
void getproc_got_string
    (
    SNMP_PKT_T * pPkt,       /* internal representation of the snmp packet */
    VB_T *       pVarBind,   /* var bind being processed                   */
    ALENGTH_T    size,       /* size of string in octets                   */
    OCTET_T *    data,       /* string data                                */
    int          dynamicFlg, /* storage type - dynamic or static           */
    OCTET_T      type        /* SNMP type of string data                   */
    )
```

DESCRIPTION This routine is called from **getproc** or **nextproc** when a string is retrieved for a variable binding. The string data is stored in an extended buffer in the variable-binding structure. *dynamicFlg* indicates the storage type used by the extended buffer; if *dynamicFlg* is non-zero, the buffer is assumed to have been allocated dynamically via *snmpdMemoryAlloc***( )** and is freed later with *snmpdMemoryFree***( )**. Otherwise, the buffer is assumed to be static.

RETURNS N / A

SEE ALSO **snmpProcLib**

*getproc_got_uint32***( )**

NAME *getproc_got_uint32***( )** – indicate retrieval of a 32-bit unsigned integer

SYNOPSIS

```
void getproc_got_uint32
    (
    SNMP_PKT_T *  pPkt,      /* internal representation of the snmp packet */
    VB_T *        pVarBind,  /* var bind being processed                   */
    UINT_32_T     data,      /* unsigned 32 bit integer value for varbind  */
    OCTET_T       type       /* SNMP type of value                         */
    )
```

DESCRIPTION This routine is called from **getproc** or **nextproc** when a 32-bit unsigned integer value is retrieved for a variable binding.

SEE ALSO **snmpProcLib**

*getproc_got_uint64***( )**

NAME *getproc_got_uint64***( )** – indicate retrieval of a 64-bit unsigned integer

SYNOPSIS

```
void getproc_got_uint64
    (
    SNMP_PKT_T *  pPkt,      /* internal representation of the snmp packet */
    VB_T *        pVarBind,  /* var bind being processed                   */
    UINT_64_T *   data       /* 64 bit data                                */
    )
```

DESCRIPTION This routine is called from **getproc** or **nextproc** when a 64-bit unsigned integer value is retrieved for a variable binding.

RETURNS N / A

SEE ALSO **snmpProcLib**

*getproc_got_uint64_high_low***( )**

NAME *getproc_got_uint64_high_low***( )** – indicate retrieval of a 64-bit unsigned integer with high and low halves

SYNOPSIS

```
void getproc_got_uint64_high_low
    (
    SNMP_PKT_T *  pPkt,      /* internal representation of the snmp packet */
    VB_T *        pVarBind,  /* var bind being processed                   */
    UINT_32_T     high,      /* high half of data                          */
    UINT_32_T     low        /* lower half of data                         */
    )
```

DESCRIPTION This routine is called from **getproc** or **nextproc** when a 64-bit unsigned integer value with both high and low halves is retrieved for a variable binding.

RETURNS N / A

SEE ALSO **snmpProcLib**

*getproc_nosuchins***( )**

NAME *getproc_nosuchins***( )** – indicates that no such instance exists

SYNOPSIS

```
void getproc_nosuchins
    (
    SNMP_PKT_T *  pPkt,     /* internal representation of the snmp packet */
    VB_T *        pVarBind  /* var bind being processed                   */
    );
```

DESCRIPTION This routine is called from **getproc** if the requested instance does not exist.

RETURNS N / A

SEE ALSO **snmpProcLib**

*getproc_started***( )**

NAME *getproc_started***( )** – indicate that a **getproc** operation has begun

SYNOPSIS

```
void getproc_started
    (
    SNMP_PKT_T *  pPkt,     /* internal representation of the snmp packet */
    VB_T *        pVarBind  /* var bind being processed                   */
    )
```

DESCRIPTION This routine indicates that **getproc** for the specified SNMP variable binding has been started and should not be started again.

RETURNS N / A

SEE ALSO **snmpProcLib**

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*. *namelen* 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**

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.

RETURNS OK, or ERROR if there is an invalid socket, an unknown option, or the call is unable to get the specified option.

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**

RETURNS 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**

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).

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*

help( )

NAME

help( ) – print a synopsis of selected routines

SYNOPSIS

```
void help (void)
```

DESCRIPTION

This command prints the following list of the calling sequences for commonly used routines, mostly contained in **usrLib**.

```
help                        Print this list
dbgHelp                     Print debug help info
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=0, 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
cd        "path"            Set current working path
pwd                         Print working path
devs                        List devices
ls        ["path"[,long]]   List contents of directory
ll        ["path"]          List contents of directory - long format
rename    "old","new"       Change name of file
copy      ["in"][,"out"]    Copy in file to out file (0 = std in/out)
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)
diskFormat  "device"       Format disk
diskInit  "device"         Initialize file system on disk
squeeze   "device"         Squeeze free space on RT-11 device
NOTE:  Arguments specifying <task> can be either task ID or name.
```

RETURNS N / A

SEE ALSO **usrLib**, *VxWorks Programmer's Guide: Target 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 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 host is unknown.

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 preallocated with (**MAXHOSTNAMELEN** + 1) bytes of memory and is NULL-terminated unless insufficient space is provided.

NOTE: This routine does not look for aliases. Host names are limited to **MAXHOSTNAMELEN** (from **hostLib.h**) characters.

RETURNS OK, or ERROR if 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*( ).

RETURNS The Internet address (as an integer), or ERROR if the host is unknown.

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_NET_INIT** is defined in **configAll.h**.

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'
```

CAVEAT 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*

i0( )

NAME *i0*( ) – return the contents of register **i0** (also **i1** – **i7**) (SPARC)

SYNOPSIS
```
int i0
    (
    int  taskId  /* task ID, 0 means default task */
    )
```

DESCRIPTION This command extracts the contents of in register **i0** from the TCB of a specified task. If *taskId* is omitted or 0, the current default task is assumed.

Similar routines are provided for all in registers (**i0** – **i7**): *i0*( ) – *i7*( ).

The frame pointer is accessed via **i6**.

RETURNS The contents of register **i0** (or the requested register).

SEE ALSO **dbgArchLib**, *VxWorks Programmer's Guide: Target Shell*

i8250HrdInit( )

NAME *i8250HrdInit*( ) – initialize the chip

SYNOPSIS
```
void i8250HrdInit
    (
    I8250_CHAN  *pChan
    )
```

DESCRIPTION This routine is called to reset the chip in a quiescent state.

SEE ALSO **i8250Sio**

i8250Int( )

NAME *i8250Int*( ) – handle a receiver / transmitter interrupt

SYNOPSIS
```
void i8250Int
    (
    I8250_CHAN  *pChan
    )
```

DESCRIPTION This routine gets called to handle interrupts. If there is another character to be transmitted, it sends it. If not, or if a device has never been created for this channel, just disable the interrupt.

SEE ALSO **i8250Sio**

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*( )

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 **netShow**

ideDevCreate( )

NAME *ideDevCreate*( ) – create a device for a IDE disk

SYNOPSIS
```
BLK_DEV *ideDevCreate
    (
    int  drive,     /* drive number for hard drive (0 or 1)   */
    int  nBlocks,   /* device size in blocks (0 = whole disk) */
    int  blkOffset  /* offset from start of device            */
    )
```

DESCRIPTION This routine creates a device for a specified IDE disk.

drive is a drive number for the hard drive: it must be 0 or 1.

The *nBlocks* parameter specifies the size of the device, in blocks. If *nBlocks* is zero, the whole disk is used.

The *blkOffset* parameter specifies an offset, in blocks, from the start of the device to be used when writing or reading the hard disk. This offset is added to the block numbers passed by the file system during disk accesses. (VxWorks file systems always use block numbers beginning at zero for the start of a device.)

RETURNS A pointer to a block device structure (**BLK_DEV**), or NULL if memory cannot be allocated for the device structure.

SEE ALSO **ideDrv**, *dosFsMkfs*( ), *dosFsDevInit*( ), *rt11FsDevInit*( ), *rt11FsMkfs*( ), *rawFsDevInit*( )

ideDrv( )

NAME

ideDrv( ) – initialize the IDE driver

SYNOPSIS

```
STATUS ideDrv
    (
    int   vector,       /* interrupt vector                */
    int   level,        /* interrupt level                 */
    BOOL  manualConfig  /* 1 = initialize drive parameters */
    )
```

DESCRIPTION

This routine initializes the IDE driver, sets up interrupt vectors, and performs hardware initialization of the IDE chip.

This routine should be called exactly once, before any reads, writes, or calls to *ideDevCreate*( ). Normally, it is called by *usrRoot*( ) in **usrConfig.c**.

The *ideDrv*( ) call requires a configuration type, *manualConfig*. If this argument is 1, the driver will initialize drive parameters; if the argument is 0, the driver will not initialize drive parameters.

The drive parameters are the number of sectors per track, the number of heads, and the number of cylinders. They are stored in the structure table **ideTypes[]** in **sysLib.c**. The table has two entries: the first is for drive 0; the second is for drive 1. The table has two other members which are used by the driver: the number of bytes per sector and the precompensation cylinder. These two members should be set properly. Definitions of the structure members are:

```
int cylinders;              /* number of cylinders */
int heads;                  /* number of heads */
int sectorsTrack;           /* number of sectors per track */
int bytesSector;            /* number of bytes per sector */
int precomp;                /* precompensation cylinder */
```

RETURNS

OK, or ERROR if initialization fails.

SEE ALSO

ideDrv, *ideDevCreate*( )

ideRawio( )

NAME *ideRawio*( ) – provide raw I/O access

SYNOPSIS

```
STATUS ideRawio
    (
    int        drive,   /* drive number for hard drive (0 or 1) */
    IDE_RAW *  pIdeRaw  /* pointer to IDE_RAW structure         */
    )
```

DESCRIPTION This routine is called when the raw I/O access is necessary.

drive is a drive number for the hard drive: it must be 0 or 1.

The *pIdeRaw* is a pointer to the structure **IDE_RAW** which is defined in **ideDrv.h**

RETURNS OK or ERROR.

SEE ALSO **ideDrv**

ifAddrGet( )

NAME *ifAddrGet*( ) – get the Internet address of a network interface

SYNOPSIS

```
STATUS ifAddrGet
    (
    char  *interfaceName,    /* name of interface           */
    char  *interfaceAddress  /* buffer for Internet address */
    )
```

DESCRIPTION This routine gets the Internet address of a specified network interface and copies it to *interfaceAddress*.

RETURNS OK or ERROR.

SEE ALSO **ifLib**, *ifAddrSet*( ), *ifDstAddrSet*( ), *ifDstAddrGet*( )

ifAddrSet( )

NAME *ifAddrSet*( ) – set an interface address for a network interface

SYNOPSIS

```
STATUS ifAddrSet
    (
    char  *interfaceName,    /* name of interface to configure          */
    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*( ).

RETURNS OK, or ERROR if the interface cannot be set.

SEE ALSO **ifLib**, *ifAddrGet*( ), *ifDstAddrSet*( ), *ifDstAddrGet*( )

ifBroadcastGet( )

NAME *ifBroadcastGet*( ) – get the broadcast address for a network interface

SYNOPSIS

```
STATUS ifBroadcastGet
    (
    char  *interfaceName,    /* name of interface           */
    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             */
    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             */
    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           */
    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 */
    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 */
    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 */
    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
Brings the network up or down.

IFF_DEBUG
Turns on debugging for the driver interface if supported.

IFF_LOOPBACK
Set for a loopback network.

IFF_NOTRAILERS
Always set (VxWorks does not use the trailer protocol).

IFF_PROMISC
Tells the driver to accept all packets, not just broadcast packets and packets addressed to itself.

IFF_ALLMULTI
Tells the driver to accept all multicast packets.

IFF_NOARP
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
Identifies a point-to-point interface such as PPP or SLIP.

IFF_RUNNING
Set when the device turns on.

IFF_BROADCAST
Identifies a broadcast interface.

RETURNS OK or ERROR.

SEE ALSO **ifLib**, *ifFlagChange***( )**, *ifFlagGet***( )**

---

*ifMaskGet***( )**

NAME *ifMaskGet***( )** – get the subnet mask for a network interface

SYNOPSIS
```
STATUS ifMaskGet
    (
    char  *interfaceName,  /* name of interface      */
    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 for which to set mask */
    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 */
    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 */
    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*( )

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 are associated with the specified 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 (e.g., loopback, point-to-point, broadcast, promiscuous, arp, running, debug).

EXAMPLE The following call displays all interfaces whose names begin with “ln”:

```
-> 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**, **etherLib**

---

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*( ).

---

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.

RETURNS N / A

SEE ALSO **netShow**

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.

EXAMPLE The following example returns 0x5a000002:

```
inet_addr ("90.0.0.2");
```

RETURNS The Internet address, or ERROR.

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 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 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 a buffer for the resulting value must provided.

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 dot 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 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 given an ASCII Internet network number.

EXAMPLE The following example returns 0x5a:

```
inet_network ("90");
```

RETURNS The Internet address version of an ASCII string.

SEE ALSO **inetLib**

inet_ntoa( )

NAME *inet_ntoa*( ) – convert network address to dot notation

SYNOPSIS

```
char *inet_ntoa
    (
    struct in_addr  inetAddress  /* inet address */
    )
```

DESCRIPTION This routine converts an Internet address in network format to dot 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 network address to dot notation and store in 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 dot notation.

This routine is identical to the UNIX *inet_ntoa***( )** routine except that a buffer of size **INET_ADDR_LEN** must be provided.

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**

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*. The call performs two sanity checks on the data to be decompressed:

(1) It looks for a magic number at the start of the data to verify that it is really a compressed stream.

(2) It optionally checksums the entire data 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 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 simply calls *intHandlerCreate*( ) and *intVecSet*( ). The address of the handler returned by *intHandlerCreate*( ) is what actually goes in the interrupt vector.

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**

intDisable( )

NAME *intDisable*( ) – disable corresponding interrupt bits (MIPS, PowerPC)

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 MIPS For MIPS, the macros **SR_IBIT1** – **SR_IBIT8** define bits that may be set.

RETURNS MIPS: The previous contents of the status register.
PowerPC: OK or ERROR.

SEE ALSO **intArchLib**

intEnable( )

NAME *intEnable*( ) – enable corresponding interrupt bits (MIPS, PowerPC)

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 MIPS For MIPS, it is advised that the level be a combination of **SR_IBIT1** – **SR_IBIT8**.

RETURNS MIPS: The previous contents of the status register. PowerPC: OK or ERROR.

SEE ALSO **intArchLib**

intHandlerCreate( )

NAME *intHandlerCreate*( ) – construct an interrupt handler for a C routine (MC680x0, SPARC, i960, x86, MIPS)

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**

intLevelSet( )

NAME *intLevelSet*( ) – set the interrupt level (MC680x0, SPARC, i960, x86)

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

SPARC: 0 – 15

i960: 0 – 31

On SPARC systems, traps must be enabled before the call.

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**, *sysVwTrap*( )

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, SPARC, i960, and i386/i486 architectures, interrupts are disabled at the level set by *intLockLevelSet*( ). The default lock-out level is the highest interrupt level (MC680x0 = 7, SPARC = 15, i960 = 31, i386/i486 = 1).

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 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
SPARC:	interrupt level (0 – 15)
i960:	interrupt level (0 – 31)
i386/i486:	interrupt enable flag (IF) bit from **EFLAGS** register
MIPS:	status register
PowerPC:	MSR register value

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 (for example, *taskSuspend***( )** or *taskDelay***( )**) or that cause a higher priority task to be ready (for example., *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 (MC680x0, SPARC, i960, x86)

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, SPARC = 15, i960 = 31, i386/i486 = 1) is initially set by *kernelInit*( ) when VxWorks is initialized.

RETURNS The interrupt level currently stored in the interrupt lock-out mask.

SEE ALSO **intArchLib**, *intLockLevelSet*( )

intLockLevelSet( )

NAME *intLockLevelSet*( ) – set the current interrupt lock-out level (MC680x0, SPARC, i960, x86)

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, SPARC = 15, i960 = 31, i386/i486 = 1) is initially set by *kernelInit*( ) when VxWorks is initialized.

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**

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 (MC680x0, SPARC, i960, x86, MIPS)

SYNOPSIS
```
FUNCPTR *intVecBaseGet (void)
```

DESCRIPTION This routine returns the current vector base address, which is set with *intVecBaseSet***( )**.

RETURNS The current vector base address (i960 = value of **sysIntTable** set in **sysLib**, MIPS = 0 always).

SEE ALSO **intArchLib**, *intVecBaseSet***( )**

*intVecBaseSet***( )**

NAME *intVecBaseSet***( )** – set the vector (trap) base address (MC680x0, SPARC, i960, x86, MIPS)

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 (0x1000 for SPARC), until modified by calls to this routine.

NOTE SPARC On SPARC processors, the vector base address must be on a 4 Kbyte boundary (that is, its bottom 12 bits must be zero).

NOTE 68000 The 68000 has no vector base register; thus, this routine is a no-op for 68000 systems.

NOTE I960 This routine is a no-op for i960 systems. The interrupt vector table is located in **sysLib**, and moving it by *intVecBaseSet***( )** would require resetting the processor. Also, the vector base is cached on-chip in the PRCB and thus cannot be set from this routine.

NOTE MIPS The MIPS processors have no vector base register; thus this routine is a no-op for this architecture.

RETURNS N / A

SEE ALSO **intArchLib**, *intVecBaseGet***( )**, *intVecGet***( )**, *intVecSet***( )**

---

*intVecGet***( )**

NAME *intVecGet***( )** – get an interrupt vector (MC680x0, SPARC, i960, x86, MIPS)

SYNOPSIS
```
FUNCPTR intVecGet
    (
    FUNCPTR *  vector  /* vector offset */
    )
```

DESCRIPTION This routine returns a pointer to the exception / interrupt handler attached to a specified vector. The vector is specified as an offset into the CPU's vector table. This vector table starts, by default, at:

MC680x0:	0
SPARC:	0x1000
i960:	**sysIntTable** in **sysLib**
MIPS:	**excBsrTbl** in **excArchLib**
i386 / i486:	0

However, the vector table may be set to start at any address with *intVecBaseSet***( )** (on CPUs for which it is available).

NOTE I960 The interrupt table location is reinitialized to *sysIntTable* after booting. This location is returned by *intVecBaseGet***( )**.

RETURNS A pointer to the exception / interrupt handler attached to the specified vector.

SEE ALSO **intArchLib**, *intVecSet***( )**, *intVecBaseSet***( )**

intVecSet( )

NAME

intVecSet( ) – set a CPU vector (trap) (MC680x0, SPARC, i960, x86, MIPS)

SYNOPSIS

```
void intVecSet
    (
    FUNCPTR *  vector,   /* vector offset              */
    FUNCPTR    function  /* address to place in vector */
    )
```

DESCRIPTION

This routine attaches an exception / interrupt / trap handler to a specified vector. The vector is specified as an offset into the CPU's vector table. This vector table starts, by default, at:

MC680x0:	0
SPARC:	0x1000
i960:	**sysIntTable** in **sysLib**
MIPS:	**excBsrTbl** in **excArchLib**
i386 / i486:	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*( ).

NOTE SPARC

This routine generates code to:

(1) save volatile registers;

(2) fix possible window overflow;

(3) read the processor state register into register **%L**0; and

(4) jump to the specified address.

The *intVecSet*( ) routine puts this generated code into the trap table entry corresponding to *vector*.

Window overflow and window underflow are sacred to the kernel and may not be pre-empted. They are written here only to track changing trap base registers (TBRs). With the "branch anywhere" scheme (as opposed to the branch PC-relative + / -8 megabytes) the first instruction in the vector table must not be a change of flow control nor affect any critical registers. The JMPL that replaces the BA will always execute the next vector's first instruction.

NOTE I960

Vectors 0-7 are illegal vectors; using them puts the vector into the priorities / pending portion of the table, which yields undesirable actions. The i960CA caches the NMI vector in internal RAM at system power-up. This is where the vector is taken when the NMI

occurs. Thus, it is important to check to see if the vector being changed is the NMI vector, and, if so, to write it to internal RAM.

NOTE MIPS On MIPS CPUs the vector table is set up statically in software.

RETURNS N / A

SEE ALSO **intArchLib**, *intVecBaseSet***( )**, *intVecGet***( )**

*intVecTableWriteProtect***( )**

NAME *intVecTableWriteProtect***( )** – write-protect exception vector table (MC680x0, SPARC, i960, x86)

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.

RETURNS OK, or ERROR if memory cannot be write-protected.

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. Most requests are passed on to the driver for handling. The following example, which places the filename of the file descriptor in *nameBuf*, is handled at the I/O interface level:

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

Since the availability of *ioctl*( ) functions is driver-specific, these functions are discussed separately in **tyLib**, **pipeDrv**, **nfsDrv**, **dosFsLib**, **rt11FsLib**, and **rawFsLib**.

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*

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*( )

ioMmuMicroSparcInit( )

NAME *ioMmuMicroSparcInit*( ) – initialize the microSparc I / II I / O MMU data structures

SYNOPSIS
```
STATUS ioMmuMicroSparcInit
    (
    void *  physBase,  /* first valid DMA physical address */
    UINT    range      /* range covered by I/O Page Table  */
    )
```

DESCRIPTION This routine initializes the I / O MMU for S-Bus DMA with the TMS390S10 and Mb86904. This function is executed after the VxWorks kernel is initialized. The memory allocated for the **ioPage** tables is write protected and cache inhibited only if one of the MMU libraries (**vmBaseLib** or **vmLib**) is initialized. It has been implemented this way because boot ROMs do not initialize the MMU library in **bootConfig.c**; instead, they initialize the MMU separately from **romInit.s**.

RETURNS OK, or ERROR if unable to satisfy request.

SEE ALSO **ioMmuMicroSparcLib**, *ioMmuMicroSparcMap*( )

ioMmuMicroSparcMap( )

NAME

ioMmuMicroSparcMap( ) – map the I/O MMU for microSparc I/II (TMS390S10/MB86904)

SYNOPSIS

```
STATUS ioMmuMicroSparcMap
    (
    UINT    dvmaAdrs,  /* ioDvma virtual address to map */
    void *  physBase,  /* physical address to add       */
    UINT    size       /* size to map                   */
    )
```

DESCRIPTION

This routine maps the specified amount of memory (*size*), starting at the specified **ioDvma** virtual address (*dvmaAdrs*), to the specified physical base (*physBase*).

Do not call *ioMmuMicroSparcMap*( ) without first calling the initialization routine *ioMmuMicroSparcInit*( ), because this routine depends on the data structures initialized there. The *ioMmuMicroSparcMap*( ) routine checks that the I/O MMU range specified at initialization is sufficient for the size of the memory being mapped. The physical base specified should be on a page boundary. Similarly, the size of the memory being mapped must be a multiple of the page size.

RETURNS

OK, or ERROR if unable to satisfy request.

SEE ALSO

ioMmuMicroSparcLib, *ioMmuMicroSparcInit*( )

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     /* no. of servicing driver,      */
                        /* returned by iosDrvInstall()   */
    )
```

DESCRIPTION

This routine adds a device to the I/O system device list, making the device available for subsequent *open*( ) and *creat*( ) calls.

The parameter *pDevHdr* is a pointer to a device header, **DEV_HDR** (defined in **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*( )

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**

*ip_to_rlist***( )**

NAME *ip_to_rlist***( )** – convert an IP address to an array of OID components

SYNOPSIS

```
int ip_to_rlist
    (
    UINT_32_T  ip_address,  /* ip address                   */
    OIDC_T *   object_id    /* pointer to object identifier */
    );
```

DESCRIPTION This routine uses an IP address to fill an array of values (of type **long**) with the byte components of the IP address. The return value is the number of components filled, always 4.

RETURNS 4, always.

SEE ALSO **snmpAuxLib**

*irint***( )**

NAME *irint***( )** – convert a double-precision value to an integer

SYNOPSIS

```
int irint
    (
    double  x  /* argument */
    )
```

DESCRIPTION This routine converts a double-precision value *x* to an integer using the selected IEEE rounding direction.

CAVEAT The rounding direction is not pre-selectable and is fixed for round-to-the-nearest.

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.

CAVEAT 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 character, 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*.

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*

l0( )

NAME *l0*( ) – return the contents of register **l0** (also **l1** – **l7**) (SPARC)

SYNOPSIS
```
int l0
    (
    int  taskId  /* task ID, 0 means default task */
    )
```

DESCRIPTION This command extracts the contents of local register **l0** from the TCB of a specified task. If *taskId* is omitted or 0, the current default task is assumed.

Similar routines are provided for all local registers (**l0** – **l7**): *l0*( ) – *l7*( ).

RETURNS The contents of register **l0** (or the requested register).

SEE ALSO **dbgArchLib**, *VxWorks Programmer's Guide: Target 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.

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**.

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 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*

ll( )

NAME *ll*( ) – do 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:

```
-> ls dirName, TRUE
```

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 **usrLib**, *ls*( ), *stat*( ), *VxWorks Programmer's Guide: Target Shell*

lnattach( )

NAME *lnattach*( ) – publish the **ln** network interface and initialize the driver and device

SYNOPSIS

```
STATUS lnattach
    (
    int    unit,      /* unit number                                  */
    char   *devAdrs,  /* LANCE I/O address                            */
    int    ivec,      /* interrupt vector                             */
    int    ilevel,    /* interrupt level                              */
    char   *memAdrs,  /* address of memory pool (-1 = malloc it)      */
    ULONG  memSize,   /* only used if memory pool is NOT malloc()'d */
    int    memWidth,  /* byte-width of data (-1 = any width)          */
    int    spare,     /* not used                                     */
    int    spare2     /* not used                                     */
    )
```

DESCRIPTION This routine publishes the **ln** interface by filling in a network interface record and adding this record to the system list. This routine also initializes the driver and the device to the operational state.

The *memAdrs* parameter can be used to specify the location of the memory that will be shared between the driver and the device. The value NONE is used to indicate that the driver should obtain the memory.

The *memSize* parameter is valid only if the *memAdrs* parameter is not set to NONE, in which case *memSize* indicates the size of the provided memory region.

The *memWidth* parameter sets the memory pool's data port width (in bytes); if it is NONE, any data width is used.

BUGS To zero out LANCE data structures, this routine uses *bzero***( )**, which ignores the *memWidth* specification and uses any size data access to write to memory.

RETURNS OK or ERROR.

SEE ALSO **if_ln**

---

*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 | LOCAL | GLOBAL | ALL]_SYMBOLS) */
    )
```

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 | LOCAL | GLOBAL | ALL]_SYMBOLS)    */
    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 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. In this case, NULL will be returned after the module is loaded.

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***( )**.

In addition, 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.

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 internally use another storage class known as *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 VxWorks loader is most often used to load partially linked object modules. When the VxWorks loader encounters a variable labeled as common, memory for the variable is allocated, with *malloc***( )**, and the variable is entered in the system symbol table (if specified) at that address. Note that most UNIX loaders have an option that forces resolution of the common storage while leaving the module relocatable (e.g., 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, or the file format is illegal.

SEE ALSO **loadLib**, *VxWorks Programmer's Guide: Basic OS*

---

*loattach***( )**

NAME *loattach***( )** – publish the **lo** network interface and initialize the driver and pseudo-device

SYNOPSIS

```
STATUS loattach (void)
```

DESCRIPTION This routine attaches an **lo** Ethernet interface to the network, if the interface exists. It makes the interface available by filling in the network interface record. The system initializes the interface when it is ready to accept packets.

RETURNS OK.

SEE ALSO **if_loop**

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 nonmonetary quantities.

char *thousands_sep
The character used to separate groups of digits before the decimal-point character in formatted nonmonetary quantities.

char *grouping
A string whose elements indicate the size of each group of digits in formatted nonmonetary 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.

INCLUDE FILES **time.h**

RETURNS A pointer to a **tm** structure containing the local broken-down time.

SEE ALSO **ansiTime**

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**

---

*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**

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**

*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 routine **vxencrypt** in **host/***hostOs*/bin should also be replaced by a host version of *encryptionRoutine*. For more information, see the manual entry for *loginEncryptInstall*( ).

RETURNS OK, or ERROR if the password is invalid.

SEE ALSO **loginLib**, *loginEncryptInstall***( )**, **vxencrypt**

*loginEncryptInstall***( )**

NAME *loginEncryptInstall***( )** – install an encryption routine

SYNOPSIS

```
void loginEncryptInstall
    (
    FUNCPTR  rtn,  /* function pointer to encryption routine      */
    int      var   /* argument to the encryption routine (unused) */
    )
```

DESCRIPTION This routine allows the user to install a custom encryption routine. The custom routine *rtn* must be of the following form:

```
STATUS encryptRoutine
       (
       char *password,               /* string to encrypt    */
       char *encryptedPassword       /* resulting encryption */
       )
```

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 **INCLUDE_SECURITY** is defined in **configAll.h**, 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.

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**

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, the *logMsg*( ) routine 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).

For more detailed information about the use of *logMsg***( )**, see the manual entry for **logLib**.

EXAMPLE If the following code were executed by task 20:

```
{
name = "GRONK";
num = 123;
logMsg ("ERROR - name = %s, num = %d.\n", name, num, 0, 0, 0, 0);
}
```

the following error message would appear on the system log:

```
0x180400 (t20): ERROR - name = GRONK, num = 123.
```

RETURNS The number of bytes written to the log queue, or EOF if the routine is unable to write a message.

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 the *setjmp***( )** routine that used the same **jmp_buf** specified in the argument *env*. The restored environment includes the program counter, thus transferring control to the *setjmp***( )** caller.

If there was no corresponding *setjmp***( )** call, or if the function containing the corresponding *setjmp***( )** routine call 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***( )**

lptDevCreate( )

NAME *lptDevCreate*( ) – create a device for an LPT port

SYNOPSIS

```
STATUS lptDevCreate
    (
    char  *name,   /* name to use for this device             */
    int   channel  /* physical channel for this device (0 - 2) */
    )
```

DESCRIPTION This routine creates a device for a specified LPT port. Each port to be used should have exactly one device associated with it by calling this routine.

For instance, to create the device **/lpt/0**, the proper call would be:

```
lptDevCreate ("/lpt/0", 0);
```

RETURNS OK, or ERROR if the driver is not installed, the channel is invalid, or the device already exists.

SEE ALSO **lptDrv**, *lptDrv*( )

lptDrv( )

NAME *lptDrv*( ) – initialize the LPT driver

SYNOPSIS

```
STATUS lptDrv
    (
    int           channels,   /* LPT channels  */
    LPT_RESOURCE  *pResource  /* LPT resources */
    )
```

DESCRIPTION This routine initializes the LPT driver, sets up interrupt vectors, and performs hardware initialization of the LPT ports.

This routine should be called exactly once, before any reads, writes, or calls to *lptDevCreate*( ). Normally, it is called by *usrRoot*( ) in **usrConfig.c**.

RETURNS OK, or ERROR if the driver cannot be installed.

SEE ALSO **lptDrv**, *lptDevCreate*( )

lptShow( )

NAME *lptShow*( ) – show LPT statistics

SYNOPSIS
```
void lptShow
    (
    UINT  channel  /* channel (0 - 2) */
    )
```

DESCRIPTION This routine shows statistics for a specified LPT port.

RETURNS N / A

SEE ALSO **lptDrv**

ls( )

NAME *ls*( ) – list the contents of a directory

SYNOPSIS
```
STATUS ls
    (
    char  *dirName,  /* name of dir to list      */
    BOOL  doLong     /* if TRUE, do long listing */
    )
```

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. For a long listing, any entries that describe subdirectories are also flagged with the label “DIR”.

The *dirName* parameter specifies which directory to list. If *dirName* is omitted or NULL, the current working directory is listed.

Empty directory entries and dosFs volume label entries are not reported.

NOTE: When used with **netDrv** devices (FTP or RSH), *doLong* has no effect.

RETURNS OK or ERROR.

SEE ALSO **usrLib**, *ll*( ), *lsOld*( ), *stat*( ), *VxWorks Programmer’s Guide: Target Shell*

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**

lsOld( )

NAME *lsOld*( ) – list the contents of an RT-11 directory

SYNOPSIS
```
STATUS lsOld
    (
    char  *dirName  /* device to list */
    )
```

DESCRIPTION This command is the old version of *ls*( ), which used the old-style *ioctl*( ) function **FIODIRENTRY** to get information about entries in a directory. Since VxWorks 5.0, a new version of *ls*( ), which uses POSIX directory and file functions, has replaced the older routine.

This version remains in the system to support certain drivers that do not currently support the POSIX directory and file functions. This includes **netDrv**, which provides the Remote Shell (RSH) and File Transfer Protocol (FTP) mode remote file access (although

nfsDrv, which uses NFS, does support the directory calls). Also, the new *ls***( )** no longer reports empty directory entries on RT-11 disks (i.e., the entries that describe unallocated sections of an RT-11 disk).

If no directory name is specified, the current working directory is listed.

RETURNS OK, or ERROR if the directory cannot be opened.

SEE ALSO **usrLib**, *ls***( )**, *VxWorks Programmer's Guide: Target Shell*

---

*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**

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*( )

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**

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**

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***( )**

*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***( )**

*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 out 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.

RETURNS OK, always.

ERRNO **S_m2Lib_CANT_CREATE_IF_SEM**

SEE ALSO **m2IfLib**, *m2IfGroupInfoGet*( ), *m2IfTblEntryGet*( ), *m2IfTblEntrySet*( ), *m2IfDelete*( )

m2IfTblEntryGet( )

NAME *m2IfTblEntryGet*( ) – get a MIB-II interface-group table entry

SYNOPSIS
```
STATUS m2IfTblEntryGet
    (
    int                search,      /* M2_EXACT_VALUE or
M2_NEXT_VALUE      */
    M2_INTERFACETBL *  pIfReqEntry  /* pointer to requested interface entry
*/
    )
```

DESCRIPTION This routine maps the MIB-II interface index to the system's internal interface index. 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**. If the status of the interface has changed since it was last read, the user trap routine is called.

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
    (
    M2_INTERFACETBL *  pIfTblEntry  /* pointer to requested entry to change
*/
    )
```

DESCRIPTION This routine selects the interface specified in the input parameter *pIfTblEntry* and sets the interface 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).

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***( )**

*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 *pMib2SysDescr pMib2SysContact*, *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***( )**, *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 the routing table entry is deleted.

– If *varToSet* has the either the **M2_IP_ROUTE_DEST** or **M2_IP_ROUTE_NEXT_HOP** bit set, then either a new route entry is added to the table or an existing route entry is changed.

RETURNS OK if successful, otherwise ERROR.

SEE ALSO **m2IpLib**, *m2IpInit***( )**, *m2IpGroupInfoGet***( )**, *m2IpGroupInfoSet***( )**, *m2IpAddrTblEntryGet***( )**, *m2IpRouteTblEntryGet***( )**, *m2IpDelete***( )**

*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***( )**

*m68302SioInit***( )**

NAME *m68302SioInit***( )** – initialize an **M68302_CP**

SYNOPSIS

```
void m68302SioInit
    (
    M68302_CP *  pCp
    )
```

DESCRIPTION This routine initializes the driver function pointers and then resets the chip to a quiescent state. The BSP must already have initialized all the device addresses and the **baudFreq** fields in the **M68302_CP** structure before passing it to this routine. The routine resets the device and initializes everything to support polled mode (if possible), but does not enable interrupts.

RETURNS N / A

SEE ALSO **m68302Sio**

m68302SioInit2( )

NAME *m68302SioInit2*( ) – initialize a **M68302_CP** (part 2)

SYNOPSIS
```
void m68302SioInit2
    (
    M68302_CP *  pCp
    )
```

DESCRIPTION Enables interrupt mode of operation.

RETURNS N / A

SEE ALSO **m68302Sio**

m68332DevInit( )

NAME *m68332DevInit*( ) – initialize the SCC

SYNOPSIS
```
void m68332DevInit
    (
    M68332_CHAN  *pChan
    )
```

DESCRIPTION This initializes the chip to a quiescent state.

RETURNS N / A

SEE ALSO **m68332Sio**

m68332Int( )

NAME *m68332Int*( ) – handle an SCC interrupt

SYNOPSIS
```
void m68332Int
    (
    M68332_CHAN  *pChan
    )
```

DESCRIPTION This routine handles SCC interrupts.

RETURNS N / A

SEE ALSO **m68332Sio**

m68360DevInit( )

NAME *m68360DevInit*( ) – initialize the SCC

SYNOPSIS
```
void m68360DevInit
    (
    M68360_CHAN  *pChan
    )
```

DESCRIPTION This routine is called to initialize the chip to a quiescent state.

SEE ALSO **m68360Sio**

m68360Int( )

NAME *m68360Int*( ) – handle an SCC interrupt

SYNOPSIS
```
void m68360Int
    (
    M68360_CHAN  *pChan
    )
```

DESCRIPTION This routine gets called to handle SCC interrupts.

SEE ALSO **m68360Sio**

m68562HrdInit( )

NAME *m68562HrdInit*( ) – initialize the DUSCC

SYNOPSIS
```
void m68562HrdInit
    (
    M68562_QUSART  *pQusart
    )
```

DESCRIPTION The BSP must have already initialized all the device addresses, etc in **M68562_DUSART** structure. This routine resets the chip in a quiescent state.

SEE ALSO **m68562Sio**

m68562RxInt( )

NAME *m68562RxInt*( ) – handle a receiver interrupt

SYNOPSIS
```
void m68562RxInt
    (
    M68562_CHAN  *pChan
    )
```

RETURNS N / A

SEE ALSO **m68562Sio**

*m68562RxTxErrInt***( )**

NAME *m68562RxTxErrInt***( )** – handle a receiver / transmitter error interrupt

SYNOPSIS

```
void m68562RxTxErrInt
    (
    M68562_CHAN  *pChan
    )
```

DESCRIPTION Only the receive overrun condition is handled.

RETURNS N / A

SEE ALSO **m68562Sio**

*m68562TxInt***( )**

NAME *m68562TxInt***( )** – handle a transmitter interrupt

SYNOPSIS

```
void m68562TxInt
    (
    M68562_CHAN  *pChan
    )
```

DESCRIPTION If there is another character to be transmitted, it sends it. If not, or if a device has never been created for this channel, disable the interrupt.

RETURNS N / A

SEE ALSO **m68562Sio**

m68681Acr( )

NAME *m68681Acr*( ) – return the contents of the DUART auxiliary control register

SYNOPSIS

```
UCHAR m68681Acr
    (
    M68681_DUART *  pDuart
    )
```

DESCRIPTION This routine returns the contents of the auxilliary control register (ACR). The ACR is not directly readable; a copy of the last value written is kept in the DUART data structure.

RETURNS The contents of the auxilliary control register.

SEE ALSO **m68681Sio**

m68681AcrSetClr( )

NAME *m68681AcrSetClr*( ) – set and clear bits in the DUART auxiliary control register

SYNOPSIS

```
void m68681AcrSetClr
    (
    M68681_DUART *  pDuart,
    UCHAR           setBits,   /* which bits to set in the ACR   */
    UCHAR           clearBits  /* which bits to clear in the ACR */
    )
```

DESCRIPTION This routine sets and clears bits in the DUART auxiliary control register (ACR). It sets and clears bits in a local copy of the ACR, then writes that local copy to the DUART. This means that all changes to the ACR must be performed by this routine. Any direct changes to the ACR are lost the next time this routine is called.

Set has priority over clear. Thus you can use this routine to update multiple bit fields by specifying the field mask as the clear bits.

RETURNS N / A

SEE ALSO **m68681Sio**

m68681DevInit( )

NAME *m68681DevInit*( ) – intialize a **M68681_DUART**

SYNOPSIS
```
void m68681DevInit
    (
    M68681_DUART *  pDuart
    )
```

DESCRIPTION The BSP must already have initialized all the device addresses and register pointers in the **M68681_DUART** structure as described in **m68681Sio**. This routine initializes some transmitter and receiver status values to be used in the interrupt mask register and then resets the chip to a quiescent state.

RETURNS N / A

SEE ALSO **m68681Sio**

m68681DevInit2( )

NAME *m68681DevInit2*( ) – intialize a **M68681_DUART**, part 2

SYNOPSIS
```
void m68681DevInit2
    (
    M68681_DUART *  pDuart
    )
```

DESCRIPTION This routine is called as part of *sysSerialHwInit2*( ). It tells the driver that interrupt vectors are connected and that it is safe to allow interrupts to be enabled.

RETURNS N / A

SEE ALSO **m68681Sio**

m68681Imr( )

NAME *m68681Imr*( ) – return the current contents of the DUART interrupt-mask register

SYNOPSIS

```
UCHAR m68681Imr
    (
    M68681_DUART *  pDuart
    )
```

DESCRIPTION This routine returns the contents of the interrupt-mask register (IMR). The IMR is not directly readable; a copy of the last value written is kept in the DUART data structure.

RETURNS The contents of the interrupt-mask register.

SEE ALSO **m68681Sio**

m68681ImrSetClr( )

NAME *m68681ImrSetClr*( ) – set and clear bits in the DUART interrupt-mask register

SYNOPSIS

```
void m68681ImrSetClr
    (
    M68681_DUART *  pDuart,
    UCHAR           setBits,   /* which bits to set in the IMR   */
    UCHAR           clearBits  /* which bits to clear in the IMR */
    )
```

DESCRIPTION This routine sets and clears bits in the DUART interrupt-mask register (IMR). It sets and clears bits in a local copy of the IMR, then writes that local copy to the DUART. This means that all changes to the IMR must be performed by this routine. Any direct changes to the IMR are lost the next time this routine is called.

Set has priority over clear. Thus you can use this routine to update multiple bit fields by specifying the field mask as the clear bits.

RETURNS N / A

SEE ALSO **m68681Sio**

*m68681Int***( )**

NAME *m68681Int***( )** – handle all DUART interrupts in one vector

SYNOPSIS

```
void m68681Int
    (
    M68681_DUART *  pDuart
    )
```

DESCRIPTION This routine handles all interrupts in a single interrupt vector. It identifies and services each interrupting source in turn, using edge-sensitive interrupt controllers.

RETURNS N / A

SEE ALSO **m68681Sio**

*m68681Opcr***( )**

NAME *m68681Opcr***( )** – return the state of the DUART output port configuration register

SYNOPSIS

```
UCHAR m68681Opcr
    (
    M68681_DUART *  pDuart
    )
```

DESCRIPTION This routine returns the state of the output port configuration register (OPCR) from the saved copy in the DUART data structure. The actual OPCR contents are not directly readable.

RETURNS The state of the output port configuration register.

SEE ALSO **m68681Sio**

m68681OpcrSetClr( )

NAME *m68681OpcrSetClr*( ) – set and clear bits in the DUART output port configuration register

SYNOPSIS

```
void m68681OpcrSetClr
    (
    M68681_DUART *  pDuart,
    UCHAR           setBits,   /* which bits to set in the OPCR   */
    UCHAR           clearBits  /* which bits to clear in the OPCR */
    )
```

DESCRIPTION This routine sets and clears bits in the DUART output port configuration register (OPCR). It sets and clears bits in a local copy of the OPCR, then writes that local copy to the DUART. This means that all changes to the OPCR must be performed by this routine. Any direct changes to the OPCR are lost the next time this routine is called.

Set has priority over clear. Thus you can use this routine to update multiple bit fields by specifying the field mask as the clear bits.

RETURNS N / A

SEE ALSO **m68681Sio**

m68681Opr( )

NAME *m68681Opr*( ) – return the current state of the DUART output port register

SYNOPSIS

```
UCHAR m68681Opr
    (
    M68681_DUART *  pDuart
    )
```

DESCRIPTION This routine returns the current state of the output port register (OPR) from the saved copy in the DUART data structure. The actual OPR contents are not directly readable.

RETURNS The current state of the output port register.

SEE ALSO **m68681Sio**

m68681OprSetClr( )

NAME *m68681OprSetClr*( ) – set and clear bits in the DUART output port register

SYNOPSIS

```
void m68681OprSetClr
    (
    M68681_DUART *  pDuart,
    UCHAR           setBits,   /* which bits to set in the OPR   */
    UCHAR           clearBits  /* which bits to clear in the OPR */
    )
```

DESCRIPTION This routine sets and clears bits in the DUART output port register (OPR). It sets and clears bits in a local copy of the OPR, then writes that local copy to the DUART. This means that all changes to the OPR must be performed by this routine. Any direct changes to the OPR are lost the next time this routine is called.

Set has priority over clear. Thus you can use this routine to update multiple bit fields by specifying the field mask as the clear bits.

RETURNS N / A

SEE ALSO **m68681Sio**

m68901DevInit( )

NAME *m68901DevInit*( ) – initialize a **M68901_CHAN** structure

SYNOPSIS

```
void m68901DevInit
    (
    M68901_CHAN *  pChan
    )
```

DESCRIPTION This routine initializes the driver function pointers and then resets the chip to a quiescent state. The BSP must have already initialized all the device addresses and the **baudFreq** fields in the **M68901_CHAN** structure before passing it to this routine.

RETURNS N / A

SEE ALSO **m68901Sio**

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 ()
```

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.

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***( )**

*mb86940DevInit***( )**

NAME *mb86940DevInit***( )** – install the driver function table

SYNOPSIS
```
void mb86940DevInit
    (
    MB86940_CHAN  *pChan
    )
```

DESCRIPTION This routine installs the driver function table. It also prevents the serial channel from functioning by disabling the interrupt.

RETURNS N / A

SEE ALSO **mb86940Sio**

mb87030CtrlCreate( )

NAME

mb87030CtrlCreate( ) – create a control structure for an MB87030 SPC

SYNOPSIS

```
MB_87030_SCSI_CTRL *mb87030CtrlCreate
    (
    UINT8    *spcBaseAdrs,   /* base address of SPC                   */
    int      regOffset,      /* addr offset between consecutive regs. */
    UINT     clkPeriod,      /* period of controller clock (nsec)     */
    int      spcDataParity,  /* type of input to SPC DP (data parity) */
    FUNCPTR  spcDMABytesIn,  /* SCSI DMA input function               */
    FUNCPTR  spcDMABytesOut  /* SCSI DMA output function              */
    )
```

DESCRIPTION

This routine creates a data structure that must exist before the SPC chip can be used. This routine should be called once and only once for a specified SPC. It should be the first routine called, since it allocates memory for a structure needed by all other routines in the library.

After calling this routine, at least one call to *mb87030CtrlInit*( ) should be made before any SCSI transaction is initiated using the SPC chip.

A detailed description of the input parameters follows:

spcBaseAdrs
the address at which the CPU would access the lowest register of the SPC.

regOffset
the address offset (bytes) to access consecutive registers. (This must be a power of 2, for example, 1, 2, 4, etc.)

clkPeriod
the period in nanoseconds of the signal to the SPC clock input (only used for select command timeouts).

spcDataParity
the parity bit must be defined by one of the following constants, according to whether the input to SPC DP is GND, +5V, or a valid parity signal, respectively:

SPC_DATA_PARITY_LOW
SPC_DATA_PARITY_HIGH
SPC_DATA_PARITY_VALID

spcDmaBytesIn and *spcDmaBytesOut*
pointers to board-specific routines to handle DMA input and output. If these are NULL (0), SPC program transfer mode is used. DMA is possible only during SCSI data in / out phases. The interface to these DMA routines must be of the form:

```
STATUS xxDmaBytes{In, Out}
    (
    SCSI_PHYS_DEV  *pScsiPhysDev,  /* ptr to phys dev info    */
    UINT8          *pBuffer,       /* ptr to the data buffer  */
    int            bufLength       /* number of bytes to xfer */
    )
```

RETURNS A pointer to the SPC control structure, or NULL if memory is insufficient or parameters are invalid.

SEE ALSO **mb87030Lib**

---

*mb87030CtrlInit***( )**

NAME *mb87030CtrlInit***( )** – initialize a control structure for an MB87030 SPC

SYNOPSIS

```
STATUS mb87030CtrlInit
    (
    MB_87030_SCSI_CTRL *pSpc,             /* ptr to SPC struct             */
    int                scsiCtrlBusId,     /* SCSI bus ID of this SPC       */
    UINT               defaultSelTimeOut, /* default dev sel timeout (µsec) */
    int                scsiPriority       /* priority of task doing SCSI I/O*/
    )
```

DESCRIPTION This routine initializes an SPC control structure created by *mb87030CtrlCreate***( )**. It must be called before the SPC is used. This routine can be called more than once; however, it should be called only while there is no activity on the SCSI interface.

Before returning, this routine pulses RST (reset) on the SCSI bus, thus resetting all attached devices.

The input parameters are as follows:

pSpc
: a pointer to the **MB_87030_SCSI_CTRL** structure created with *mb87030CtrlCreate***( )**.

scsiCtrlBusId
: the SCSI bus ID of the SIOP, in the range 0 – 7. The ID is somewhat arbitrary; the value 7, or highest priority, is conventional.

defaultSelTimeOut
: the timeout, in microseconds, for selecting a SCSI device attached to this controller. The recommended value 0 specifies **SCSI_DEF_SELECT_TIMEOUT** (250 milliseconds). The maximum timeout possible is approximately 3 seconds. Values exceeding this revert to the maximum.

scsiPriority
the priority to which a task is set when performing a SCSI transaction. Valid priorities range from 0 to 255. Alternatively, the value -1 specifies that the priority should not be altered during SCSI transactions.

RETURNS OK, or ERROR if parameters are out of range.

SEE ALSO **mb87030Lib**

mb87030Show( )

NAME *mb87030Show*( ) – display the values of all readable MB87030 SPC registers

SYNOPSIS
```
STATUS mb87030Show
    (
    SCSI_CTRL  *pScsiCtrl  /* ptr to SCSI controller info */
    )
```

DESCRIPTION This routine displays the state of the SPC registers in a user-friendly manner. It is useful primarily for debugging.

EXAMPLE
```
-> mb87030Show
SCSI Bus ID: 7
SCTL (0x01): intsEnbl
SCMD (0x00): busRlease
TMOD (0x00): asyncMode
INTS (0x00):
PSNS (0x00): req0 ack0 atn0 sel0 bsy0 msg0 c_d0 i_o0
SSTS (0x05): noConIdle xferCnt=0 dregEmpty
SERR (0x00): noParErr
PCTL (0x00): bfIntDsbl phDataOut
MBC  (0x00): 0
XFER COUNT : 0x000000 =         0
```

RETURNS OK, or ERROR if *pScsiCtrl* and *pSysScsiCtrl* are both NULL.

SEE ALSO **mb87030Lib**

mbcattach( )

NAME

mbcattach( ) – publish the **mbc** network interface and initialize the driver

SYNOPSIS

```
STATUS mbcattach
    (
    int      unit,      /* unit number                              */
    void *   pEmBase,   /* ethernet module base address             */
    int      inum,      /* interrupt vector number                  */
    int      txBdNum,   /* number of transmit buffer descriptors    */
    int      rxBdNum,   /* number of receive buffer descriptors     */
    int      dmaParms,  /* DMA parameters                           */
    UINT8 *  bufBase    /* address of memory pool; NONE = malloc it */
    )
```

DESCRIPTION

The routine publishes the **mbc** interface by adding an **mbc** Interface Data Record (IDR) to the global network interface list.

The Ethernet controller uses buffer descriptors from an on-chip dual-ported RAM region, while the buffers are allocated in RAM external to the controller. The buffer memory pool can be allocated in a non-cacheable RAM region and passed as parameter *bufBase*. Otherwise *bufBase* is NULL and the buffer memory pool is allocated by the routine using *cacheDmaMalloc*( ). The driver uses this buffer pool to allocate the specified number of 1518-byte buffers for transmit, receive, and loaner pools.

The parameters *txBdNum* and *rxBdNum* specify the number of buffers to allocate for transmit and receive. If either of these parameters is NULL, the default value of 2 is used. The number of loaner buffers allocated is the lesser of *rxBdNum* and 16.

The on-chip dual ported RAM can only be partitioned so that the maximum receive and maximum transmit BDs are:

– Transmit BDs: 8, Receive BDs: 120

– Transmit BDs: 16, Receive BDs: 112

– Transmit BDs: 32, Receive BDs: 96

– Transmit BDs: 64, Receive BDs: 64

RETURNS

ERROR, if *unit* is out of range, device *unit* is already attached, or non-cacheable memory cannot be allocated; otherwise TRUE.

SEE ALSO

if_mbc, **ifLib**, *Motorola MC68EN302 User's Manual*

mbcIntr( )

NAME *mbcIntr*( ) – network interface interrupt handler

SYNOPSIS
```
void mbcIntr
    (
    int  unit  /* unit number */
    )
```

DESCRIPTION This routine is called at interrupt level. It handles work that requires minimal processing. Interrupt processing that is more extensive gets handled at task level. The network task, *netTask*( ), is provided for this function. Routines get added to the *netTask*( ) work queue via the *netJobAdd*( ) command.

RETURNS N / A

SEE ALSO **if_mbc**

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. 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**".

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.

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.

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.

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.

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.

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.

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.

SEE ALSO **memLib**, **smMemLib**

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.

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 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 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**, defined in **configAll.h**.

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**

---

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 **INCLUDE_SHOW_ROUTINES** is defined in **configAll.h**.

RETURNS N / A

SEE ALSO **memShow**

mkdir( )

NAME *mkdir*( ) – make a directory

SYNOPSIS
```
STATUS mkdir
    (
    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 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 **usrLib**, *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** members of the **tm** structure are ignored, and the original values of the other members 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 members 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.

ERRNO N / A

SEE ALSO **mmanPxLib**

mlockall( )

NAME *mlockall*( ) – lock all pages used by a process into memory (POSIX)

SYNOPSIS
```
int mlockall
    (
    int  flags
    )
```

DESCRIPTION This routine guarantees that all pages used by a process are memory resident. In VxWorks, the *flags* argument is ignored, since all pages are memory resident.

RETURNS 0 (OK) always.

ERRNO N / A

SEE ALSO **mmanPxLib**

mmuL64862DmaInit( )

NAME *mmuL64862DmaInit*( ) – initialize the L64862 I/O MMU DMA data structures (SPARC)

SYNOPSIS

```
STATUS mmuL64862DmaInit
    (
    void  *vrtBase,  /* First valid DMA virtual address */
    void  *vrtTop,   /* Last valid DMA virtual address  */
    UINT  range      /* range covered by I/O Page Table */
    )
```

DESCRIPTION This routine initializes the I/O MMU in the LSI Logic L64862 MBus to SBus Interface Chip (MS) for S-Bus DMA with the TI TMS390 SuperSPARC. It assumes **cacheLib** and **vmLib** have been initialized and that the TI TMS390 processor MMU is enabled. It initializes the I/O MMU to map all valid virtual addresses >= *vrtBase* and <= *vrtTop*. It is usually called as follows:

```
(void)mmuL64862DmaInit ((void *) LOCAL_MEM_LOCAL_ADRS,
                        (void *) (LOCAL_MEM_LOCAL_ADRS + LOCAL_MEM_SIZE - 1),
                        IOMMU_IOCR_RANGE);
```

RETURNS OK, or ERROR if the request cannot be satisfied.

SEE ALSO **mmuL64862Lib**

mmuSparcRomInit( )

NAME *mmuSparcRomInit*( ) – initialize the MMU for the ROM (SPARC)

SYNOPSIS

```
STATUS mmuSparcRomInit
    (
    int *  mmuTableAdrs,    /* address for the MMU tables          */
    int    mmuRomPhysAdrs,  /* ROM physical address                */
    int    romInitAdrs      /* address where romInit was linked in */
    )
```

DESCRIPTION This routine initializes the MMU when the system is booted. It should be called only from *romInit*( ). This routine is necessary because MMU libraries are not initialized by the boot code in **bootConfig**; they are initialized only in the VxWorks image in **usrConfig**. For consistency, the same **sysPhysMemDesc** is used by this routine as by *usrMmuInit*( ) in **usrConfig**.

RETURNS OK.

SEE ALSO **mmuSparcILib**

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 */
    )
```

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 possibles *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**

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.

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**

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. These routines are included automatically by defining **INCLUDE_SHOW_RTNS** in **configAll.h**.

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 to the specified message queue *mqdes* associated with the calling task. The real-time signal specified by *pNotification* is 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 will fail. A task can 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 prompts the user sequentially 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*

---

*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.

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

ERRNO

S_memLib_NOT_ENOUGH_MEMORY
unable to allocate memory for message queue and message buffers.

S_intLib_NOT_ISR_CALLABLE
called from an interrupt service routine.

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. Any task blocked on either a *msgQSend*( ) or *msgQReceive*( ) will be unblocked and receive an error from the call with **errno** set to **S_objLib_OBJECT_DELETED**. The *msgQId* parameter will no longer be a valid message queue ID.

RETURNS OK or ERROR.

ERRNO

S_objLib_OBJ_ID_ERROR
msgQId is invalid.

S_intLib_NOT_ISR_CALLABLE
called from an interrupt service routine.

SEE ALSO **msgQLib**, **msgQSmLib**

msgQInfoGet( )

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

SYNOPSIS
```
STATUS msgQInfoGet
    (
    MSG_Q_ID      msgQId,  /* message queue to query   */
    MSG_Q_INFO *  pInfo    /* where to return msg info */
    )
```

DESCRIPTION This routine gets information about the state and contents of a message queue. The parameter *pInfo* is a pointer to a structure of type **MSG_Q_INFO** defined in **msgQLib.h** as follows:

```
typedef struct              /* MSG_Q_INFO */
    {
    int     numMsgs;         /* OUT: number of messages queued          */
```

```
int     numTasks;         /* OUT: number of tasks waiting on msg q     */
int     sendTimeouts;     /* OUT: count of send timeouts               */
int     recvTimeouts;     /* OUT: count of receive timeouts            */
int     options;          /* OUT: options with which msg q was created */
int     maxMsgs;          /* OUT: max messages that can be queued      */
int     maxMsgLength;     /* OUT: max byte length of each message      */
int     taskIdListMax;    /* IN: max tasks to fill in taskIdList       */
int *   taskIdList;       /* PTR: array of task IDs waiting on msg q   */
int     msgListMax;       /* IN: max msgs to fill in msg lists         */
char ** msgPtrList;       /* PTR: array of msg ptrs queued to msg q    */
int *   msgLenList;       /* PTR: array of lengths of msgs             */
} MSG_Q_INFO;
```

If a message queue is empty, there may be tasks blocked on receiving. If a message queue is full, there may be tasks blocked on sending. This can be determined as follows:

– If *numMsgs* is 0, then *numTasks* indicates the number of tasks blocked on receiving.

– If *numMsgs* is equal to *maxMsgs*, then *numTasks* is the number of tasks blocked on sending.

– If *numMsgs* is greater than 0 but less than *maxMsgs*, then *numTasks* 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_objLib_OBJ_ID_ERROR**
msgQId is invalid.

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_objLib_OBJ_ID_ERROR**
msgQId is invalid.

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, even if the message has not been sent.

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_objLib_OBJ_ID_ERROR
msgQId is invalid.

S_objLib_OBJ_DELETED
the message queue was deleted while waiting to receive a message.

S_objLib_OBJ_UNAVAILABLE
timeout is set to **NO_WAIT**, and no messages are available.

S_objLib_OBJ_TIMEOUT
no messages were received in *timeout* ticks.

S_msgQLib_INVALID_MSG_LENGTH
nBytes is less than 0.

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 will immediately be delivered to the first waiting task. If no task is waiting to receive messages, the message is saved in the message queue.

The *timeout* parameter specifies the number of ticks to wait for free space if the message queue is full. The *timeout* parameter can also have the following special values:

NO_WAIT (0)
return immediately, even if the message has not been sent.

WAIT_FOREVER (-1)
never time out.

The *priority* parameter specifies the priority of the message being sent; 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 or ERROR.

ERRNO

S_objLib_OBJ_ID_ERROR
msgQId is invalid.

S_objLib_OBJ_DELETED
the message queue was deleted while waiting to a send message.

S_objLib_OBJ_UNAVAILABLE
timeout is set to **NO_WAIT**, and the queue is full.

S_objLib_OBJ_TIMEOUT
the queue is full for *timeout* ticks.

S_msgQLib_INVALID_MSG_LENGTH
nBytes is larger than the *maxMsgLength* set for the message queue.

S_msgQLib_NON_ZERO_TIMEOUT_AT_INT_LEVEL
called from an ISR, with *timeout* not set to **NO_WAIT**.

SEE ALSO **msgQLib**, **msgQSmLib**

---

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
```

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.

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 **INCLUDE_SHOW_ROUTINES** is defined in **configAll.h**.

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_smMemLib_NOT_ENOUGH_MEMORY**. This problem can be solved by incrementing the **SM_OBJ_MAX_MSG_Q** value in **configAll.h** and / or the shared memory objects dedicated memory size **SM_OBJ_MEM_SIZE** in **config.h**.

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_smMemLib_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**

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**

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).

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**, *taskDelay*( )

ncr5390CtrlCreate( )

NAME

ncr5390CtrlCreate( ) – create a control structure for an NCR 53C90 ASC

SYNOPSIS

```
NCR_5390_SCSI_CTRL *ncr5390CtrlCreate
    (
    UINT8    *baseAdrs,      /* base address of ASC                  */
    int      regOffset,      /* addr offset between consecutive regs. */
    UINT     clkPeriod,      /* period of controller clock (nsec)    */
    FUNCPTR  ascDmaBytesIn,  /* SCSI DMA input function              */
    FUNCPTR  ascDmaBytesOut  /* SCSI DMA output function             */
    )
```

DESCRIPTION

This routine creates a data structure that must exist before the ASC chip can be used. This routine must be called exactly once for a specified ASC, and must be the first routine called, since it calloc's a structure needed by all other routines in the library.

The input parameters are as follows:

baseAdrs
the address at which the CPU would access the lowest register of the ASC.

regOffset
the address offset (bytes) to access consecutive registers. (This must be a power of 2, for example, 1, 2, 4, etc.)

clkPeriod
the period, in nanoseconds, of the signal to the ASC clock input (used only for select command timeouts).

ascDmaBytesIn and *ascDmaBytesOut*
board-specific parameters to handle DMA input and output. If these are NULL (0), ASC program transfer mode is used. DMA is possible only during SCSI data in / out phases. The interface to these DMA routines must be of the form:

```
STATUS xxDmaBytes{In, Out}
    (
    SCSI_PHYS_DEV  *pScsiPhysDev,  /* ptr to phys dev info    */
    UINT8          *pBuffer,       /* ptr to the data buffer  */
    int            bufLength       /* number of bytes to xfer */
    )
```

RETURNS

A pointer to an **NCR_5390_SCSI_CTRL** structure, or NULL if memory is insufficient or the parameters are invalid.

SEE ALSO

ncr5390Lib1

ncr5390CtrlCreateScsi2( )

NAME

ncr5390CtrlCreateScsi2( ) – create a control structure for an NCR 53C90 ASC

SYNOPSIS

```
NCR_5390_SCSI_CTRL *ncr5390CtrlCreateScsi2
    (
    UINT8*   baseAdrs,            /* base address of ASC               */
    int      regOffset,           /* offset between consecutive regs.  */
    UINT     clkPeriod,           /* period of controller clock (nsec) */
    UINT     sysScsiDmaMaxBytes,  /* maximum byte count using DMA      */
    FUNCPTR  sysScsiDmaStart,     /* function to start SCSI DMA xfer   */
    FUNCPTR  sysScsiDmaAbort,     /* function to abort SCSI DMA xfer   */
    int      sysScsiDmaArg        /* argument to pass to above funcs.  */
    )
```

DESCRIPTION

This routine creates a data structure that must exist before the ASC chip can be used. This routine must be called exactly once for a specified ASC, and must be the first routine called, since it calloc's a structure needed by all other routines in the library.

The input parameters are as follows:

baseAdrs
the address at which the CPU would access the lowest register of the ASC.

regOffset
the address offset (bytes) to access consecutive registers.

clkPeriod
the period, in nanoseconds, of the signal to the ASC clock input.

sysScsiDmaMaxBytes, *sysScsiDmaStart*, *sysScsiDmaAbort*, and *sysScsiDmaArg*
board-specific routines to handle DMA transfers to and from the ASC; if the maximum DMA byte count is zero, programmed I/O is used. Otherwise, non-NULL function pointers to DMA start and abort routines must be provided. The specified argument is passed to these routines when they are called; it may be used to identify the DMA channel to use, for example. The interface to these DMA routines must be of the form:

```
STATUS xxDmaStart (arg, pBuffer, bufLength, direction)
    int arg;                    /* call-back argument           */
    UINT8 *pBuffer;             /* ptr to the data buffer       */
    UINT bufLength;             /* number of bytes to xfer      */
    int direction;              /* 0 = SCSI->mem, 1 = mem->SCSI */
STATUS xxDmaAbort (arg)
    int arg;                    /* call-back argument */
```

Implementation details for the DMA routines can be found in the specific DMA driver for that board.

NOTE: If there is no DMA interface, synchronous transfers are not supported. This is a limitation of the NCR5390 hardware.

RETURNS A pointer to an **NCR_5390_SCSI_CTRL** structure, or NULL if memory is insufficient or the parameters are invalid.

SEE ALSO **ncr5390Lib2**

*ncr5390CtrlInit***( )**

NAME *ncr5390CtrlInit***( )** – initialize the user-specified fields in an ASC structure

SYNOPSIS

```
STATUS ncr5390CtrlInit
    (
    int   *pAsc,              /* ptr to ASC info                          */
    int   scsiCtrlBusId,      /* SCSI bus ID of this ASC                  */
    UINT  defaultSelTimeOut,  /* default dev. select timeout (microsec) */
    int   scsiPriority        /* priority of task when doing SCSI I/O   */
    )
```

DESCRIPTION This routine initializes an ASC structure, after the structure is created with *ncr5390CtrlCreate***( )**. This structure must be initialized before the ASC can be used. It may be called more than once; however, it should be called only while there is no activity on the SCSI interface.

Before returning, this routine pulses RST (reset) on the SCSI bus, thus resetting all attached devices.

The input parameters are as follows:

pAsc
a pointer to the **NCR5390_SCSI_CTRL** structure created with *ncr5390CtrlCreate***( )**.

scsiCtrlBusId
the SCSI bus ID of the ASC, in the range 0 – 7. The ID is somewhat arbitrary; the value 7, or highest priority, is conventional.

defaultSelTimeOut
the timeout, in microseconds, for selecting a SCSI device attached to this controller. This value is used as a default if no timeout is specified in *scsiPhysDevCreate***( )**. The recommended value zero (0) specifies **SCSI_DEF_SELECT_TIMEOUT** (250 millisec).

The maximum timeout possible is approximately 2 seconds. Values exceeding this revert to the maximum.

scsiPriority
the priority to which a task is set when performing a SCSI transaction. Valid priorities are 0 to 255. Alternatively, the value -1 specifies that the priority should not be altered during SCSI transactions.

RETURNS OK, or ERROR if a parameter is out of range.

SEE ALSO **ncr5390Lib**, *scsiPhysDevCreate***( )**,

*ncr5390Show***( )**

NAME *ncr5390Show***( )** – display the values of all readable NCR5390 chip registers

SYNOPSIS
```
int ncr5390Show
    (
    int  *pScsiCtrl  /* ptr to SCSI controller info */
    )
```

DESCRIPTION This routine displays the state of the ASC registers in a user-friendly manner. It is useful primarily for debugging. It should not be invoked while another running process is accessing the SCSI controller.

EXAMPLE
```
-> ncr5390Show
REG #00 (Own ID         ) = 0x07
REG #01 (Control        ) = 0x00
REG #02 (Timeout Period ) = 0x20
REG #03 (Sectors        ) = 0x00
REG #04 (Heads          ) = 0x00
REG #05 (Cylinders MSB  ) = 0x00
REG #06 (Cylinders LSB  ) = 0x00
REG #07 (Log. Addr. MSB ) = 0x00
REG #08 (Log. Addr. 2SB ) = 0x00
REG #09 (Log. Addr. 3SB ) = 0x00
REG #0a (Log. Addr. LSB ) = 0x00
REG #0b (Sector Number  ) = 0x00
REG #0c (Head Number    ) = 0x00
REG #0d (Cyl. Number MSB) = 0x00
REG #0e (Cyl. Number LSB) = 0x00
REG #0f (Target LUN     ) = 0x00
REG #10 (Command Phase  ) = 0x00
```

```
REG #11 (Synch. Transfer) = 0x00
REG #12 (Xfer Count MSB ) = 0x00
REG #13 (Xfer Count 2SB ) = 0x00
REG #14 (Xfer Count LSB ) = 0x00
REG #15 (Destination ID ) = 0x03
REG #16 (Source ID      ) = 0x00
REG #17 (SCSI Status    ) = 0x42
REG #18 (Command        ) = 0x07
```

RETURNS OK, or ERROR if *pScsiCtrl* and *pSysScsiCtrl* are both NULL.

SEE ALSO **ncr5390Lib**

---

*ncr710CtrlCreate***( )**

NAME *ncr710CtrlCreate***( )** – create a control structure for an NCR 53C710 SIOP

SYNOPSIS

```
NCR_710_SCSI_CTRL *ncr710CtrlCreate
    (
    UINT8  *baseAdrs,  /* base address of the SIOP          */
    UINT   freqValue   /* clock controller period (nsec*100) */
    )
```

DESCRIPTION This routine creates an SIOP data structure and must be called before using an SIOP chip. It should be called once and only once for a specified SIOP. Since it allocates memory for a structure needed by all routines in **ncr710Lib**, it must be called before any other routines in the library. After calling this routine, *ncr710CtrlInit***( )** should be called at least once before any SCSI transactions are initiated using the SIOP.

A detailed description of the input parameters follows:

baseAdrs
the address at which the CPU accesses the lowest register of the SIOP.

freqValue
the value at the SIOP SCSI clock input. This is used to determine the clock period for the SCSI core of the chip and the synchronous divider value for synchronous transfer. It is important to have the right timing on the SCSI bus. The *freqValue* parameter is defined as the SCSI clock input value, in nanoseconds, multiplied by 100. Several *freqValue* constants are defined in **ncr710.h** as follows:

```
NCR710_1667MHZ  5998    /* 16.67Mhz chip */
NCR710_20MHZ    5000    /* 20Mhz chip    */
NCR710_25MHZ    4000    /* 25Mhz chip    */
```

```
NCR710_3750MHZ  2666    /* 37.50Mhz chip */
NCR710_40MHZ    2500    /* 40Mhz chip    */
NCR710_50MHZ    2000    /* 50Mhz chip    */
NCR710_66MHZ    1515    /* 66Mhz chip    */
NCR710_6666MHZ  1500    /* 66.66Mhz chip */
```

RETURNS A pointer to the **NCR_710_SCSI_CTRL** structure, or NULL if memory is insufficient or parameters are invalid.

SEE ALSO **ncr710Lib**

*ncr710CtrlCreateScsi2***( )**

NAME *ncr710CtrlCreateScsi2***( )** – create a control structure for the NCR 53C710 SIOP

SYNOPSIS

```
NCR_710_SCSI_CTRL *ncr710CtrlCreateScsi2
    (
    UINT8  *baseAdrs,  /* base address of the SIOP          */
    UINT   clkPeriod   /* clock controller period (nsec*100) */
    )
```

DESCRIPTION This routine creates an SIOP data structure and must be called before using an SIOP chip. It must be called exactly once for a specified SIOP controller. Since it allocates memory for a structure needed by all routines in **ncr710Lib**, it must be called before any other routines in the library. After calling this routine, *ncr710CtrlInitScsi2***( )** must be called at least once before any SCSI transactions are initiated using the SIOP.

A detailed description of the input parameters follows:

baseAdrs
the address at which the CPU accesses the lowest (SCNTL0/SIEN) register of the SIOP.

clkPeriod
the period of the SIOP SCSI clock input, in nanoseconds, multiplied by 100. This is used to determine the clock period for the SCSI core of the chip and affects the timing of both asynchronous and synchronous transfers. Several commonly used values are defined in **ncr710.h** as follows:

```
NCR710_1667MHZ  6000    /* 16.67Mhz chip */
NCR710_20MHZ    5000    /* 20Mhz chip    */
NCR710_25MHZ    4000    /* 25Mhz chip    */
NCR710_3750MHZ  2667    /* 37.50Mhz chip */
NCR710_40MHZ    2500    /* 40Mhz chip    */
```

```
NCR710_50MHZ    2000    /* 50Mhz chip    */
NCR710_66MHZ    1515    /* 66Mhz chip    */
NCR710_6666MHZ  1500    /* 66.66Mhz chip */
```


RETURNS A pointer to the **NCR_710_SCSI_CTRL** structure, or NULL if memory is unavailable or there are invalid parameters.

SEE ALSO **ncr710Lib2**

---

*ncr710CtrlInit***( )**

NAME *ncr710CtrlInit***( )** – initialize a control structure for an NCR 53C710 SIOP

SYNOPSIS

```
STATUS ncr710CtrlInit
    (
    NCR_710_SCSI_CTRL *pSiop,        /* ptr to SIOP struct               */
    int               scsiCtrlBusId, /* SCSI bus ID of this SIOP         */
    int               scsiPriority   /* task priority when doing SCSI I/O */
    )
```

DESCRIPTION This routine initializes an SIOP structure, after the structure is created with *ncr710CtrlCreate***( )**. This structure must be initialized before the SIOP can be used. It may be called more than once; however, it should be called only while there is no activity on the SCSI interface.

Before returning, this routine pulses RST (reset) on the SCSI bus, thus resetting all attached devices.

The input parameters are as follows:

pSiop
a pointer to the **NCR_710_SCSI_CTRL** structure created with *ncr710CtrlCreate***( )**.

scsiCtrlBusId
the SCSI bus ID of the SIOP, in the range 0 – 7. The ID is somewhat arbitrary; the value 7, or highest priority, is conventional.

scsiPriority
the priority to which a task is set when performing a SCSI transaction. Valid priorities are 0 to 255. Alternatively, the value -1 specifies that the priority should not be altered during SCSI transactions.

RETURNS OK, or ERROR if parameters are out of range.

SEE ALSO **ncr710Lib**

*ncr710CtrlInitScsi2***( )**

NAME

*ncr710CtrlInitScsi2***( )** – initialize a control structure for the NCR 53C710 SIOP

SYNOPSIS

```
STATUS ncr710CtrlInitScsi2
    (
    NCR_710_SCSI_CTRL * pSiop,        /* ptr to SIOP struct               */
    int                 scsiCtrlBusId, /* SCSI bus ID of this SIOP         */
    int                 scsiPriority   /* task priority when doing SCSI I/O */
    )
```

DESCRIPTION

This routine initializes an SIOP structure after the structure is created with *ncr710CtrlCreateScsi2***( )**. This structure must be initialized before the SIOP can be used. It may be called more than once if needed; however, it must only be called while there is no activity on the SCSI interface.

A detailed description of the input parameters follows:

pSiop
: a pointer to the **NCR_710_SCSI_CTRL** structure created with *ncr710CtrlCreateScsi2***( )**.

scsiCtrlBusId
: the SCSI bus ID of the SIOP. Its value is somewhat arbitrary: seven (7), or highest priority, is conventional. The value must be in the range 0 – 7.

scsiPriority
: this parameter is ignored. All SCSI I/O is now done in the context of the SCSI manager task; if necessary, the priority of the manager task may be changed using *taskPrioritySet***( )** or by setting the value of the global variable **ncr710ScsiTaskPriority** before calling *ncr710CtrlCreateScsi2***( )**.

RETURNS

OK, or ERROR if the parameters are out of range.

SEE ALSO

ncr710Lib2, *ncr710CtrlCreateScsi2***( )**

*ncr710SetHwRegister***( )**

NAME

*ncr710SetHwRegister***( )** – set hardware-dependent registers for the NCR 53C710 SIOP

SYNOPSIS

```
STATUS ncr710SetHwRegister
    (
    SIOP            *pSiop,   /* pointer to SIOP info          */
```

```
    NCR710_HW_REGS  *pHwRegs  /* pointer to NCR710_HW_REGS info */
    )
```


DESCRIPTION

This routine sets up the registers used in the hardware implementation of the chip. Typically, this routine is called by the *sysScsiInit***( )** routine from the board support package.

The input parameters are as follows:

pSiop
a pointer to the **NCR_710_SCSI_CTRL** structure created with *ncr710CtrlCreate***( )**.

pHwRegs
a pointer to a **NCR710_HW_REGS** structure that is filled with the logical values 0 or 1 for each bit of each register described below.

This routine includes only the bit registers that can be used to modify the behavior of the chip. The default configuration used during *ncr710CtlrCreate***( )** and *ncr710CrtlInit***( )** is {0,0,0,0,1,0,0,0,0,0,0,0,0,1,0}.

```
typedef struct
    {
    int ctest4Bit7;    /* host bus multiplex mode */
    int ctest7Bit7;    /* disable/enable burst cache capability */
    int ctest7Bit6;    /* snoop control bit1 */
    int ctest7Bit5;    /* snoop control bit0 */
    int ctest7Bit1;    /* invert tt1 pin (sync bus host mode only) */
    int ctest7Bit0;    /* enable differential SCSI bus capability */
    int ctest8Bit0;    /* set snoop pins mode */
    int dmodeBit7;     /* burst length transfer bit 1 */
    int dmodeBit6;     /* burst length transfer bit 0 */
    int dmodeBit5;     /* function code bit FC2 */
    int dmodeBit4;     /* function code bit FC1 */
    int dmodeBit3;     /* program data bit (FC0) */
    int dmodeBit1;     /* user-programmable transfer type */
    int dcntlBit5;     /* enable ACK pin */
    int dcntlBit1;     /* enable fast arbitration on host port */
    } NCR710_HW_REGS;
```

For a more detailed description of the register bits, see the *NCR 53C710 SCSI I/O Processor Programming Guide.*

NOTE: Because this routine writes to the NCR 53C710 chip registers, it cannot be used when there is any SCSI bus activity.

RETURNS

OK, or ERROR if an input parameter is NULL.

SEE ALSO

ncr710Lib, *ncr710CtlrCreate***( )**, *NCR 53C710 SCSI I/O Processor Programming Guide*

*ncr710SetHwRegisterScsi2***( )**

NAME

*ncr710SetHwRegisterScsi2***( )** – set hardware-dependent registers for the NCR 53C710

SYNOPSIS

```
STATUS ncr710SetHwRegisterScsi2
    (
    SIOP            *pSiop,   /* pointer to SIOP info             */
    NCR710_HW_REGS  *pHwRegs  /* pointer to a NCR710_HW_REGS info */
    )
```

DESCRIPTION

This routine sets up the registers used in the hardware implementation of the chip. Typically, this routine is called by the *sysScsiInit***( )** routine from the BSP.

The input parameters are as follows:

pSiop
: a pointer to the **NCR_710_SCSI_CTRL** structure created with *ncr710CtrlCreateScsi2***( )**.

pHwRegs
: a pointer to a **NCR710_HW_REGS** structure that is filled with the logical values 0 or 1 for each bit of each register described below.

This routine includes only the bit registers that can be used to modify the behavior of the chip. The default configuration used during *ncr710CtlrCreateScsi2***( )** and *ncr710CrtlInitScsi2***( )** is {0,0,0,0,1,0,0,0,0,0,0,0,1,0}.

```
typedef struct
    {
    int ctest4Bit7;    /* Host bus multiplex mode */
    int ctest7Bit7;    /* Disable/enable burst cache capability */
    int ctest7Bit6;    /* Snoop control bit1 */
    int ctest7Bit5;    /* Snoop control bit0 */
    int ctest7Bit1;    /* invert tt1 pin (sync bus host mode only) */
    int ctest7Bit0;    /* enable differential scsi bus capability*/
    int ctest8Bit0;    /* Set snoop pins mode */
    int dmodeBit7;     /* Burst Length transfer bit 1 */
    int dmodeBit6;     /* Burst Length transfer bit 0 */
    int dmodeBit5;     /* Function code bit FC2 */
    int dmodeBit4;     /* Function code bit FC1 */
    int dmodeBit3;     /* Program data bit (FC0) */
    int dmodeBit1;     /* user  programmable transfer type */
    int dcntlBit5;     /* Enable Ack pin */
    int dcntlBit1;     /* Enable fast arbitration on host port */
    } NCR710_HW_REGS;
```

For a more detailed explanation of the register bits, refer to the *NCR 53C710 SCSI I/O Processor Programming Guide.*

NOTE: Because this routine writes to the chip registers, it cannot be used if there is any SCSI bus activity.

RETURNS OK, or ERROR if any input parameter is NULL.

SEE ALSO **ncr710Lib2**, *ncr710CtrlCreateScsi2***( )**, *NCR 53C710 SCSI I/O Processor Programming Guide*

*ncr710Show***( )**

NAME *ncr710Show***( )** – display the values of all readable NCR 53C710 SIOP registers

SYNOPSIS

```
STATUS ncr710Show
    (
    SCSI_CTRL  *pScsiCtrl  /* ptr to SCSI controller info */
    )
```

DESCRIPTION This routine displays the state of the NCR 53C710 SIOP registers in a user-friendly manner. It is useful primarily for debugging. The input parameter is the pointer to the SIOP information structure returned by the *ncr710CtrlCreate***( )** call.

NOTE: The only readable register during a script execution is **Istat**. If this routine is used during the execution of a SCSI command, the result could be unpredictable.

EXAMPLE

```
-> ncr710Show
NCR710 Registers
----------------
0xfff47000: Sien    = 0xa5 Sdid    = 0x00 Scntl1  = 0x00 Scntl0  = 0x04
0xfff47004: Socl    = 0x00 Sodl    = 0x00 Sxfer   = 0x80 Scid    = 0x80
0xfff47008: Sbcl    = 0x00 Sbdl    = 0x00 Sidl    = 0x00 Sfbr    = 0x00
0xfff4700c: Sstat2  = 0x00 Sstat1  = 0x00 Sstat0  = 0x00 Dstat   = 0x80
0xfff47010: Dsa     = 0x00000000
0xfff47014: Ctest3  = ???? Ctest2  = 0x21 Ctest1  = 0xf0 Ctest0  = 0x00
0xfff47018: Ctest7  = 0x32 Ctest6  = ???? Ctest5  = 0x00 Ctest4  = 0x00
0xfff4701c: Temp    = 0x00000000
0xfff47020: Lcrc    = 0x00 Ctest8  = 0x00 Istat   = 0x00 Dfifo   = 0x00
0xfff47024: Dcmd/Ddc= 0x50000000
0xfff47028: Dnad    = 0x00066144
0xfff4702c: Dsp     = 0x00066144
0xfff47030: Dsps    = 0x00066174
0xfff47037: Scratch3= 0x00 Scratch2= 0x00 Scratch1= 0x00 Scratch0= 0x0a
0xfff47038: Dcntl   = 0x21 Dwt     = 0x00 Dien    = 0x37 Dmode   = 0x01
0xfff4703c: Adder   = 0x000cc2b8
```

RETURNS OK, or ERROR if *pScsiCtrl* and *pSysScsiCtrl* are both NULL.

SEE ALSO **ncr710Lib**, *ncr710CtrlCreate***( )**

*ncr710ShowScsi2***( )**

NAME *ncr710ShowScsi2***( )** – display the values of all readable NCR 53C710 SIOP registers

SYNOPSIS

```
STATUS ncr710ShowScsi2
    (
    SCSI_CTRL  *pScsiCtrl  /* ptr to SCSI controller info */
    )
```

DESCRIPTION This routine displays the state of the NCR 53C710 SIOP registers in a user-friendly way. It is primarily used for debugging. The input parameter is the pointer to the SIOP information structure returned by the *ncr710CtrlCreateScsi2***( )** call.

NOTE: The only readable register during a script execution is the **Istat** register. If you use this routine during the execution of a SCSI command, the result could be unpredictable.

EXAMPLE

```
-> ncr710Show
NCR710 Registers
----------------
0xfff47000: Sien    = 0xa5 Sdid    = 0x00 Scntl1  = 0x00 Scntl0  = 0x04
0xfff47004: Socl    = 0x00 Sodl    = 0x00 Sxfer   = 0x80 Scid    = 0x80
0xfff47008: Sbcl    = 0x00 Sbdl    = 0x00 Sidl    = 0x00 Sfbr    = 0x00
0xfff4700c: Sstat2  = 0x00 Sstat1  = 0x00 Sstat0  = 0x00 Dstat   = 0x80
0xfff47010: Dsa     = 0x00000000
0xfff47014: Ctest3  = ???? Ctest2  = 0x21 Ctest1  = 0xf0 Ctest0  = 0x00
0xfff47018: Ctest7  = 0x32 Ctest6  = ???? Ctest5  = 0x00 Ctest4  = 0x00
0xfff4701c: Temp    = 0x00000000
0xfff47020: Lcrc    = 0x00 Ctest8  = 0x00 Istat   = 0x00 Dfifo   = 0x00
0xfff47024: Dcmd/Ddc= 0x50000000
0xfff47028: Dnad    = 0x00066144
0xfff4702c: Dsp     = 0x00066144
0xfff47030: Dsps    = 0x00066174
0xfff47037: Scratch3= 0x00 Scratch2= 0x00 Scratch1= 0x00 Scratch0= 0x0a
0xfff47038: Dcntl   = 0x21 Dwt     = 0x00 Dien    = 0x37 Dmode   = 0x01
0xfff4703c: Adder   = 0x000cc2b8
```

RETURNS OK, or ERROR if *pScsiCtrl* and *pSysScsiCtrl* are both NULL.

SEE ALSO **ncr710Lib2**, *ncr710CtrlCreateScsi2***( )**

ncr810CtrlCreate( )

NAME

ncr810CtrlCreate( ) – create a control structure for the NCR 53C8xx SIOP

SYNOPSIS

```
NCR_810_SCSI_CTRL *ncr810CtrlCreate
    (
    UINT8   *baseAdrs,  /* base address of the SIOP          */
    UINT    clkPeriod,  /* clock controller period (nsec*100) */
    UINT16  devType     /* NCR8XX SCSI device type           */
    )
```

DESCRIPTION

This routine creates an SIOP data structure and must be called before using an SIOP chip. It must be called exactly once for a specified SIOP controller. Since it allocates memory for a structure needed by all routines in **ncr810Lib**, it must be called before any other routines in the library. After calling this routine, *ncr810CtrlInit*( ) must be called at least once before any SCSI transactions are initiated using the SIOP.

A detailed description of the input parameters follows:

baseAdrs
the address at which the CPU accesses the lowest (SCNTL0/SIEN) register of the SIOP.

clkPeriod
the period of the SIOP SCSI clock input, in nanoseconds, multiplied by 100. This is used to determine the clock period for the SCSI core of the chip and affects the timing of both asynchronous and synchronous transfers. Several commonly-used values are defined in **ncr810.h** as follows:

```
NCR810_1667MHZ  6000    /* 16.67Mhz chip */
NCR810_20MHZ    5000    /* 20Mhz chip    */
NCR810_25MHZ    4000    /* 25Mhz chip    */
NCR810_3750MHZ  2667    /* 37.50Mhz chip */
NCR810_40MHZ    2500    /* 40Mhz chip    */
NCR810_50MHZ    2000    /* 50Mhz chip    */
NCR810_66MHZ    1515    /* 66Mhz chip    */
NCR810_6666MHZ  1500    /* 66.66Mhz chip */
```

devType
the specific NCR 8xx device type. Current device types are defined in the header file **ncr810.h**.

RETURNS

A pointer to the **NCR_810_SCSI_CTRL** structure, or NULL if memory is unavailable or there are invalid parameters.

SEE ALSO

ncr810Lib

ncr810CtrlInit( )

NAME

ncr810CtrlInit( ) – initialize a control structure for the NCR 53C8xx SIOP

SYNOPSIS

```
STATUS ncr810CtrlInit
    (
    NCR_810_SCSI_CTRL  *pSiop,        /* ptr to SIOP struct       */
    int                scsiCtrlBusId  /* SCSI bus ID of this SIOP */
    )
```

DESCRIPTION

This routine initializes an SIOP structure, after the structure is created with *ncr810CtrlCreate*( ). This structure must be initialized before the SIOP can be used. It may be called more than once if needed; however, it must only be called while there is no activity on the SCSI interface.

A detailed description of the input parameters follows:

pSiop
: a pointer to the **NCR_810_SCSI_CTRL** structure created with *ncr810CtrlCreate*( ).

scsiCtrlBusId
: the SCSI bus ID of the SIOP. Its value is somewhat arbitrary: seven (7), or highest priority, is conventional. The value must be in the range 0 – 7.

RETURNS

OK, or ERROR if parameters are out of range.

SEE ALSO

ncr810Lib

ncr810SetHwRegister( )

NAME

ncr810SetHwRegister( ) – set hardware-dependent registers for the NCR 53C8xx SIOP

SYNOPSIS

```
STATUS ncr810SetHwRegister
    (
    SIOP            *pSiop,   /* pointer to SIOP info             */
    NCR810_HW_REGS  *pHwRegs  /* pointer to a NCR810_HW_REGS info */
    )
```

DESCRIPTION

This routine sets up the registers used in the hardware implementation of the chip. Typically, this routine is called by the *sysScsiInit*( ) routine from the BSP.

The input parameters are as follows:

pSiop
a pointer to the **NCR_810_SCSI_CTRL** structure created with *ncr810CtrlCreate***( )**.

pHwRegs
a pointer to a **NCR810_HW_REGS** structure that is filled with the logical values 0 or 1 for each bit of each register described below.

This routine includes only the bit registers that can be used to modify the behavior of the chip. The default configuration used during *ncr810CtlrCreate***( )** and *ncr810CrtlInit***( )** is {0,0,0,0,0,1,0,0,0,0}.

```
typedef struct
  {
  int stest1Bit7;             /* Disable external SCSI clock  */
  int stest2Bit7;             /* SCSI control enable          */
  int stest2Bit5;             /* Enable differential SCSI bus */
  int stest2Bit2;             /* Always WIDE SCSI             */
  int stest2Bit1;             /* Extend SREQ/SACK filtering   */
  int stest3Bit7;             /* TolerANT enable              */
  int dmodeBit7;              /* Burst Length transfer bit 1  */
  int dmodeBit6;              /* Burst Length transfer bit 0  */
  int dmodeBit5;              /* Source I/O memory enable     */
  int dmodeBit4;              /* Destination I/O memory enable*/
  int scntl1Bit7;             /* Slow cable mode              */
  } NCR810_HW_REGS;
```

For a more detailed explanation of the register bits, see the appropriate NCR 53C8xx data manuals.

NOTE: Because this routine writes to the NCR 53C8xx chip registers, it cannot be used when there is any SCSI bus activity.

RETURNS OK, or ERROR if any input parameter is NULL

SEE ALSO **ncr810Lib**, **ncr810.h**, *ncr810CtlrCreate***( )**

*ncr810Show***( )**

NAME *ncr810Show***( )** – display values of all readable NCR 53C8xx SIOP registers

SYNOPSIS

```
STATUS ncr810Show
    (
    SCSI_CTRL  *pScsiCtrl  /* ptr to SCSI controller info */
    )
```

DESCRIPTION This routine displays the state of the SIOP registers in a user-friendly way. It is useful primarily for debugging. The input parameter is the pointer to the SIOP information structure returned by the *ncr810CtrlCreate***( )** call.

NOTE: The only readable register during a script execution is the **Istat** register. If you use this routine during the execution of a SCSI command, the result could be unpredictable.

EXAMPLE

```
-> ncr810Show
NCR810 Registers
----------------
0xfff47000: Sien    = 0xa5 Sdid    = 0x00 Scntl1  = 0x00 Scntl0  = 0x04
0xfff47004: Socl    = 0x00 Sodl    = 0x00 Sxfer   = 0x80 Scid    = 0x80
0xfff47008: Sbcl    = 0x00 Sbdl    = 0x00 Sidl    = 0x00 Sfbr    = 0x00
0xfff4700c: Sstat2  = 0x00 Sstat1  = 0x00 Sstat0  = 0x00 Dstat   = 0x80
0xfff47010: Dsa     = 0x00000000
0xfff47014: Ctest3  = ???? Ctest2  = 0x21 Ctest1  = 0xf0 Ctest0  = 0x00
0xfff47018: Ctest7  = 0x32 Ctest6  = ???? Ctest5  = 0x00 Ctest4  = 0x00
0xfff4701c: Temp    = 0x00000000
0xfff47020: Lcrc    = 0x00 Ctest8  = 0x00 Istat   = 0x00 Dfifo   = 0x00
0xfff47024: Dcmd/Ddc= 0x50000000
0xfff47028: Dnad    = 0x00066144
0xfff4702c: Dsp     = 0x00066144
0xfff47030: Dsps    = 0x00066174
0xfff47037: Scratch3= 0x00 Scratch2= 0x00 Scratch1= 0x00 Scratch0= 0x0a
0xfff47038: Dcntl   = 0x21 Dwt     = 0x00 Dien    = 0x37 Dmode   = 0x01
0xfff4703c: Adder   = 0x000cc2b8
```

RETURNS OK, or ERROR if *pScsiCtrl* and *pSysScsiCtrl* are both NULL.

SEE ALSO **ncr810Lib**, *ncr810CtrlCreate***( )**

*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***( )**

---

*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_NETWORK** is defined in **configAll.h**.

RETURNS OK or ERROR.

SEE ALSO **netDrv**

---

*netHelp***( )**

NAME *netHelp***( )** – print a synopsis of network routines

SYNOPSIS

```
void netHelp (void)
```

DESCRIPTION This command prints the following brief synopsis of network facilities that are typically called from the shell:

```
hostAdd      "hostname","inetaddr" - add a host to remote host table;
                                     "inetaddr" must be in standard
                                     Internet address format e.g. "90.0.0.4"
hostShow                           - print current remote host table
netDevCreate "devname","hostname",protocol
                                   - create an I/O device to access
                                     files on the specified host
                                     (protocol 0=rsh, 1=ftp)
routeAdd     "destaddr","gateaddr" - add route to route table
routeDelete  "destaddr","gateaddr" - delete route from route table
routeShow                          - print current route table
iam          "usr"[,"passwd"]      - specify the user name by which
                                     you will be known to remote
                                     hosts (and optional password)
whoami                             - print the current remote ID
rlogin       "host"                - log in to a remote host;
                                     "host" can be inet address or
                                     host name in remote host table
ifShow       ["ifname"]            - show info about network interfaces
inetstatShow                       - show all Internet protocol sockets
tcpstatShow                        - show statistics for TCP
udpstatShow                        - show statistics for UDP
ipstatShow                         - show statistics for IP
icmpstatShow                       - show statistics for ICMP
arptabShow                         - show a list of known ARP entries
mbufShow                           - show mbuf statistics
EXAMPLE:  -> hostAdd "wrs", "90.0.0.2"
          -> netDevCreate "wrs:", "wrs", 0
          -> iam "fred"
          -> copy <wrs:/etc/passwd   /* copy file from host "wrs" */
          -> rlogin "wrs"            /* rlogin to host "wrs"      */
```

RETURNS N / A

SEE ALSO **usrLib**, *VxWorks Programmer's Guide: Target Shell*

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_NETWORK** is defined in **configAll.h**.

RETURNS OK, or ERROR if network support cannot be initialized.

SEE ALSO **netLib**, **usrConfig**, *netTask***( )**

*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_SHOW** is defined in **configAll.h**.

RETURNS N / A

SEE ALSO **netShow**

*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. The task is spawned by *netLibInit***( )**.

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.

RETURNS N / A

SEE ALSO **netLib**, *netLibInit***( )**

nextproc_error( )

NAME *nextproc_error*( ) – indicate that a **nextproc** operation encountered an error

SYNOPSIS

```
void nextproc_error
    (
    SNMP_PKT_T *  pPkt,      /* internal representation of the snmp packet */
    VB_T *        pVarBind,  /* var bind being processed                   */
    INT_32_T      error      /* error code                                 */
    )
```

DESCRIPTION This routine is called when **nextproc** encounters an error and cannot retreive a next instance for the requested variable binding.

RETURNS N / A

SEE ALSO **snmpProcLib**

nextproc_good( )

NAME *nextproc_good*( ) – indicate successful completion of a **nextproc** procedure

SYNOPSIS

```
void nextproc_good
    (
    SNMP_PKT_T *  pPkt,     /* internal representation of the snmp packet */
    VB_T *        pVarBind  /* var bind being processed                   */
    )
```

DESCRIPTION This routine indicates that **nextproc** for the specified variable binding has completed successfully.

RETURNS N / A

SEE ALSO **snmpProcLib**

nextproc_next_instance( )

NAME *nextproc_next_instance*( ) – install instance part of next instance

SYNOPSIS

```
void nextproc_next_instance
    (
    SNMP_PKT_T *  pPkt,      /* internal representation of the snmp packet */
    VB_T *        pVarBind,  /* var bind being processed                   */
    int           length,    /* The length of the instance par             */
    OIDC_T *      pOid       /* The instance part                          */
    );
```

DESCRIPTION This routine is called from **nextproc** to install into the variable-binding list the instance part (*pOid*) of the next instance that it is going to place into the variable binding.

RETURNS N / A

SEE ALSO **snmpProcLib**

nextproc_no_next( )

NAME *nextproc_no_next*( ) – indicate that there exists no next instance

SYNOPSIS

```
void nextproc_no_next
    (
    SNMP_PKT_T *  pPkt,     /* internal representation of the snmp packet */
    VB_T *        pVarBind  /* var bind being processed                   */
    )
```

DESCRIPTION This routine is called from within **nextproc** if there is no next instance for the requested variable binding.

RETURNS N / A

SEE ALSO **snmpProcLib**

nextproc_started( )

NAME *nextproc_started*( ) – indicate that a **nextproc** operation has begun

SYNOPSIS

```
void nextproc_started
    (
    SNMP_PKT_T *  pPkt,     /* internal representation of the snmp packet */
    VB_T *        pVarBind  /* var bind being processed                   */
    )
```

DESCRIPTION This routine indicates that **nextproc** has begun for the specified variable binding.

RETURNS N / A

SEE ALSO **snmpProcLib**

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
```

SEE ALSO **nfsLib**, *nfsAuthUnixShow*( ), *nfsAuthUnixSet*( ), *nfsAuthUnixGet*( ), *nfsIdSet*( )

nfsAuthUnixSet( )

NAME *nfsAuthUnixSet*( ) – set the NFS UNIX authentication parameters

SYNOPSIS
```
void nfsAuthUnixSet
    (
    char  *machname,  /* host machine        */
    int   uid,        /* user ID             */
    int   gid,        /* group ID            */
    int   ngids,      /* number of group IDs */
    int   *aup_gids   /* array of group IDs  */
    )
```

DESCRIPTION This routine sets UNIX authentication parameters. It is initially called by *usrNetInit*( ) in **usrConfig.c**.

RETURNS N / A

SEE ALSO **nfsLib**, *nfsAuthUnixPrompt*( ), *nfsAuthUnixShow*( ), *nfsAuthUnixGet*( ), *nfsIdSet*( ), **usrConfig**

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*.

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**

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.

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. It is called automatically when **INCLUDE_NFS** is defined in **configAll.h**.

RETURNS OK, or ERROR if there is no room for the driver.

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
/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  *host,        /* name of remote host                       */
    char  *clientName,  /* name of client specified in access list   */
    BOOL  quiet         /* FALSE = print names of file systems mounted */
    )
```

DESCRIPTION This routine mounts the file systems exported by *host* which are marked as accessible either by all clients or only by *clientName*. The *nfsMount*( ) routine is called to mount each file system. This creates a local device for each mounted file system that has the same name as the file system.

The file systems are listed to standard output as they are mounted.

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.

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*( )

nicattach( )

NAME *nicattach*( ) – publish the **nic** network interface and initialize the driver and device

SYNOPSIS

```
STATUS nicattach
    (
    int         unit,   /* unit number             */
    NIC_DEVICE  *pNic,  /* address of NIC chip     */
    int         ivec    /* interrupt vector to use */
    )
```

DESCRIPTION This routine publishes the **nic** interface by filling in a network interface record and adding this record to the system list. It also initializes the driver and the device to the operational state.

RETURNS OK or ERROR.

SEE ALSO **if_nic**

npc( )

NAME *npc*( ) – return the contents of the next program counter (SPARC)

SYNOPSIS

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

DESCRIPTION This command extracts the contents of the next program counter from the TCB of a specified task. If *taskId* is omitted or 0, the current default task is assumed.

RETURNS The contents of the next program counter.

SEE ALSO **dbgArchLib**, *ti*( )

ns16550DevInit( )

NAME *ns16550DevInit*( ) – intialize an NS16550 channel

SYNOPSIS
```
void ns16550DevInit
    (
    NS16550_CHAN *  pChan
    )
```

DESCRIPTION This routine initializes some **SIO_CHAN** function pointers and then resets the chip in a quiescent state. Before this routine is called, the BSP must already have initialized all the device addresses, etc. in the **NS16550_CHAN** structure.

RETURNS N / A

SEE ALSO **ns16550Sio**

ns16550Int( )

NAME *ns16550Int*( ) – interrupt level processing

SYNOPSIS
```
void ns16550Int
    (
    NS16550_CHAN  *pChan
    )
```

DESCRIPTION This routine handles interrupts from the UART.

RETURNS N / A

SEE ALSO **ns16550Sio**

ns16550IntEx( )

NAME *ns16550IntEx*( ) – miscellaneous interrupt processing

SYNOPSIS
```
void ns16550IntEx
    (
    NS16550_CHAN  *pChan
    )
```

DESCRIPTION This routine handles miscellaneous interrupts on the UART.

RETURNS N / A

SEE ALSO **ns16550Sio**

ns16550IntRd( )

NAME *ns16550IntRd*( ) – handle a receiver interrupt

SYNOPSIS
```
void ns16550IntRd
    (
    NS16550_CHAN  *pChan
    )
```

DESCRIPTION This routine handles read interrupts from the UART.

RETURNS N / A

SEE ALSO **ns16550Sio**

ns16550IntWr( )

NAME *ns16550IntWr*( ) – handle a transmitter interrupt

SYNOPSIS
```
void ns16550IntWr
    (
    NS16550_CHAN  *pChan
    )
```

DESCRIPTION This routine handles write interrupts from the UART.

RETURNS N / A

SEE ALSO **ns16550Sio**

o0( )

NAME *o0*( ) – return the contents of register **o0** (also **o1** – **o7**) (SPARC)

SYNOPSIS
```
int o0
    (
    int  taskId  /* task ID, 0 means default task */
    )
```

DESCRIPTION This command extracts the contents of out register **o0** from the TCB of a specified task. If *taskId* is omitted or 0, the current default task is assumed.

Similar routines are provided for all out registers (**o0** – **o7**): *o0*( ) – *o7*( ).

The stack pointer is accessed via **o6**.

RETURNS The contents of register **o0** (or the requested register).

SEE ALSO **dbgArchLib**, *VxWorks Programmer's Guide: Target Shell*

oidcmp( )

NAME *oidcmp*( ) – compare two object identifiers

SYNOPSIS

```
int oidcmp
    (
    int       length_1,  /* length of  first object identifier  */
    OIDC_T *  oid_1,     /* pointer to first object identifier  */
    int       length_2,  /* length of  second object identifier */
    OIDC_T *  oid_2      /* pointer to second object identifier */
    );
```

DESCRIPTION This routine compares two object-identifiers. It returns 1 if the OIDs are the same and 0 if not. The two object idenifiers may be of different lengths.

RETURNS 1 if the OIDs are equal, otherwise 0.

SEE ALSO **snmpAuxLib**

oidcmp2( )

NAME *oidcmp2*( ) – compare two object identifiers

SYNOPSIS

```
int oidcmp2
    (
    int       length_1,  /* length of  first object identifier  */
    OIDC_T *  oid_1,     /* pointer to first object identifier  */
    int       length_2,  /* length of  second object identifier */
    OIDC_T *  oid_2      /* pointer to second object identifier */
    );
```

DESCRIPTION This routine compares two object identifiers.

RETURNS -1 if *oid_1* is less than *oid_2*, 0 if *oid_1* is equal to *oid_2*, and 1 if *oid_1* is greater than *oid_2*.

SEE ALSO **snmpAuxLib**

oid_to_ip( )

NAME

oid_to_ip( ) – convert an object identifier to an IP address

SYNOPSIS

```
int oid_to_ip
    (
    int          count,      /* length of ip address components  */
    OIDC_T *     object_id,  /* pointer to ip address components */
    UINT_32_T *  addr        /* pointer to ip address            */
    );
```

DESCRIPTION

This routine converts an IP address encoded in the instance section of an object identifier to an IP address. The parameter *count* contains the number of octets that corresponds to the IP address in the object identifier. This is usually 4, but may be less, in which case this routine fills the rest of the IP-address portion with zeroes.

The routine puts the IP address in *addr* and returns 0 if successful. If the instance contains a component that is larger than is legal for an IP address (greater than 255), the routine returns 1.

RETURNS

0 on success, otherwise 1.

SEE ALSO

snmpAuxLib

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 in which 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.

SEE ALSO **ioLib**, *creat***( )**

---

*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
    )
```

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, the 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.

RETURNS Pointer to new object, or 0 if allocation failed and a new-handler is not defined.

SEE ALSO **cplusLib**

*operator~new***( )**

NAME *operator~new***( )** – run-time support for operator new with placement (C++)

SYNOPSIS

```
extern void * operator new
    (
    size_t,         // size of object to allocate (unused)
    void     *pMem  // pointer to allocated memory
    )
```

DESCRIPTION This function provides the default implementation of the global new operator, with support for the placement syntax. New-with-placement is used to initialize objects for which memory has already been allocated. *pMem* points to the previously allocated memory. memory.

RETURNS *pMem*

SEE ALSO **cplusLib**

*passFsDevInit***( )**

NAME *passFsDevInit***( )** – associate a device with passFs file system functions (VxSim)

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 (VxSim)

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*( ). To enable this initialization, define **INCLUDE_PASSFS** in **configAll.h**.

NOTE: The maximum number of pass-through file systems is 1.

RETURNS OK, or ERROR.

SEE ALSO **passFsLib**

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*

pccardAtaEnabler( )

NAME *pccardAtaEnabler*( ) – enable the PCMCIA-ATA device

SYNOPSIS

```
STATUS pccardAtaEnabler
    (
    int           sock,           /* socket no.                      */
    ATA_RESOURCE  *pAtaResource,  /* pointer to ATA resources        */
    int           numEnt,         /* number of ATA resource entries */
    FUNCPTR       showRtn         /* ATA show routine                */
    )
```

DESCRIPTION This routine enables the PCMCIA-ATA device.

RETURNS OK, **ERROR_FIND** if there is no ATA card, or ERROR if another error occurs.

SEE ALSO **pccardLib**

pccardEltEnabler( )

NAME *pccardEltEnabler*( ) – enable the PCMCIA Etherlink III card

SYNOPSIS

```
STATUS pccardEltEnabler
    (
    int           sock,           /* socket no.                      */
    ELT_RESOURCE  *pEltResource,  /* pointer to ELT resources        */
    int           numEnt,         /* number of ELT resource entries */
    FUNCPTR       showRtn         /* show routine                    */
    )
```

DESCRIPTION This routine enables the PCMCIA Etherlink III (ELT) card.

RETURNS OK, **ERROR_FIND** if there is no ELT card, or ERROR if another error occurs.

SEE ALSO **pccardLib**

pccardMkfs( )

NAME *pccardMkfs*( ) – initialize a device and mount a DOS file system

SYNOPSIS

```
STATUS pccardMkfs
    (
    int   sock,   /* socket number    */
    char  *pName  /* name of a device */
    )
```

DESCRIPTION This routine initializes a device and mounts a DOS file system.

RETURNS OK or ERROR.

SEE ALSO **pccardLib**

pccardMount( )

NAME *pccardMount*( ) – mount a DOS file system

SYNOPSIS

```
STATUS pccardMount
    (
    int   sock,   /* socket number    */
    char  *pName  /* name of a device */
    )
```

DESCRIPTION This routine mounts a DOS file system.

RETURNS OK or ERROR.

SEE ALSO **pccardLib**

pccardSramEnabler( )

NAME *pccardSramEnabler*( ) – enable the PCMCIA-SRAM driver

SYNOPSIS

```
STATUS pccardSramEnabler
    (
    int            sock,            /* socket no.                        */
    SRAM_RESOURCE  *pSramResource,  /* pointer to SRAM resources         */
    int            numEnt,          /* number of SRAM resource entries */
    FUNCPTR        showRtn          /* SRAM show routine                 */
    )
```

DESCRIPTION This routine enables the PCMCIA-SRAM driver.

RETURNS OK, **ERROR_FIND** if there is no SRAM card, or ERROR if another error occurs.

SEE ALSO **pccardLib**

pcicInit( )

NAME *pcicInit*( ) – initialize the PCIC chip

SYNOPSIS

```
STATUS pcicInit
    (
    int      ioBase,    /* IO base address  */
    int      intVec,    /* interrupt vector */
    int      intLevel,  /* interrupt level  */
    FUNCPTR  showRtn    /* show routine     */
    )
```

DESCRIPTION This routine initializes the PCIC chip.

RETURNS OK, or ERROR if the PCIC chip cannot be found.

SEE ALSO **pcic**

pcicShow( )

NAME *pcicShow*( ) – show all configurations of the PCIC chip

SYNOPSIS

```
void pcicShow
    (
    int  sock  /* socket no. */
    )
```

DESCRIPTION This routine shows all configurations of the PCIC chip.

RETURNS N / A

SEE ALSO **pcicShow**

pcmciad( )

NAME *pcmciad*( ) – handle task-level PCMCIA events

SYNOPSIS
```
void pcmciad (void)
```

DESCRIPTION This routine is spawned as a task by *pcmciaInit*( ) 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 **pcmciaLib**, *pcmciaInit*( )

pcmciaInit( )

NAME *pcmciaInit*( ) – initialize the PCMCIA event-handling package

SYNOPSIS
```
STATUS pcmciaInit (void)
```

DESCRIPTION This routine installs the PCMCIA event-handling facilities and spawns *pcmciad*( ), which performs special PCMCIA event-handling functions that need to be done at task level. It also creates the message queue used to communicate with *pcmciad*( ).

RETURNS OK, or ERROR if a message queue cannot be created or *pcmciad*( ) cannot be spawned.

SEE ALSO **pcmciaLib**, *pcmciad*( )

pcmciaShow( )

NAME *pcmciaShow*( ) – show all configurations of the PCMCIA chip

SYNOPSIS
```
void pcmciaShow
    (
    int  sock  /* socket no. */
    )
```

DESCRIPTION This routine shows all configurations of the PCMCIA chip.

RETURNS N / A

SEE ALSO **pcmciaShow**

pcmciaShowInit( )

NAME *pcmciaShowInit*( ) – initialize all show routines for PCMCIA drivers

SYNOPSIS
```
void pcmciaShowInit (void)
```

DESCRIPTION This routine initializes all show routines related to PCMCIA drivers.

RETURNS N / A

SEE ALSO **pcmciaShow**

pcw( )

NAME *pcw*( ) – return the contents of the **pcw** register (i960)

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

DESCRIPTION This command extracts the contents of the **pcw** register from the TCB of a specified task. If *taskId* is omitted or 0, the current default task is assumed.

RETURNS The contents of the **pcw** register.

SEE ALSO **dbgArchLib**, *VxWorks Programmer's Guide: Target Shell*

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*

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***( )**

pfp( )

NAME *pfp*( ) – return the contents of register **pfp** (i960)

SYNOPSIS

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

DESCRIPTION This command extracts the contents of register **pfp**, the previous frame pointer, from the TCB of a specified task. If *taskId* is omitted or 0, the current default task is assumed.

RETURNS The contents of the **pfp** register.

SEE ALSO **dbgArchLib**, *VxWorks Programmer's Guide: Target Shell*

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.

RETURNS OK, or ERROR if the remote host is not reachable.

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 must be called before *ping***( )** is used. It is called automatically when **INCLUDE_PING** is defined in **configAll.h**.

RETURNS OK, or ERROR if the *ping***( )** utility could not be initialized.

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 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.

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 **INCLUDE_PIPES** is defined in **configAll.h**.

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**

ppc403DevInit( )

NAME *ppc403DevInit*( ) – initialize the serial port unit

SYNOPSIS

```
void ppc403DevInit
    (
    PPC403_CHAN *  pChan
    )
```

DESCRIPTION The BSP must already have initialized all the device addresses in the **PPC403_CHAN** structure. This routine initializes some **SIO_CHAN** function pointers and then resets the chip in a quiescent state.

SEE ALSO **ppc403Sio**

ppc403DummyCallback( )

NAME *ppc403DummyCallback*( ) – dummy callback routine

SYNOPSIS

```
STATUS ppc403DummyCallback (void)
```

RETURNS ERROR (always).

SEE ALSO **ppc403Sio**

ppc403IntEx( )

NAME *ppc403IntEx*( ) – handle error interrupts

SYNOPSIS
```
void ppc403IntEx
    (
    PPC403_CHAN *  pChan
    )
```

DESCRIPTION This routine handles miscellaneous interrupts on the seial communication controller.

RETURNS N / A

SEE ALSO **ppc403Sio**

ppc403IntRd( )

NAME *ppc403IntRd*( ) – handle a receiver interrupt

SYNOPSIS
```
void ppc403IntRd
    (
    PPC403_CHAN *  pChan
    )
```

DESCRIPTION This routine handles read interrupts from the serial commonication controller.

RETURNS N / A

SEE ALSO **ppc403Sio**

ppc403IntWr( )

NAME *ppc403IntWr*( ) – handle a transmitter interrupt

SYNOPSIS
```
void ppc403IntWr
    (
    PPC403_CHAN *  pChan
    )
```

DESCRIPTION This routine handles write interrupts from the serial communication controller.

RETURNS N / A

SEE ALSO **ppc403Sio**

ppc860DevInit( )

NAME *ppc860DevInit*( ) – initialize the SMC

SYNOPSIS
```
void ppc860DevInit
    (
    PPC860SMC_CHAN  *pChan
    )
```

DESCRIPTION This routine is called to initialize the chip to a quiescent state. Note that the **smcNum** field of **PPC860SMC_CHAN** must be either 1 or 2.

SEE ALSO **ppc860Sio**

ppc860Int( )

NAME *ppc860Int*( ) – handle an SMC interrupt

SYNOPSIS
```
void ppc860Int
    (
    PPC860SMC_CHAN  *pChan
    )
```

DESCRIPTION This routine is called to handle SMC interrupts.

SEE ALSO **ppc860Sio**

---

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***( )**

---

*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.

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 rechallenge 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) connnection 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 to the specified value.

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.

pap_passwd *password*
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 rechallenge 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*

printf( )

NAME

printf( ) – write a formatted string to the standard output stream (ANSI)

SYNOPSIS

```
int printf
    (
    const char *  fmt,  /* format string to write             */
    ...                 /* optional arguments to format string */
    )
```

DESCRIPTION

This routine writes output to standard output under control of the string *fmt*. The string *fmt* contains ordinary characters, which are written unchanged, plus conversion specifications, which cause the arguments that follow *fmt* to be converted and printed as part of the formatted string.

The number of arguments for the format is arbitrary, but they must correspond to the conversion specifications in *fmt*. If there are insufficient arguments, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored. The routine returns when the end of the format string is encountered.

The format is a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: ordinary multibyte characters

(not **%**) that are copied unchanged to the output stream; and conversion specification, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the **%** character. After the **%**, the following appear in sequence:

– Zero or more flags (in any order) that modify the meaning of the conversion specification.

– An optional minimum field width. If the converted value has fewer characters than the field width, it will be padded with spaces (by default) on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of an asterisk (*****) (described later) or a decimal integer.

– An optional precision that gives the minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, and **X** conversions, the number of digits to appear after the decimal-point character for **e**, **E**, and **f** conversions, the maximum number of significant digits for the **g** and **G** conversions, or the maximum number of characters to be written from a string in the **s** conversion. The precision takes the form of a period (**.**) followed either by an asterisk (*****) (described later) or by an optional decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.

– An optional **h** specifying that a following **d**, **i**, **o**, **u**, **x**, and **X** conversion specifier applies to a **short int** or **unsigned short int** argument (the argument will have been promoted according to the integral promotions, and its value converted to **short int** or **unsigned short int** before printing); an optional **h** specifying that a following **n** conversion specifier applies to a pointer to a **short int** argument; an optional **l** (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 to be converted can be represented in fewer digits, it is 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 unspecified or 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 initializes the proxy ARP library by initializing tables and structures and adding the hooks to process ARP, proxy messages, and broadcasts. *clientSizeLog2* specifies the client hash table size as a power of two. *portSizeLog2* specifies the port hash table as a power of two. If either of these parameters is zero, a default value will be used. By default, *proxyArpLibInit*( ) enables broadcast forwarding of the BOOTP server port.

This routine should be called only once; subsequent calls have no effect.

RETURNS OK, or ERROR if unsuccessful.

SEE ALSO **proxyArpLib**

proxyNetCreate( )

NAME *proxyNetCreate*( ) – create a proxy ARP network

SYNOPSIS

```
STATUS proxyNetCreate
    (
    char *  proxyAddr,  /* proxy network address   */
    char *  mainAddr    /* main network address    */
    )
```

DESCRIPTION This routine creates a proxy network with the interface *proxyAddr* as the proxy network and the interface *mainAddr* as the main network. The interfaces and the routing tables must be set up correctly, prior to calling this routine. That is, the interfaces must be attached, addresses must be set, and there should be a network route to *mainAddr* and no routes to *proxyAddr*.

proxyAddr and *mainAddr* must reside in the same network address space.

RETURNS OK, or ERROR if unsuccessful.

ERRNO
S_proxyArpLib_INVALID_INTERFACE
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**

proxyPortFwdOn( )

NAME *proxyPortFwdOn*( ) – enable broadcast forwarding for a particular port

SYNOPSIS
```
STATUS proxyPortFwdOn
    (
    int  port  /* port number          */
    )
```

DESCRIPTION This routine enables broadcasts destined for 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 enabled ports

SYNOPSIS
```
void proxyPortShow (void)
```

DESCRIPTION This routine displays the ports currently enabled.

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**

psr( )

NAME

psr( ) – return the contents of the processor status register (SPARC)

SYNOPSIS

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

DESCRIPTION

This command extracts the contents of the processor status register from the TCB of a specified task. If *taskId* is omitted or 0, the default task is assumed.

RETURNS

The contents of the processor status register.

SEE ALSO

dbgArchLib, *psrShow*( ), *VxWorks Programmer's Guide: Target Shell*

psrShow( )

NAME

psrShow( ) – display the meaning of a specified **psr** value, symbolically (SPARC)

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***( )**, *SPARC Architecture Manual*

*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**

*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**

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***( )**

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.

RETURNS N / A

SEE ALSO **usrLib**, *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

quattach( )

NAME

quattach( ) – publish the **qu** network interface and initialize driver structures

SYNOPSIS

```
STATUS quattach
    (
    int     unit,      /* unit number                                */
    UINT32  quAddr,    /* base address of QUICC chip internal mem  */
    int     ivec,      /* interrupt vector                           */
    int     sccNum,    /* SCC number used                            */
    int     txBdNum,   /* number of transmit buffer descriptors    */
    int     rxBdNum,   /* number of receive buffer descriptors     */
```

```
    UINT32  txBdBase,  /* transmit buffer descriptor offset        */
    UINT32  rxBdBase,  /* receive buffer descriptor offset         */
    UINT32  bufBase    /* address of memory pool; NONE = malloc it */
    )
```

DESCRIPTION

The routine publishes the **qu** interface by filling in a network Interface Data Record (IDR) and adding this record to the system list.

The MC68EN360 shares a region of memory with the driver. The caller of this routine can specify the address of a non-cacheable memory region with *bufBase*, or if this parameter is "NONE" the driver will obtain this memory region from calls to *cacheDmaMalloc***( )**. Non-cacheable memory space is important for cases where the MC68EN360 is operating in companion mode, and is attached to a processor with cache memory.

Once non-cacheable memory is obtained, this routine will divide up the memory between the various buffer descriptors (BDs). The number of BDs can be specified by *txBdNum* and *rxBdNum*, or if "NULL", a default value of 32 BDs will be used. An additional number of buffers are reserved as receive loaner buffers. The number of loaner buffers is the lesser of *rxBdNum* and a default number of 16.

The user must specify the location of the transmit and receive BDs in the 68EN360's dual ported RAM. *txBdBase* and *rxBdBase* give the offsets from *quAddr* for the base of the BD rings. Each BD uses 8 bytes. Care must be taken so that the specified locations for Ethernet BDs do not conflict with other dual ported RAM structures.

Up to four individual device units are supported by this driver. Device units may reside on different MC68EN360 chips, or may be on different SCCs within a single MC68EN360. The *sccNum* parameter is used to explicitly state which SCC is being used. SCC1 is most commonly used, thus this parameter most often equals "1".

Before this routine returns, it calls *quReset***( )** to put the Ethernet controller in a quiescent state, and connects up the interrupt vector *ivec*.

RETURNS

OK or ERROR.

SEE ALSO

if_qu, **ifLib**, *Motorola MC68360 User's Manual*

---

*r3***( )**

NAME

*r3***( )** – return the contents of register **r3** (also **r4** – **r15**) (i960)

SYNOPSIS

```
int r3
    (
    int  taskId  /* task ID, 0 means default task */
    )
```

DESCRIPTION This command extracts the contents of register **r3** from the TCB of a specified task. If *taskId* is omitted or 0, the current default task is assumed.

Routines are provided for all local registers (**r3** – **r15**): *r3***( )** – *r15***( )**.

RETURNS The contents of the **r3** register (or the requested register).

SEE ALSO **dbgArchLib**, *VxWorks Programmer's Guide: Target Shell*

---

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***( )**

*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     *volName,  /* volume name                  */
    BLK_DEV  *pBlkDev   /* pointer to block device info */
    )
```

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 *volName* 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 **BLK_DEV** structure specified by *pBlkDev* contains configuration data describing the device and the addresses of five routines which will be called to read blocks, write blocks, reset the device, check device status, and perform other control functions (*ioctl***( )**). 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

rawFsModeChange( )

NAME

rawFsModeChange( ) – modify the mode of a raw device volume

SYNOPSIS

```
void rawFsModeChange
    (
    RAW_VOL_DESC  *vdptr,  /* 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 **BLK_DEV** 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  *vdptr  /* 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  *vdptr  /* 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 executes a command on a remote machine, using the remote shell daemon, **rshd**, on the remote system. It is analogous to the UNIX routine *rcmd***( )**.

RETURNS A socket descriptor if the remote shell daemon accepts, or ERROR if the remote command fails.

SEE ALSO **remLib**, UNIX BSD 4.3 manual entry for *rcmd***( )**

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**)

*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:

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.

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, currently unused            */
    )
```

DESCRIPTION This routine unloads a specified object module from the system, and then calls *ld***( )** 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.

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*( ), intended for use 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**)

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: This routine does not work for files mounted remotely using the VxWorks NFS client.

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*

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 the stream *fp* 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**

rip( )

NAME *rip*( ) – return the contents of register **rip** (i960)

SYNOPSIS

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

DESCRIPTION This command extracts the contents of register **rip**, the return instruction pointer, from the TCB of a specified task. If *taskId* is omitted or 0, the current default task is assumed.

RETURNS The contents of the **rip** register.

SEE ALSO **dbgArchLib**, *VxWorks Programmer's Guide: Target Shell*

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*( )

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 in **configAll.h**, it 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.

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 defined in **configAll.h**, it is called automatically at boot time.

RETURNS OK or ERROR.

SEE ALSO **rlogLib**, **ptyDrv**

---

*rm***( )**

NAME *rm***( )** – remove a file

SYNOPSIS
```
STATUS rm
    (
    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 **usrLib**, *remove***( )**, *VxWorks Programmer's Guide: Target Shell*

rmdir( )

NAME *rmdir*( ) – remove a directory

SYNOPSIS
```
STATUS rmdir
    (
    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 **usrLib**, *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 would **RNG_ELEM_PUT** would put a byte, except that the input pointer is not updated.

Bytes written ahead in the ring buffer with this routine can be made available all at once by subsequently moving the ring buffer pointers with the routine *rngMoveAhead*( ).

Before calling *rngPutAhead*( ), the caller must verify that at least *offset* + 1 bytes are available in the ring buffer.

RETURNS N / A

SEE ALSO **rngLib**

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 uncompressor. 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).

Both *destination* and *gateway* can be specified in standard Internet address format (e.g., 90.0.0.2), or they can be specified by their host names, as specified with *hostAdd*( ).

EXAMPLES The following 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*( ):

```
-> routeAdd "90.0.0.0", "gate"
```

The following call tells VxWorks that the machine with the Internet address 91.0.0.3 is the gateway to network 90.0.0.0:

```
-> routeAdd "90.0.0.0", "91.0.0.3"
```

The following 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***( )**:

```
-> routeAdd "destination", "gate"
```

The following 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***( )**:

```
-> routeAdd "0", "gate"
```

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***( )**

---

*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.

RETURNS OK or ERROR.

SEE ALSO **routeLib**

---

*routeShow***( )**

NAME *routeShow***( )** – display host and network routing tables

SYNOPSIS

```
void routeShow (void)
```

DESCRIPTION This routine displays the current routing information contained in the routing table.

EXAMPLE

```
-> routeShow
ROUTE NET TABLE
destination      gateway         flags  Refcnt  Use      Interface
------------------------------------------------------------------
90.0.0.0         90.0.0.63       1      1       142      enp0
------------------------------------------------------------------
ROUTE HOST TABLE
destination      gateway         flags  Refcnt  Use      Interface
------------------------------------------------------------------
127.0.0.1        127.0.0.1       5      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 (i.e., “up”)
0x2 – destination is a gateway
0x4 – host specific routing entry
0x10 – created dynamically (by redirect)
0x20 – modified dynamically (by redirect)

In the above display example, the entry under 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 under 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.

RETURNS N / A

SEE ALSO **netShow**

routestatShow( )

NAME *routestatShow*( ) – display routing statistics

SYNOPSIS
```
void routestatShow (void)
```

DESCRIPTION This routine displays routing statistics.

RETURNS N / A

SEE ALSO **netShow**

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 in **configAll.h**.

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.

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**

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** if **INCLUDE_RT11FS** is defined in **configAll.h**.

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.

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 */
    INSTR *  addr1          /* address for npc, 0 = next instruction    */
    )
```

DESCRIPTION This routine single-steps a task that is stopped at a breakpoint.

To execute, enter:

-> s [*taskNameOrId* [**,***addr*[**,***addr1*]]]

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

CAVEAT

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

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

RETURNS

OK, or ERROR if the debugging package is not installed, the task cannot be found, or the task is not suspended.

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_getparam( )

NAME *sched_getparam*( ) – get the scheduling parameters for a specified task (POSIX)

SYNOPSIS

```
int sched_getparam
    (
    pid_t                tid,   /* task ID                              */
    struct sched_param *  param  /* scheduling param to store priority */
    )
```

DESCRIPTION This routine gets the scheduling priority for a specified task, *tid*. If *tid* is 0, it gets the priority of the calling task. The task's priority is copied to the **sched_param** structure pointed to by *param*.

NOTE: If the global variable **posixPriorityNumbering** is FALSE, the VxWorks native priority numbering scheme is used, in which higher priorities are indicated by smaller numbers. This is different than the priority numbering scheme specified by POSIX, in which higher priorities are indicated by larger numbers.

RETURNS 0 (OK) if successful, or -1 (ERROR) on error.

ERRNO

ESRCH
invalid task ID.

SEE ALSO **schedPxLib**

sched_getscheduler( )

NAME *sched_getscheduler*( ) – get the current scheduling policy (POSIX)

SYNOPSIS

```
int sched_getscheduler
    (
    pid_t  tid  /* task ID */
    )
```

DESCRIPTION This routine returns the currents scheduling policy (i.e., **SCHED_FIFO** or **SCHED_RR**).

RETURNS Current scheduling policy (**SCHED_FIFO** or **SCHED_RR**), or -1 (ERROR) on error.

ERRNO **ESRCH**
invalid task ID.

SEE ALSO **schedPxLib**

sched_get_priority_max( )

NAME *sched_get_priority_max*( ) – get the maximum priority (POSIX)

SYNOPSIS
```
int sched_get_priority_max
    (
    int  policy  /* scheduling policy */
    )
```

DESCRIPTION This routine returns the value of the highest possible task priority for a specified scheduling policy (**SCHED_FIFO** or **SCHED_RR**).

NOTE: If the global variable **posixPriorityNumbering** is FALSE, the VxWorks native priority numbering scheme is used, in which higher priorities are indicated by smaller numbers. This is different than the priority numbering scheme specified by POSIX, in which higher priorities are indicated by larger numbers.

RETURNS Maximum priority value, or -1 (ERROR) on error.

ERRNO **EINVAL**
invalid scheduling policy.

SEE ALSO **schedPxLib**

sched_get_priority_min( )

NAME *sched_get_priority_min*( ) – get the minimum priority (POSIX)

SYNOPSIS
```
int sched_get_priority_min
    (
    int  policy  /* scheduling policy */
    )
```

DESCRIPTION This routine returns the value of the lowest possible task priority for a specified scheduling policy (**SCHED_FIFO** or **SCHED_RR**).

NOTE: If the global variable **posixPriorityNumbering** is FALSE, the VxWorks native priority numbering scheme is used, in which higher priorities are indicated by smaller numbers. This is different than the priority numbering scheme specified by POSIX, in which higher priorities are indicated by larger numbers.

RETURNS Minimum priority value, or -1 (ERROR) on error.

ERRNO

EINVAL
invalid scheduling policy.

SEE ALSO **schedPxLib**

*sched_rr_get_interval***( )**

NAME *sched_rr_get_interval***( )** – get the current time slice (POSIX)

SYNOPSIS

```
int sched_rr_get_interval
    (
    pid_t              tid,      /* task ID                    */
    struct timespec *  interval  /* struct to store time slice */
    )
```

DESCRIPTION This routine sets *interval* to the current time slice period if round-robin scheduling is currently enabled.

RETURNS 0 (OK) if successful, -1 (ERROR) on error.

ERRNO

EINVAL
round-robin scheduling is not currently enabled.

ESRCH
invalid task ID.

SEE ALSO **schedPxLib**

sched_setparam( )

NAME

sched_setparam( ) – set a task's priority (POSIX)

SYNOPSIS

```
int sched_setparam
    (
    pid_t                       tid,   /* task ID              */
    const struct sched_param *  param  /* scheduling parameter */
    )
```

DESCRIPTION

This routine sets the priority of a specified task, *tid*. If *tid* is 0, it sets the priority of the calling task. Valid priority numbers are 0 through 255.

The *param* argument is a structure whose member **sched_priority** is the integer priority value. For example, the following program fragment sets the calling task's priority to 13 using POSIX interfaces:

```
#include "sched.h"
 ...
struct sched_param AppSchedPrio;
 ...
AppSchedPrio.sched_priority = 13;
if ( sched_setparam (0, &AppSchedPrio) != OK )
    {
    ... /* recovery attempt or abort message */
    }
 ...
```

NOTE: If the global variable **posixPriorityNumbering** is FALSE, the VxWorks native priority numbering scheme is used, in which higher priorities are indicated by smaller numbers. This is different than the priority numbering scheme specified by POSIX, in which higher priorities are indicated by larger numbers.

RETURNS

0 (OK) if successful, or -1 (ERROR) on error.

ERRNO

EINVAL
scheduling priority is outside valid range.

ESRCH
task ID is invalid.

SEE ALSO

schedPxLib

sched_setscheduler( )

NAME

sched_setscheduler( ) – set scheduling policy and scheduling parameters (POSIX)

SYNOPSIS

```
int sched_setscheduler
    (
    pid_t                       tid,     /* task ID                          */
    int                         policy,  /* scheduling policy requested      */
    const struct sched_param *  param    /* scheduling parameters requested */
    )
```

DESCRIPTION

This routine sets the scheduling policy and scheduling parameters for a specified task, *tid*. If *tid* is 0, it sets the scheduling policy and scheduling parameters for the calling task.

Because VxWorks does not set scheduling policies (e.g., round-robin scheduling) on a task-by-task basis, setting a scheduling policy that conflicts with the current system policy simply fails and errno is set to **EINVAL**. If the requested scheduling policy is the same as the current system policy, then this routine acts just like *sched_setparam*( ).

NOTE: If the global variable **posixPriorityNumbering** is FALSE, the VxWorks native priority numbering scheme is used, in which higher priorities are indicated by smaller numbers. This is different than the priority numbering scheme specified by POSIX, in which higher priorities are indicated by larger numbers.

RETURNS

The previous scheduling policy (**SCHED_FIFO** or **SCHED_RR**), or -1 (ERROR) on error.

ERRNO

EINVAL
scheduling priority is outside valid range, or it is impossible to set the specified scheduling policy.

ESRCH
invalid task ID.

SEE ALSO

schedPxLib

sched_yield( )

NAME

sched_yield( ) – relinquish the CPU (POSIX)

SYNOPSIS

```
int sched_yield (void)
```

DESCRIPTION

This routine forces the running task to give up the CPU.

RETURNS 0 (OK) if successful, or -1 (ERROR) on error.

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**

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 neccessary cache coherency code. In order to make **scsi2Lib** aware that hardware snooping is disabled, this routine should be called after all SCSI-2 initializations, especially after *scsi2CtrlInit*( ).

RETURNS N / A

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 unfilled 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

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
    )
```

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, reselection, or disconnection occurs or when a thread is activated. It manages a simple finite-state machine for the SCSI controller.

NOTE: This function should not be called by application programs.

RETURNS N / A

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( )

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 occassionally 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 page-format bit (0-1)     */
    int             saveParams,   /* value of 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 page-control field (0-3) */
    int              pageCode,      /* value of 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**

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          */
    )
```

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, /* lngth of REQUEST SENSE data dev returns */
    int         devType,        /* type of SCSI device                     */
    BOOL        removable,      /* whether medium is removable             */
    int         numBlocks,      /* number of blocks on device              */
    int         blockSize       /* size of a block in bytes                */
    )
```

DESCRIPTION This routine enables access to a SCSI device and must be the first routine invoked. It must be called once for each physical device on the SCSI bus.

If *reqSenseLength* is NULL (0), one or more **REQUEST_SENSE** commands are issued to the device to determine the number of bytes of sense data it typically returns. Note that if the device returns variable amounts of sense data depending on its state, you must consult the device manual to determine the maximum amount of sense data that can be returned.

If *devType* is NONE (-1), an INQUIRY command is issued to determine the device type; as an added benefit, it acquires the device's make and model number. The *scsiShow***( )** routine displays this information. Common values of *devType* can be found in **scsiLib.h** or in the SCSI specification.

If *numBlocks* or *blockSize* are 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_DEV**s 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**

---

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 from a SCSI tape device

SYNOPSIS
```
STATUS scsiRdTape
    (
    SCSI_SEQ_DEV  *pScsiSeqDev,  /* ptr to SCSI sequential device info */
    int           numBytes,      /* 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 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 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 bytes cannot be read or zero bytes are read.

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 put last logical block addr */
    int *           pBlkLength     /* where to put 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**

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.

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 put maximum block length */
    UINT16         *pMinBlockLength   /* where to put minimum block length */
    )
```

DESCRIPTION This routine issues a **READ_BLOCK_LIMITS** command to a specified SCSI device.

RETURNS OK, or ERROR if the command fails.

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.

SEE ALSO

scsiSeqLib

scsiStartStopUnit( )

NAME

scsiStartStopUnit( ) – issue a **START_STOP_UNIT** command to a SCSI device

SYNOPSIS

```
STATUS scsiStartStopUnit
    (
    SCSI_PHYS_DEV  *pScsiPhysDev,  /* ptr to SCSI physical device  */
    BOOL           start           /* TRUE == start, FALSE == stop */
    )
```

DESCRIPTION This routine issues a **START_STOP_UNIT** command to a specified SCSI device.

RETURNS OK, or ERROR if the command fails.

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.

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.

NOTE: 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 page-format bit (0-1)     */
    int            saveParams,     /* value of 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 page-control field (0-3) */
    int            pageCode,       /* value of 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**

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 bitmask *which* for the specified target (or all targets if *devBusId* is **SCSI_SET_OPT_ALL_TARGETS**).

The bitmask *which* can be any combination of the following, bitwise OR'd together (corresponding fields in the **SCSI_OPTIONS** structure are shown in the second column):

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 in bits

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**

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.

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.

NOTE: 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**

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, or simply **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.

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***( )**.

SEE ALSO **selectLib**, *VxWorks Programmer's Guide: I/O System*

*selectInit***( )**

NAME *selectInit***( )** – initialize the select facility

SYNOPSIS
```
void selectInit (void)
```

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 delete hook that cleans up after a task if the task is deleted while pended in *select***( )**.

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          /* read (SELREAD) or write (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**

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.

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.

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**, defined in **configAll.h**.

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_smMemLib_NOT_ENOUGH_MEMORY
S_semLib_INVALID_QUEUE_TYPE
S_semLib_INVALID_STATE
S_smObjLib_LOCK_TIMEOUT

SEE ALSO **semSmLib**, **semLib**, **semBLib**, **smObjLib**, **semShow**

---

*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.

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 (the number of times the semaphore should be taken before it can be given) 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**, defined in **configAll.h**.

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_smMemLib_NOT_ENOUGH_MEMORY
S_semLib_INVALID_QUEUE_TYPE
S_smObjLib_LOCK_TIMEOUT

SEE ALSO **semSmLib**, **semLib**, **semCLib**, **smObjLib**, **semShow**

---

*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. Any pended tasks 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.

ERRNOS Possible errnos generated by this routine include:

S_intLib_NOT_ISR_CALLABLE
This routine cannot be called from an ISR.

S_objLib_OBJ_ID_ERROR
This is an incorrect semaphore ID.

S_smObjLib_NO_OBJECT_DESTROY
This semaphore is shared so you cannot delete it at this time.

SEE ALSO **semLib**, **semBLib**, **semCLib**, **semMLib**, **semSmLib**

---

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*( ).

RETURNS OK, or ERROR if the semaphore ID is invalid or the operation is not supported.

ERRNOS Possible errnos generated by this routine include:

S_objLib_OBJ_ID_ERROR
This is an incorrect semaphore ID.

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. The behavior of *semGive*( ) is discussed fully in the library description of the specific semaphore type being used.

RETURNS

OK, or ERROR if the semaphore ID is invalid.

ERRNOS

Possible errnos generated by this routine include:

S_intLib_NOT_ISR_CALLABLE
This routine cannot be called from an ISR.

S_objLib_OBJ_ID_ERROR
This is an incorrect semaphore ID.

S_semLib_INVALID_OPERATION
You don't own the mutex semaphore so you can't give it.

SEE ALSO

semLib, **semBLib**, **semCLib**, **semMLib**, **semSmLib**

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***( )**

*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.

RETURNS The semaphore ID, or NULL if memory cannot be allocated.

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.

CAVEATS This routine should be used only as a debugging aid, when the condition of the semaphore is known.

RETURNS OK, or ERROR if the semaphore ID is invalid.

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**

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. These routines are included automatically by defining **INCLUDE_SHOW_RTNS** in **configAll.h**.

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}
```

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:

```
   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 **INCLUDE_SHOW_ROUTINES** is defined in **configAll.h**.

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.

ERRNOS Possible errnos generated by this routine include:

S_intLib_NOT_ISR_CALLABLE
This routine cannot be called from an ISR.

S_objLib_OBJ_ID_ERROR
This is an incorrect semaphore ID.

S_objLib_OBJ_UNAVAILABLE
You have specified NOWAIT and the resource is currently 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: 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 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.

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: 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*( )

send( )

NAME

send( ) – send data to a socket

SYNOPSIS

```
int send
    (
    int   s,       /* socket to send to             */
    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***( )**

---

*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.

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* will be 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 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*( )

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 0 or -1.

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 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;

– as the operand of a single-argument **!** operator, in the controlling expression of a selection or iteration statement; or

– as a complete C statement 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 selects the appropriate portion of the program's locale as specified by the *category* and *localeName* arguments. This routine can be used to change or query the program's entire current locale or portions thereof.

Values for *category* affect the locale as follows:

LC_ALL
specifies the program's entire locale.

LC_COLLATE
affects the behavior of the *strcoll***( )** and *strxfrm***( )** functions.

LC_CTYPE
affects the behavior of the character-handling functions and the multi-byte functions.

LC_MONETARY
affects the monetary-formatting information returned by *localeconv***( )**.

LC_NUMERIC
affects the decimal-point character for the formatted input/output functions and the string-conversion functions, as well as the nonmonetary-formatting information returned by *localeconv***( )**.

LC_TIME
affects the behavior of the *strftime***( )** function.

A value of "C" for *localeName* specifies the minimal environment for C translation; a value of "" specifies the implementation-defined native environment. Other implementation-defined strings may be passed as the second argument.

At program start-up, the equivalent of the following is executed:

```
setlocale (LC_ALL, "C");
```

The implementation behaves as if no library function calls *setlocale***( )**.

If *localeName* is a pointer to a string and the selection can be honored, *setlocale***( )** returns a pointer to the string associated with the specified category for the new locale. If the selection cannot be honored, it returns a null pointer and the program's locale is unchanged.

If *localeName* is null pointer, *setlocale***( )** returns a pointer to the string associated with the category for the program's current locale; the program's locale is unchanged.

The string pointer returned by *setlocale***( )** is such that a subsequent call with that string value and its associated category will restore that part of the program's locale. The string is not modified by the program, but may be overwritten by a subsequent call to *setlocale***( )**.

Currently, only the "C" locale is implemented in this library.

INCLUDE FILES

locale.h, **string.h**, **stdlib.h**

RETURNS A pointer to the string “C”.

SEE ALSO **ansiLocale**

setproc_error( )

NAME *setproc_error*( ) – indicate that a **setproc** operation encountered an error

SYNOPSIS
```
void setproc_error
    (
    SNMP_PKT_T *  pPkt,      /* internal representation of the snmp packet */
    VB_T *        pVarBind,  /* var bind being processed                   */
    INT_32_T      error      /* error code                                 */
    )
```

DESCRIPTION This routine indicates that **setproc** encountered an error while perorming a **set** operation for a variable binding.

RETURNS N / A

SEE ALSO **snmpProcLib**

setproc_good( )

NAME *setproc_good*( ) – indicates successful completion of a **setproc** procedure

SYNOPSIS
```
void setproc_good
    (
    SNMP_PKT_T *  pPkt,     /* internal representation of the snmp packet */
    VB_T *        pVarBind  /* var bind being processed                   */
    )
```

DESCRIPTION This routine indicate that **setproc** has successfully completed the **set** operation for a variable binding.

RETURNS N / A

SEE ALSO **snmpProcLib**

*setproc_started***( )**

NAME *setproc_started***( )** – indicate that a **setproc** operation has begun

SYNOPSIS

```
void setproc_started
    (
    SNMP_PKT_T *  pPkt,
    VB_T *        pVarBind
    )
```

DESCRIPTION This routine indicates that **setproc** has been started by the user for the specified variable binding.

RETURNS N / A

SEE ALSO **snmpProcLib**

*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.

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 parameters **TCPTV_KEEP_INIT**, **TCPTV_KEEP_IDLE**, **TCPTV_KEEPCNT**, and **TCPTV_KEEPINTVL** are defined in the file **target/h/net/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));
```

For a "graceful" close, when a connection is shut down TCP tries to make sure that all the unacknowledged data in transmission channel are acknowledged, and the peer is shut down properly, by going through an elaborate set of state transitions.

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; 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.

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 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 even if it is still bound to another stream socket, if that other socket is

associated with a “zombie” protocol control block context 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 which is being used by a zombie protocol control block, the *bind***( )** call fails.

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 4096 bytes. For UDP, the default size of the send buffer is 2048 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, but 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 4096 bytes. UDP reserves enough space for up to four 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 which 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**

---

*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.

RETURNS A pointer to the previous value of new_handler.

SEE ALSO **cplusLib**

---

*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 **INCLUDE_SHELL** is defined in **configAll.h**, this is done 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*

*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*

shutdown( )

NAME *shutdown*( ) – shut down a network connection

SYNOPSIS

```
STATUS shutdown
    (
    int  s,   /* the socket to shutdown            */
    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 will be disallowed. If *how* is 1, sends will be disallowed. If *how* is 2, both sends and receives are disallowed.

RETURNS OK, or ERROR if the socket is invalid or not connected.

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*( )

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
    (
    int               signo,
    void (*pHandler)  ()
    ))                ()
```

DESCRIPTION This routine chooses one of three ways in which receipt of the signal number *signo* is to be subsequently handled. If the value of *pHandler* is **SIG_DFL**, default handling for that signal will occur. If the value of *pHandler* is **SIG_IGN**, the signal will be ignored. Otherwise, *pHandler* must point to a function to be called when that signal occurs.

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 asynchromously generated signals. A task should use *sigprocmask***( )** to block any signals it wants to handle synchronously and leave their signal handlers in the default state. The task can then make repeated calls to *sigtimedwait***( )** to remove any signals that are sent to it.

RETURNS

Upon successful completion (that is, one of the signals specified by *pSet* is pending or is generated) *sigtimedwait***( )** will return the selected signal number. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

ERRNO

EINTR
The wait was interrupted by an unblocked, caught signal.

EAGAIN
No signal specified by *pSet* was delivered within the specified timeout period.

EINVAL
The *pTimeout* argument specified a **tv_nsec** value less than zero or greater than or equal to 1000 million.

SEE ALSO

sigLib

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**

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**

---

*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**

*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**

---

slattach( )

NAME *slattach*( ) – publish the **sl** network interface and initialize the driver and device

SYNOPSIS
```
STATUS slattach
    (
    int   unit,            /* SLIP device unit number            */
    int   fd,              /* fd of tty device for SLIP interface */
    BOOL  compressEnable,  /* explicitly enable CSLIP compression */
    BOOL  compressAllow,   /* enable CSLIP compression on Rx      */
    int   mtu              /* user setable MTU                    */
    )
```

DESCRIPTION This routine publishes the **sl** interface by filling in a network interface record and adding this record to the system list. It also initializes the driver and the device to the operational state. This routine is usually called by *slipInit*( ).

RETURNS OK or ERROR.

SEE ALSO **if_sl**

---

slipBaudSet( )

NAME *slipBaudSet*( ) – set the baud rate for a SLIP interface

SYNOPSIS
```
STATUS slipBaudSet
    (
    int  unit,  /* SLIP device unit number */
    int  baud   /* baud rate               */
    )
```

DESCRIPTION This routine adjusts the baud rate of a tty device attached to a SLIP interface. It provides a way to modify the baud rate of a tty device being used as a SLIP interface.

RETURNS OK, or ERROR if the unit number is invalid or uninitialized.

SEE ALSO **if_sl**

slipDelete( )

NAME *slipDelete*( ) – delete a SLIP interface

SYNOPSIS

```
STATUS slipDelete
    (
    int  unit  /* SLIP unit number */
    )
```

DESCRIPTION This routine resets a specified SLIP interface. It detaches the tty from the **sl** unit and deletes the specified SLIP interface from the list of network interfaces. For example, the following call will delete the first SLIP interface from the list of network interfaces:

```
slipDelete (0);
```

RETURNS OK, or ERROR if the unit number is invalid or uninitialized.

SEE ALSO **if_sl**

slipInit( )

NAME *slipInit*( ) – initialize a SLIP interface

SYNOPSIS

```
STATUS slipInit
    (
    int   unit,           /* SLIP device unit number (0 - 19)          */
    char  *devName,       /* name of the tty device to be initialized  */
    char  *myAddr,        /* address of the SLIP interface             */
    char  *peerAddr,      /* address of the remote peer SLIP interface */
    int   baud,           /* baud rate of SLIP device: 0=don't set rate */
    BOOL  compressEnable, /* explicitly enable CSLIP compression       */
    BOOL  compressAllow,  /* enable CSLIP compression on Rx            */
    int   mtu             /* user set-able MTU                         */
    )
```

DESCRIPTION

This routine initializes a SLIP device. Its parameters specify the name of the tty device, the Internet addresses of both sides of the SLIP point-to-point link (i.e., the local and remote sides of the serial line connection), and CSLIP options.

The Internet address of the local side of the connection is specified in *myAddr* and the name of its tty device is specified in *devName*. The Internet address of the remote side is specified in *peerAddr*. If *baud* is not zero, the baud rate will be the specified value; otherwise, the default baud rate will be the rate set by the tty driver. The *unit* parameter specifies the SLIP device unit number. Up to twenty units may be created.

The parameters *compressEnable* and *compressAllow* determine support for TCP/IP header compression. If *compressAllow* is TRUE (1), CSLIP is enabled only if a CSLIP type packet is received by this device. If *compressEnable* is TRUE (1), CSLIP compression is enabled explicitly for all transmitted packets, and compressed packets can be received.

The MTU option parameter allows the setting of the MTU for the link.

For example, the following call initializes a SLIP device, using the console's second port, where the Internet address of the local host is 192.10.1.1 and the address of the remote host is 192.10.1.2. The baud rate will be the default rate for /tyCo/1. CLSIP is enabled if a CSLIP type packet is received. The MTU of the link is 1006.

```
slipInit (0, "/tyCo/1", "192.10.1.1", "192.10.1.2", 0, 0, 1, 1006);
```

RETURNS

OK, or ERROR if the device cannot be opened, memory is insufficient, or the route is invalid.

SEE ALSO

if_sl

---

smIfAttach( )

NAME

smIfAttach( ) – publish the **sm** interface and initialize the driver and device

SYNOPSIS

```
STATUS smIfAttach
    (
    int          unit,          /* interface unit number  */
    SM_ANCHOR *  pAnchor,       /* local addr of anchor   */
    int          maxInputPkts,  /* max no. of input pkts  */
    int          intType,       /* method of notif.       */
    int          intArg1,       /* interrupt argument #1  */
    int          intArg2,       /* interrupt argument #2  */
    int          intArg3,       /* interrupt argument #3  */
    int          ticksPerBeat,  /* heartbeat freq.        */
    int          numLoan        /* no. of buffers to loan */
    )
```

DESCRIPTION This routine attaches an **sm** Ethernet interface to the network, if the interface exists. This routine makes the interface available by filling in the network interface record. The system will initialize the interface when it is ready to accept packets.

The shared memory region must have been initialized, via *smPktSetup***( )**, prior to calling this routine (typically by an OS-specific initialization routine). The *smIfAttach***( )** routine can be called only once per unit number.

Parameters:

pAnchor
local address by which the local CPU may access the shared memory anchor.

maxInputPkts
maximum number of incoming shared memory packets which may be queued to this CPU at one time.

intType, *intArg1*, *intArg2*, and *intArg3*
allow a CPU to announce the method by which it is to be notified of input packets which have been queued to it.

ticksPerBeat
frequency of the shared memory anchor's heartbeat. The frequency is expressed in terms of the number of CPU ticks on the local CPU corresponding to one heartbeat period.

numLoan
number of shared memory packets available to be loaned out, if non-zero.

RETURNS OK or ERROR.

SEE ALSO **if_sm**

---

*smMemAddToPool***( )**

NAME *smMemAddToPool***( )** – add memory to the 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
: 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
: size in bytes of shared memory added to the partition.

AVAILABILITY This call is 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 an array from the 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 call is 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 the largest free block in the shared memory system partition (VxMP Opt.)

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 call is 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 call is a component of the unbundled shared memory objects support option, VxMP.

RETURNS OK, or ERROR if the block is invalid.

ERRNO **S_memLib_BLOCK_ERROR**
S_smObjLib_LOCK_TIMEOUT

SEE ALSO **smMemLib**, *smMemMalloc*( ), *smMemCalloc*( )

smMemMalloc( )

NAME *smMemMalloc*( ) – allocate a block of memory from the 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 call is 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 the debug options for the 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 manual entry for **smMemLib**.

AVAILABILITY This call is 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 a block of memory from the 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 call is 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 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 the number of blocks, the average block size, and the maximum block size. It also shows the number of blocks currently allocated, and the average allocated block size.

If *type* is 1, it displays a list of all the blocks in the free list of the shared memory system partition.

WARNING: This routine locks access to the shared memory system partition while displaying the information. This can compromise the access time to the partition from other CPUs in the system. Generally, this routine is used for debugging purposes only.

EXAMPLE

```
-> smMemShow 1
FREE LIST:
  num    addr       size
  --- ---------- ----------
    1   0x4ffef0        264
    2   0x4fef18       1700
SUMMARY:
    status        bytes    blocks   ave block  max block
--------------- --------- -------- ---------- ----------
        current
           free      1964        2        982       1700
          alloc      2356        1       2356         -
     cumulative
          alloc      2620        2       1310         -
value = 0 = 0x0
```

AVAILABILITY

This call is a component of the unbundled shared memory objects support option, VxMP.

RETURNS

N / A

SEE ALSO

smMemShow

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** :

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 call is a component of the unbundled shared memory objects support option, VxMP.

RETURNS OK, or ERROR if there is insufficient memory for *name* to be allocated, if *name* is already in the database, or if the database is already full.

ERRNO
S_smNameLib_NOT_INITIALIZED
S_smNameLib_NAME_TOO_LONG
S_smNameLib_NAME_ALREADY_EXIST
S_smNameLib_DATABASE_FULL
S_smObjLib_LOCK_TIMEOUT

SEE ALSO **smNameLib**, **smNameShow**

smNameFind( )

NAME

smNameFind( ) – look up a shared memory object by name (VxMP 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 call is 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 call is a component of the unbundled shared memory objects support option, VxMP.

RETURNS

OK, or ERROR if *value* is not found or if the wait type is invalid.

ERRNO

S_smNameLib_NOT_INITIALIZED
S_smNameLib_VALUE_NOT_FOUND
S_smNameLib_INVALID_WAIT_TYPE
S_smObjLib_LOCK_TIMEOUT

SEE ALSO

smNameLib, **smNameShow**

smNameRemove( )

NAME

smNameRemove( ) – remove object from 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 call is 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 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 call is 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**

---

smNetAttach( )

NAME *smNetAttach*( ) – attach the shared memory network interface

SYNOPSIS

```
STATUS smNetAttach
    (
    int          unit,          /* interface unit number */
    SM_ANCHOR *  pAnchor,       /* addr of anchor        */
    int          maxInputPkts,  /* max queued packets    */
    int          intType,       /* interrupt method      */
    int          intArg1,       /* interrupt argument #1 */
    int          intArg2,       /* interrupt argument #2 */
    int          intArg3        /* interrupt argument #3 */
    )
```

DESCRIPTION This routine attaches the shared memory interface to the network. It is called once by each CPU on the shared memory network. Parameters:

unit
: specifies the backplane unit number.

pAnchor
: local address by which the local CPU may access the shared memory anchor.

maxInputPkts
: maximum number of incoming shared memory packets which may be queued to this CPU at one time.

intType, *intArg1*, *intArg2*, and *intArg3*
allow a CPU to announce the method by which it is to be notified of input packets which have been queued to it.

RETURNS OK, or ERROR if the shared memory interface cannot be attached.

SEE ALSO **smNetLib**

smNetInetGet( )

NAME *smNetInetGet*( ) – get an address associated with a shared memory network interface

SYNOPSIS
```
STATUS smNetInetGet
    (
    char *  smName,  /* device name */
    char *  smInet,  /* return inet */
    int     cpuNum   /* cpu number  */
    )
```

DESCRIPTION This routine returns the Internet address in *smInet* for the CPU specified by *cpuNum* on the shared memory network specified by *smName*. If *cpuNum* is NONE (-1), this routine returns information about the local (calling) CPU.

This routine can only be called after a call to *smNetAttach*( ). It will block if the shared memory region has not yet been initialized.

This routine is only applicable if sequential addressing is being used over the backplane.

RETURNS OK, or ERROR if the Internet address cannot be found.

SEE ALSO **smNetLib**

smNetInit( )

NAME *smNetInit*( ) – initialize the shared memory network driver

SYNOPSIS
```
STATUS smNetInit
    (
    SM_ANCHOR *  pAnchor,      /* local addr of anchor         */
    char *       pMem,         /* local addr of shared memory  */
```

```
int         memSize,      /* size of shared memory        */
BOOL        tasType,      /* TRUE = hardware supports TAS */
int         cpuMax,       /* max numbers of cpus          */
int         maxPktBytes,  /* size of data packets         */
u_long      startAddr     /* beginning address            */
)
```

DESCRIPTION

This routine is called once by the backplane master. It sets up and initializes the shared memory region of the shared memory network and starts the shared memory heartbeat.

The *pAnchor* parameter is the local memory address by which the master CPU accesses the shared memory anchor. *pMem* contains either the local address of shared memory or the value NONE (-1), which implies that shared memory is to be allocated dynamically. *memSize* is the size, in bytes, of the shared memory region.

The *tasType* parameter specifies the test-and-set operation to be used to obtain exclusive access to the shared data structures. It is preferable to use a genuine test-and-set instruction, if the hardware permits it. In this case, *tasType* should be **SM_TAS_HARD**. If any of the CPUs on the backplane network do not support the test-and-set instruction, *tasType* should be **SM_TAS_SOFT**.

The *maxCpus* parameter specifies the maximum number of CPUs that may use the shared memory region.

The *maxPktBytes* parameter specifies the size, in bytes, of the data buffer in shared memory packets. This is the largest amount of data that may be sent in a single packet. If this value is not an exact multiple of 4, it will be rounded up to the next multiple of 4.

The *startAddr* parameter is only applicable if sequential addressing is desired. If *startAddr* is non-zero, it specifies the starting address to use for sequential addressing on the backplane. If *startAddr* is zero, sequential addressing is disabled.

RETURNS

OK, or ERROR if the shared memory network cannot be initialized.

SEE ALSO

smNetLib

---

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**

---

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 if **INCLUDE_SM_OBJ** is defined in **configAll.h**.

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 an error (**S_smLib_DOWN**).

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 call is 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_DOWN**
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.

AVAILABILITY This call is a component of the unbundled shared memory objects support option, VxMP.

RETURNS The local shared memory address pointed to by *globalAdrs*.

SEE ALSO **smObjLib**

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 # of tries to obtain spinLock */
    int            intType,          /* interrupt method                  */
    int            intArg1,          /* interrupt argument #1             */
    int            intArg2,          /* interrupt argument #2             */
    int            intArg3           /* interrupt argument #3             */
    )
```

DESCRIPTION

This routine initializes a shared memory descriptor. The descriptor must already be allocated in the CPU's local memory. Once the descriptor has been initialized by this routine, the CPU may attach itself to the shared memory area by calling *smObjAttach*( ).

This routine is called automatically if **INCLUDE_SM_OBJ** is defined in **configAll.h**.

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).

cpuNum
: the number to be used to identify this CPU during shared memory operations. CPUs are numbered starting with zero for the master CPU, up to 1 less than the maximum number of CPUs defined during the master CPU's *smObjSetup*( ) call. CPUs can attach in any order, regardless of their CPU number.

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. Its default value is 100, but it can be set to a higher value for a heavily loaded system.

intType, *intArg1*, *intArg2*, and *intArg3*
allow a CPU to announce the method by which it is to be notified of shared memory events. See the manual entry for **if_sm** for a discussion about interrupt types and their associated parameters.

AVAILABILITY This call is 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 **INCLUDE_SM_OBJ** is defined in **configAll.h**.

AVAILABILITY This call is 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**

---

*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.

AVAILABILITY This call is a component of the unbundled shared memory objects support option, VxMP.

RETURNS The global shared memory address pointed to by *localAdrs*.

SEE ALSO **smObjLib**

---

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 (processor number 0). It is called automatically only by the master CPU, if **INCLUDE_SM_OBJ** is defined in **configAll.h**.

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 call is 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 ()
```

DESCRIPTION This routine displays useful information about the current status of shared memory objects facilities.

WARNING: The information returned by this routine is not static and may be obsolete by the time it is examined. This information is generally used for debugging purposes only.

EXAMPLE

```
-> smObjShow
Shared Mem Anchor Local Addr: 0x600.
Shared Mem Hdr Local Addr:    0xb1514.
Attached CPU :                5
Max Tries to Take Lock:       1
Shared Object Type    Current    Maximum  Available
------------------- ---------- --------- ----------
Tasks                         1        20         19
Binary Semaphores             8        30         20
Counting Semaphores           2        30         20
Messages Queues               3        10          7
Memory Partitions             1         4          3
Names in Database            16       100         84
```

AVAILABILITY This call is 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**
S_smLib_NOT_ATTACHED

SEE ALSO **smObjShow**, **smObjLib**

smObjTimeoutLogEnable( )

NAME *smObjTimeoutLogEnable*( ) – enable / disable 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 call is a component of the unbundled shared memory objects support option, VxMP.

RETURNS N / A

SEE ALSO **smObjLib**

snattach( )

NAME *snattach*( ) – publish the **sn** network interface and initialize the driver and device

SYNOPSIS
```
STATUS snattach
    (
    int     unit,      /* unit number           */
    char *  pDevRegs,  /* addr of device's regs */
    int     ivec       /* vector number         */
    )
```

DESCRIPTION This routine publishes the **sn** interface by filling in a network interface record and adding this record to the system list. It also initializes the driver and the device to the operational state.

RETURNS OK or ERROR.

SEE ALSO **if_sn**

---

snmpdContinue( )

NAME *snmpdContinue*( ) – continue processing of an SNMP packet

SYNOPSIS

```
void snmpdContinue
    (
    SNMP_PKT_T *  pktp  /* snmp packet */
    );
```

DESCRIPTION This routine continues processing a packet. *snmpdPktProcess*( ) begins the method routines needed to fulfill the request, while this routine determines whether the current set of method routines have finished; it then either starts more method routines or sends a response packet.

If a method routine returns before completing its task, it must arrange for *snmpdContinue*( ) to be called when the task is completed. You must obtain the write-lock on this packet by calling *snmpdPktLockGet*( ) prior to calling *snmpdContinue*( ). *snmpdContinue*( ) releases the write-lock when it completes.

pktp is a pointer to the structure that contains the packet.

RETURNS N / A

SEE ALSO **snmpdLib**

---

snmpdExit( )

NAME *snmpdExit*( ) – exit the SNMP agent

SYNOPSIS

```
void snmpdExit (void)
```

DESCRIPTION This routine causes the SNMP agent to exit. It is available in case the user encounters some problem after a successful startup.

This routine must be called from the main thread. Any other tasks spawned by the user (for asynchronous method routines) should be deleted prior to calling this function.

RETURNS N / A

SEE ALSO **snmpdLib**

*snmpdGroupByGetprocAndInstance***( )**

NAME *snmpdGroupByGetprocAndInstance***( )** – gather set of similar variable bindings

SYNOPSIS

```
void snmpdGroupByGetprocAndInstance
    (
    SNMP_PKT_T *  pktp,      /* snmp packet      */
    VB_T *        firstVbp,  /* first var bind   */
    int           compc,     /* component count  */
    OIDC_T *      compl      /* component length */
    );
```

DESCRIPTION This routine gathers a set of similar variable bindings together by searching the variable bindings in *pktp*, starting at *firstVbp* for a match of the **getproc** pointer of *firstVbp* and with an instance specified by *compc* and *compl*. This routine then links the variable bindings found which match *firsVbp*.

This routine does not set any flags in the variable bindings. It is left to the calling routine to decide whether the *tested* or *set* flags should be set.

RETURNS N / A

SEE ALSO **snmpdLib**

*snmpdInitFinish***( )**

NAME

*snmpdInitFinish***( )** – complete the initialization of the agent

SYNOPSIS

```
void snmpdInitFinish
    (
    VOIDFUNCPTR  pPrivRlse,    /* user's private release routine  */
    FUNCPTR      pSetPduVldt,  /* user's set pdu validate routine */
    FUNCPTR      pPreSet,      /* user's pre set routine          */
    FUNCPTR      pPostSet,     /* user's post set routine         */
    FUNCPTR      pSetFailed    /* user's set failed routine       */
    )
```

DESCRIPTION

This routine is required to be called by the user to complete the initialization of the SNMP agent after the transport endpoint has been initialized. This routine must be called *after the endpoint has been established, since it depends on that endpoint to function correctly*.

This routine also installs any user-supplied hooks for customizing the agent. If a hook is not required, a NULL pointer should be passed in that position. The function *snmpIoTrapSend***( )** is invoked by this routine, so any configuration required by that routine should be done prior to calling *snmpdInitFinish***( )**.

The hook routines are as follows:

pPrivRlse

This routine is used by the agent to free any memory attached to the private field of the **SNMP_PKT_T** structure.

pSetPduVldt

This routine is called before any processing of a **set** request has taken place. It can be used to perform global validation of the request, if needed. The return values for this routine must meet the following requirements:

0 – indicates that the **set** PDU is valid, and processing should continue.

1 – indicates that the **set** PDU is valid, and that this routine has already completed all the required **set** operations.

-1 – indicates that the **set** PDU is invalid and should be rejected.

pPreSet

This routine is called after all **testproc** operations have completed successfully, but before any **setproc** operation is begun. Return value requirements are as follows

0 – indicates that the **set** PDU is valid, and processing should continue.

1 – indicates that the **set** PDU is valid, and that this routine has already completed all the required **set** operations.

-1 – indicates that the **set** PDU is invalid and should be rejected.

pPostSet
This routine is called after all **setproc** have completed successfully. This routine can be used to free resources allocated during **set** processing.

pSetFailed
This routine is possibly called under two sets of circumstances: after all **testproc** operations have returned and some of them have failed, and / or after all **undoproc** operations have returned and some **setproc** operations have failed. It can be used to do any required cleanup.

RETURNS N / A

SEE ALSO **snmpdLib**

snmpdLog( )

NAME *snmpdLog*( ) – log messgaes from the SNMP agent

SYNOPSIS
```
void snmpdLog
    (
    int     level,  /* level of this message */
    char *  string  /* message string        */
    )
```

DESCRIPTION This routine logs messages generated by the SNMP agent. Messages are sent to the standard console. If *level* is less than or equal to **SNMP_TRACE_LEVEL** (in **configAll.h**), the message is printed, otherwise it is ignored. The value of *level* must be one of 1, 2, or 3.

RETURNS N / A

SEE ALSO **snmpdLib**

*snmpdMemoryAlloc***( )**

NAME *snmpdMemoryAlloc***( )** – allocate memory for the SNMP agent

SYNOPSIS

```
void *   snmpdMemoryAlloc
    (
    size_t  size
    )
```

DESCRIPTION This routine allocates memory for the SNMP agent. The required size of the block is passed in *size*. This memory must be deallocated later with *snmpdMemoryFree***( )**.

RETURNS a pointer to the allocated buffer on success, otherwise NULL.

SEE ALSO **snmpIoLib**, *snmpdMemoryFree***( )**

*snmpdMemoryFree***( )**

NAME *snmpdMemoryFree***( )** – free memory allocated by the SNMP agent

SYNOPSIS

```
void   snmpdMemoryFree
    (
    void *  pBuf  /* buffer to free */
    )
```

DESCRIPTION This routine deallocates memory which was previously allocated by the SNMP agent with *snmpdMemoryAlloc***( )**.

RETURNS N / A

SEE ALSO **snmpIoLib**, *snmpdMemoryAlloc***( )**

snmpdPktLockGet( )

NAME *snmpdPktLockGet*( ) – lock an SNMP packet

SYNOPSIS

```
STATUS  snmpdPktLockGet
    (
    SNMP_PKT_T *  pktp  /* snmp packet */
    )
```

DESCRIPTION This routine obtains a lock on the SNMP packet being processed, which must be obtained by any asynchronous method routine prior to calling any of the routines **getproc_***, **nextproc_***, **testproc_***, or **setproc_***, or also *snmpdContinue*( ). *snmpdContinue*( ) releases the lock before returning. No other routine can release this lock.

This routine blocks until the lock is obtained.

RETURNS OK, or ERROR on failure.

SEE ALSO **snmpdLib**, **snmpProcLib**, *snmpdContinue*( )

snmpdPktProcess( )

NAME *snmpdPktProcess*( ) – process a packet returned by the transport

SYNOPSIS

```
void snmpdPktProcess
    (
    int     pktSize,       /* packet length            */
    char *  pBuf,          /* packet buffer            */
    void *  pRemoteAddr,   /* remote transport address */
    void *  pLocalAddr,    /* local transport address  */
    void *  pSnmpEndpoint  /* snmp transport end point */
    )
```

DESCRIPTION This routine is invoked by the user-IO layer to process a received packet. The buffer *pBuf* (provided by the agent designer) must contain a packet of size *pktSize*. *pRemoteAddr* specifies the source address of the sending machine; *pLocalAddr* specifies the address of the receiver; and *pSnmpEndpoint* specifies the transport endpoint. *pRemoteAddr*, *pLocalAddr*, and *pSnmpEndpoint* are passed to user-provided transport routines.

RETURNS N / A

SEE ALSO **snmpdLib**

snmpdTrapSend( )

NAME

snmpdTrapSend( ) – general interface to trap facilities

SYNOPSIS

```
void snmpdTrapSend
    (
    void *    pSnmpEndpoint,    /* snmp agent transport endpoint     */
    int       numDestn,         /* number of destinations            */
    void **   ppDestAddrTbl,    /* array of ptrs to destinations     */
    void *    pLocalAddr,       /* local address                     */
    int       version,          /* SNMP version                      */
    char *    pTrapCmnty,       /* trap community string             */
    OIDC_T *  pMyOid,           /* agent object identifier           */
    int       myOidLen,         /* length of agent object identifier */
    u_long *  pIpAddr,          /* ip address of sender              */
    int       trapType,         /* trap type                         */
    int       trapSpecific,     /* trap specific code                */
    int       numVarBinds,      /* number of varbinds in packet      */
    FUNCPTR   trapVarBindsRtn,  /* routine to bind varbinds          */
    void *    pCookie           /* argument to binding routine       */
    )
```

DESCRIPTION

This routine sends a trap of type *trapType* and specific *trapSpecific* to all the specified destinations, given an array of destinations in *ppDestAddrTbl* of len *numDestn*.

The *version* parameter is either **SNMP_VERSION_1** or **SNMP_VERSION_2**, depending on whether a v1-style or v2-style trap is to be generated. The community used is specified by *pTrapCmnty*. *pLocalAddr* indicates a local address to use for a sending endpoint. *pMyOid* and *myOidlen* specify the system object identifier (*sysObjId*) to use.

numVarBinds indicates the number of variable bindings that are to be encoded in the packet. *trapVarBindsRtn* is the routine to use for doing the variable bindings. *pCookie* is passed to this routine as shown below:

```
(*trapVarBindsRtn) (pPkt, pCookie)
```

RETURNS

N / A

SEE ALSO

snmpdLib

snmpdTreeAdd( )

NAME *snmpdTreeAdd*( ) – dynamically add a subtree to the SNMP agent MIB tree

SYNOPSIS

```
STATUS  snmpdTreeAdd
    (
    char *       pTreeOidStr,
    MIBNODE_T *  pTreeAddr
    )
```

DESCRIPTION This routine adds the specified MIB subtree, located in memory at the address *pTreeAddr*, to the agent's MIB data tree at the node with the object identifier specified by *pTreeOidStr*. This subtree is normally generated with the **-start** option to **mibcomp**.

RETURNS OK on success, otherwise ERROR.

SEE ALSO **snmpdLib**, *WindNet SNMPv1v2c VxWorks Component Release Supplement*

snmpdTreeRemove( )

NAME *snmpdTreeRemove*( ) – dynamically remove part of the SNMP agent MIB tree

SYNOPSIS

```
void snmpdTreeRemove
    (
    char *  pTreeOidStr  /* char string specifying oid of tree to remove */
    )
```

DESCRIPTION This routine deletes part of the SNMP agent MIB tree at runtime. Once the specified subtree is deleted, any further requests to access objects of that subtree fail.

RETURNS N / A

SEE ALSO **snmpdLib**, *WindNet SNMPv1v2c VxWorks Component Release Supplement*

*snmpdVbExtractRowLoose***( )**

NAME

*snmpdVbExtractRowLoose***( )** – incrementally extract pieces of a row for a set

SYNOPSIS

```
VB_T * snmpdVbExtractRowLoose
    (
    SNMP_PKT_T *  pktp,    /* snmp packet      */
    int           indx,    /* index            */
    MIBLEAF_T **  leaves,  /* mib leaves       */
    int           compc,   /* component count  */
    OIDC_T *      compl    /* component length */
    );
```

DESCRIPTION

This routine assists in implementing row-creation **set** operations, especially in "dribble" creation. In dribble creation, the network management station may choose to break the creation of a new entry in a table into multiple packets. Using multiple packets does not work with *snmpdVbRowExtract***( )**, which requires at least one item to be in the packet.

In dribble creation, there is no single item that is guaranteed to be in every packet. As its name implies, this routine is similar to *snmpdVbRowExtract***( )**. It searches the variable bindings in *pktp* starting at *indx* for a match of the MIB leaf pointer specified in the *leaves* array and with an instance specified by *compc* and *compl*. The routine then links the variable bindings found from the last variable binding found.

If no variable bindings are found, this routine returns a NULL pointer. This routine does not set any of the flags in the variable bindings. It is left to the calling routine to decide if the *tested* or *set* flags should be set.

RETURNS

A pointer to the last variable binding if successful, otherwise NULL.

SEE ALSO

snmpdLib

*snmpdVbRowExtract***( )**

NAME

*snmpdVbRowExtract***( )** – extract required pieces of a row for a **set** operation

SYNOPSIS

```
VB_T *   snmpdVbRowExtract
    (
    SNMP_PKT_T *        pktp,               /* snmp packet          */
    int                 start_index,        /* starting index       */
    int                 compc,              /* component count      */
```

```
    OIDC_T *            compl,                  /* component length       */
    int                 row_structure_length,   /* length of row structure */
    struct create_row *  row                    /* row structure          */
    );
```

DESCRIPTION This routine assists in implementing row-creation **set** operations. It is typically used by the **testproc** routine for a table. It scans the SNMP packet looking for all the pieces that go together for the purposes of creating a new row in an SNMP table.

The parameter *row* describes what this routine is looking for. *row* is a list of MIB leaf pointers referring to variables in the table in question, and a flag indicating if this variable (*column*) in the table is required. This routine searches the variable bindings in *pktp* for a match of the MIB leaf pointer specified in the row array and the instance specified by *compc* and *compl*. It links the variable bindings found from the first MIB leaf node in the row array. This first variable binding must be in the packet whether it is marked as needed or not, and is the return value of *snmpdVbRowExtract***( )**.

If *snmpdVbRowExtract***( )** does not find a matching variable binding and that leaf is flagged **ROW_FLAG_NEEDED** in the row array, the routine returns NULL to indicate error.

This routine marks the variable bindings in the resulting list as already tested except for the first entry, which is marked as already set. As a result, the agent only calls the **set** routine associated with the first MIB leaf in the row array. This **set** routine should handle creation of the necessary data structures and reading the variable binding list to execute the required **set** operations. You can force other entries set routines to be called by flagging entries with **ROW_FLAG_CALL_SET**.

RETURNS A pointer to the first variable binding if successful, otherwise NULL.

SEE ALSO **snmpdLib**

---

*snmpdViewEntryRemove***( )**

NAME *snmpdViewEntryRemove***( )** – remove an entry from the view table

SYNOPSIS

```
void snmpdViewEntryRemove
    (
    OIDC_T *   pTreeOid,    /* oid of view subtree to remove */
    int        treeOidLen,  /* length of view subtree oid    */
    UINT_16_T  index        /* index of entry                */
    )
```

DESCRIPTION This routine removes an entry from the view table and deallocates the associated resources. The entry should have been previously created with *snmpdViewEntrySet***( )**.

RETURNS N / A

SEE ALSO **snmpdLib**, *snmpdViewEntrySet***( )**

*snmpdViewEntrySet***( )**

NAME *snmpdViewEntrySet***( )** – install an entry in the view table

SYNOPSIS

```
STATUS snmpdViewEntrySet
    (
    OIDC_T *   pTreeOid,    /* sub tree for view                */
    int        treeOidLen,  /* length of subtree oid            */
    UINT_16_T  index,       /* index of entry                   */
    uchar_t *  pMask,       /* mask for entry                   */
    int        maskLen,     /* mask  length in bytes            */
    int        viewType     /* type of view (INCLUDED/EXCLUDED) */
    )
```

DESCRIPTION This function creates an entry in the view table. The view subtree is specified by *pTreeOid* and *treeOidLen*.

The installed entry has an index of *index*. The specified view is included in or excluded from the view table depending on the value of *viewType* (**VIEW_INCLUDED** or **VIEW_EXCLUDED**, respectively). The entry mask is specified by *pmask*; *maskLen* must be the mask length in bytes.

RETURNS OK on success, otherwise ERROR

SEE ALSO **snmpdLib**

*snmpIoClose***( )**

NAME *snmpIoClose***( )** – close the transport endpoint

SYNOPSIS

```
void snmpIoClose (void)
```

DESCRIPTION This routine is invoked to deallocate the transport endpoint. It is invoked from the task deletion hook and also from *snmpdExit***( )**.

RETURNS N / A

SEE ALSO **snmpIoLib**

snmpIoCommunityValidate( )

NAME *snmpIoCommunityValidate*( ) – sample community validation routine

SYNOPSIS

```
int snmpIoCommunityValidate
    (
    SNMP_PKT_T *  pPkt,         /* ptr to snmp pkt */
    SNMPADDR_T *  pRemoteAddr,  /* remote address  */
    SNMPADDR_T *  pLocalAddr    /* local address   */
    )
```

DESCRIPTION This routine is used to set up the view-index field in the SNMP packet. This product is shipped with defaults such that the “priv”community is allowed to **set** variables, and the “pub” community is allowed to **get** variables.

The agent designer must write this function according to the design of the application.

RETURNS 0 if the community is acceptable, otherwise 1.

SEE ALSO **snmpIoLib**

snmpIoInit( )

NAME *snmpIoInit*( ) – initialization routine for SNMP transport endpoint

SYNOPSIS

```
STATUS snmpIoInit ()
```

DESCRIPTION This is the routine to called to initialize the transport endpoint used by the SNMP agent. This routine is invoked from *snmpIoMain*( ). This implementation is for a socket based system.

GLOBALS **snmpSocket**

RETURNS ERROR if unable to create bound socket else OK.

SEE ALSO **snmpIoLib**, *snmpIoMain*( )

snmpIoMain( )

NAME *snmpIoMain*( ) – main SNMP I/O routine

SYNOPSIS
```
void snmpIoMain ()
```

DESCRIPTION This routine is invoked by the agent after it has successsfully completed the preliminary initializations. In this routine, the user is required to initialize the transport endpoint and the views, and then complete initialization of the agent by calling *snmpdInitFinish*( ). Any configuration required for *snmpIoTrapSend*( ) must be done before calling *snmpdInitFinish*( ) since it will be invoked in there. Any hooks that are required by the user must be passed to *snmpdInitFinish*( ). The transport endpoint must be initialized before *snmpdInitFinish*( ) is called; the main loop is then invoked.

RETURNS This routine should never return, except on failure.

SEE ALSO **snmpIoLib**, *snmpdInitFinish*( )

snmpIoTrapSend( )

NAME *snmpIoTrapSend*( ) – send a standard SNMP or MIB-II trap

SYNOPSIS
```
void snmpIoTrapSend
    (
    int  trapType,
    int  trapSpecific
    )
```

DESCRIPTION This routine sends a standard SNMP or MIB-II trap message to the network. It is called by the SNMP agent at startup (to indicate a cold start) and when interface states change. It takes two arguments: *trapType*, the trap type, and *trapSpecific*, the user-defined specifics on this trap.

The agent designer must rewrite this according to specific transport needs.

RETURNS N/A

SEE ALSO **snmpIoLib**, *snmpdTrapSend*( )

snmpIoWrite( )

NAME *snmpIoWrite*( ) – write a packet to the transport

SYNOPSIS
```
void snmpIoWrite
    (
    void *  pSocket,
    char *  pBuf,
    int     bufSize,
    void *  remote,
    void *  local
    )
```

DESCRIPTION This routine writes a datagram to the socket. The routine calls *sendto*( ) with flags always set to 0. The *local* parameter is not used in this case, but exists for conformance with our transport-independent interface.

RETURNS N / A

SEE ALSO **snmpIoLib**, *sendto*( )

SNMP_Bind_64_Unsigned_Integer( )

NAME *SNMP_Bind_64_Unsigned_Integer*( ) – bind a 64-bit unsigned-integer variable

SYNOPSIS
```
int SNMP_Bind_64_Unsigned_Integer
    (
    SNMP_PKT_T *  pktp,       /* internal representation of snmp packet */
    int           index,      /* index of varbind entry                 */
    int           compc,      /* component count                        */
    OIDC_T *      compl,      /* component length                       */
    OCTET_T       typeFlags,  /* type flags                             */
    UINT_32_T     high,       /* high 32 bits of value                  */
    UINT_32_T     low         /* low 32 bits of value                   */
    );
```

DESCRIPTION This routine binds a 64-bit unsigned-integer variable in the variable-binding list of an SNMP packet structure. Parameters:

pktp references the packet structure.

index a zero-based index indicating which variable-binding entry is to be used.

compc and *compl*
the component count and components, respectively, of the object identifier of the variable being bound.

typeFlags
the manifest constant **VT_COUNTER64**.

high and *low*
the high and low 32-bit sections of the 64-bit value to be bound.

A macro form of this procedure may be more convenient: *SNMP_Bind_Counter64***( )**.

RETURNS 0 if successful, otherwise -1.

SEE ALSO **snmpBindLib**

---

*SNMP_Bind_Integer***( )**

NAME *SNMP_Bind_Integer***( )** – bind an integer variable

SYNOPSIS

```
int SNMP_Bind_Integer
    (
    SNMP_PKT_T *  pktp,   /* internal representation of snmp packet */
    int           index,  /* index of varbind entry                 */
    int           compc,  /* component count                        */
    OIDC_T *      compl,  /* component length                       */
    INT_32_T      value   /* value for varbind                      */
    );
```

DESCRIPTION This routine binds an integer variable in the variable-binding list of an SNMP packet structure. Parameters:

pktp references the packet structure.

index a zero-based index indicating which variable-binding entry is to be used.

compc and *compl*
the component count and components, respectively, of the object identifier of the variable being bound.

value the integer value to be bound.

RETURNS 0 if successful, otherwise -1.

SEE ALSO **snmpBindLib**

SNMP_Bind_IP_Address( )

NAME *SNMP_Bind_IP_Address*( ) – bind an IP address variable

SYNOPSIS

```
int SNMP_Bind_IP_Address
    (
    SNMP_PKT_T *  pktp,    /* internal representation of snmp packet */
    int           index,   /* index of varbind entry                 */
    int           compc,   /* component count                        */
    OIDC_T *      compl,   /* component length                       */
    OCTET_T *     pIpAddr  /* ip address                             */
    );
```

DESCRIPTION This routine binds an IP address variable in the variable-binding list of an SNMP packet structure. Parameters:

pktp references the packet structure.

index a zero-based index indicating which variable-binding entry is to be used.

compc and *compl*
the component count and components, respectively, of the object identifier of the variable being bound.

pIpAddr a pointer to a four-byte area containing the IP address to be bound.

The four bytes contain the network address in standard TCP/IP network-byte order.

RETURNS 0 if successful, otherwise -1.

SEE ALSO **snmpBindLib**

SNMP_Bind_Null( )

NAME *SNMP_Bind_Null*( ) – bind a null-valued variable

SYNOPSIS

```
int SNMP_Bind_Null
    (
    SNMP_PKT_T *  pktp,   /* internal representation of snmp packet */
    int           index,  /* index of varbind entry                 */
    int           compc,  /* component count                        */
    OIDC_T *      compl   /* component length                       */
    );
```

DESCRIPTION This routine binds a null-valued variable in the variable-binding list of an SNMP packet structure.

pktp references the packet structure.

index a zero-based index indicating which variable-binding entry is to be used.

compc and *compl*
the component count and components, respectively, of the object identifier of the variable being bound.

RETURNS 0 if successful, otherwise -1.

SEE ALSO **snmpBindLib**

SNMP_Bind_Object_ID( )

NAME *SNMP_Bind_Object_ID*( ) – bind an object-identifier variable

SYNOPSIS

```
int SNMP_Bind_Object_ID
    (
    SNMP_PKT_T *  pktp,   /* internal representation of snmp packet */
    int           index,  /* index of varbind entry                 */
    int           compc,  /* component count                        */
    OIDC_T *      compl,  /* component length                       */
    int           valc,   /* varbind value count                    */
    OIDC_T *      vall    /* varbind value length                   */
    );
```

DESCRIPTION This routine binds an object-identifier variable in the variable-binding list of an SNMP packet structure.

pktp references the packet structure.

index a zero-based index indicating which variable-binding entry is to be used.

compc and *compl*
the component count and components, respectively, of the object identifier of the variable being bound.

valc and *vall*
the component count and components, respectively, of the value being bound.

RETURN 0 if successful, otherwise -1.

SEE ALSO **snmpBindLib**

*SNMP_Bind_String***( )**

NAME

*SNMP_Bind_String***( )** – bind a string variable

SYNOPSIS

```
int SNMP_Bind_String
    (
    SNMP_PKT_T *  pktp,       /* internal representation of snmp packet */
    int           index,      /* index of varbind entry                 */
    int           compc,      /* component count                        */
    OIDC_T *      compl,      /* component length                       */
    OCTET_T       typeFlags,  /* type flags                             */
    int           leng,       /* string length                          */
    OCTET_T *     strp,       /* pointer to string                      */
    int           statflg     /* static memory flag                     */
    );
```

DESCRIPTION

This routine binds an octet-string variable in the variable-binding list of an SNMP packet structure.

pktp references the packet structure.

index a zero-based index indicating which variable-binding entry is to be used.

compc and *compl*
the component count and components, respectively, of the object identifier of the variable being bound.

typeFlags
one of the following manifest constants: **VT_STRING** or **VT_OPAQUE.**

leng and *strp*
the length and address, respectively, of the string to be bound.

statflg indicates whether the string must be copied or whether it may be used in its current location. A value of 0 means copy, 1 means “use as is.’

A macro form of this routine may be more convenient: *SNMP_Bind_Opaque***( )**.

RETURNS

0 if successful, otherwise -1.

SEE ALSO

snmpBindLib

*SNMP_Bind_Unsigned_Integer***( )**

NAME

*SNMP_Bind_Unsigned_Integer***( )** – bind an unsigned-integer variable

SYNOPSIS

```
int SNMP_Bind_Unsigned_Integer
    (
    SNMP_PKT_T *  pktp,       /* internal representation of snmp packet */
    int           index,      /* index of varbind entry                 */
    int           compc,      /* component count                        */
    OIDC_T *      compl,      /* component length                       */
    OCTET_T       typeFlags,  /* type flags                             */
    UINT_32_T     value       /* value for varbind                      */
    );
```

DESCRIPTION

This routine binds an unsigned-integer variable in the variable-binding list of an SNMP packet structure.

pktp references the packet structure.

index a zero-based index indicating which variable-binding entry is to be used.

compc and *compl*
the component count and components, respectively, of the object identifier of the variable being bound.

typeFlags
one of the following manifest constants: **VT_COUNTER**, **VT_GAUGE**, or **VT_TIMETICKS**.

value
the unsigned-integer value to be bound.

Macro forms of this procedure may be more convenient: *SNMP_Bind_Gauge***( )**, *SNMP_Bind_Timeticks***( )**, and *SNMP_Bind_Counter***( )**.

RETURNS

0 if successful, otherwise -1.

SEE ALSO

snmpBindLib

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*

socket( )

NAME *socket*( ) – open a socket

SYNOPSIS

```
int socket
    (
    int  domain,   /* address family (e.g. 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 connection-based (stream) socket.

SOCK_DGRAM datagram (UDP) socket.

SOCK_RAW raw socket.

RETURNS A socket descriptor, or ERROR.

SEE ALSO **sockLib**

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

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*( ).

NOTE: This routine is now ANSI compatible. Previous VxWorks versions of *sprintf*( ) returned the number of characters put into *buffer* including the null termination. The ANSI C standard specifies that the null character should not be included in the returned count. Thus, the return value will be one less than in previous versions.

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 **INCLUDE_SPY** is defined in **configAll.h**, 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*.

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**

squeeze( )

NAME *squeeze*( ) – reclaim fragmented free space on an RT-11 volume

SYNOPSIS
```
STATUS squeeze
    (
    char  *devName  /* RT-11 device to squeeze, e.g., "/fd0/" */
    )
```

DESCRIPTION This command moves data around on an RT-11 volume so that any areas of free space are merged.

NOTE: No device files should be open when this procedure is called. The subsequent condition of such files would be unknown and writing to them could corrupt the entire disk.

RETURNS OK, or ERROR if the device cannot be opened or squeezed.

SEE ALSO **usrLib**, *VxWorks Programmer's Guide: Target Shell*

sr( )

NAME *sr*( ) – return the contents of the status register (MC680x0)

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

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

RETURNS The contents of the status register.

SEE ALSO **dbgArchLib**, *VxWorks Programmer's Guide: Target Shell*

sramDevCreate( )

NAME *sramDevCreate*( ) – create a PCMCIA memory disk device

SYNOPSIS

```
BLK_DEV *sramDevCreate
    (
    int  sock,          /* socket no.                              */
    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 PCMCIA memory disk device.

RETURNS A pointer to a block device structure (**BLK_DEV**), or NULL if memory cannot be allocated for the device structure.

SEE ALSO **sramDrv**, *ramDevCreate*( )

sramDrv( )

NAME s*ramDrv*( ) – install a PCMCIA SRAM memory driver

SYNOPSIS

```
STATUS sramDrv
    (
    int  sock  /* socket no. */
    )
```

DESCRIPTION This routine initializes a PCMCIA SRAM memory driver. It must be called once, before any other routines in the driver.

RETURNS OK, or ERROR if the I/O system cannot install the driver.

SEE ALSO **sramDrv**

sramMap( )

NAME *sramMap*( ) – map PCMCIA memory onto a specified ISA address space

SYNOPSIS

```
STATUS sramMap
    (
    int  sock,    /* socket no.              */
    int  type,    /* 0: common  1: attribute */
    int  start,   /* ISA start address       */
    int  stop,    /* ISA stop address        */
    int  offset,  /* card offset address     */
    int  extraws  /* extra wait state        */
    )
```

DESCRIPTION This routine maps PCMCIA memory onto a specified ISA address space.

RETURNS OK, or ERROR if the memory cannot be mapped.

SEE ALSO **sramDrv**

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** 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 *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 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. 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**)

---

*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. Upon 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**

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. This routine is included automatically when **INCLUDE_SHOW_ROUTINES** is defined in **configAll.h**.

RETURNS OK, or ERROR if an error occurs installing the file pointer show routine.

SEE ALSO **ansiStdio**

strace( )

NAME *strace*( ) – print STREAMS trace messages (STREAMS Opt.)

SYNOPSIS
```
strace
    (
    char  *arg
    )
```

DESCRIPTION This routine, when invoked without arguments, writes all STREAMS event trace messages from all drivers and modules to a file named **streams.log** in the directory **STREAMS_STRACE_OUTPUT_DIR**. This directory constant is set in **configAll.h**. If the module name is provided to *strace*( ) then its output is sent to a file named **module-id.log** in the **STREAMS_STRACE_OUTPUT_DIR** directory.

These messages are obtained from the STREAMS log driver. If arguments are provided, they must be in triplets of the form *mid*, *sid*, and *level*, where *mid* is a STREAMS module ID number, *sid* is a sub-ID number, and *level* is a tracing priority level. Each triplet indicates that tracing messages are to be received from the specified module/driver, sub-ID (usually indicating minor device), and priority level equal to or less than the specified level. The token *all* can be used for any member to indicate no restriction for that attribute.

For additional information, see the *WindNet STREAMS for Tornado Component Release*

RETURNS N/A

WindNet STREAMS for Tornado Component Release

SEE ALSO **straceLib**

straceStop( )

NAME *straceStop*( ) – stop the *strace*( ) task (STREAMS Opt.)

SYNOPSIS
```
void straceStop ( void )
```

DESCRIPTION This routine deletes the *strace*( ) task from the system and performs a cleanup operation. It removes the file descriptor associated with *strace*( ) and flushes any remaining trace-log messages from the log device to the *strace*( ) output file.

SEE ALSO **straceLib**, *strace*( )

RETURNS N / A

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**

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*( )

strerr( )

NAME *strerr*( ) – STREAMS error logger task (STREAMS Opt.)

SYNOPSIS

```
void strerr ( void )
```

DESCRIPTION This routine receives error log messages from the STREAMS log driver and appends them to a log file. The error log files reside in the directory **STREAMS_STRERR_OUTPUT_DIR** specified in **configAll.h**. The error log message format is a string of the following fields:

seq time ticks flags mid sid text

as defined below:

seq	error sequence number
time	time of message in hh:mm:ss
ticks	time of message in machine ticks since boot
flags	T indicates the message was also sent to a tracing task;F also indicates fatal error
mid	module ID number of source
sid	sub-ID number of source
text	formatted text of the error message

RETURNS N / A

SEE ALSO **strerrLib**

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.

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 code    */
    char *  buffer    /* string buffer */
    )
```

DESCRIPTION This routine maps the error number in *errcode* to an error message string. It stores the error string in *buffer*. This routine is the POSIX reentrant version of *strerror*( ).

INCLUDE FILES **string.h**

RETURNS OK or ERROR.

SEE ALSO **ansiString**, *strerror*( )

strerrStop( )

NAME *strerrStop*( ) – stop the *strerr*( ) task (STREAMS Opt.)

SYNOPSIS

```
strerrStop ( void )
```

DESCRIPTION This routine deletes the *strerr*( ) task from the system and performs a cleanup operation. It removes the file descriptor associated with *strerr*( ) and flushes any remaining error-log messages from the log device to the *strerr*( ) output file.

SEE ALSO **strerrLib**, strerr

RETURNS N / A

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 (00-61).
%U	the week number of the year (first Sunday as the 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.
%%	%.

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**

strmBandShow( )

NAME *strmBandShow*( ) – display messages in a particular band (STREAMS Opt.)

SYNOPSIS
```
void strmBandShow
    (
    char *     msg,  /* user specified Message to display      */
    queue_t *  q,    /* queue pointer to display messages from */
    int        pri   /* priority Band to display               */
    )
```

DESCRIPTION This routine displays information about all messages in a particular band. It displays the message type, band, and length.

RETURNS N / A

SEE ALSO **strmShow**

strmDebugInit( )

NAME *strmDebugInit*( ) – include STREAMS debugging facility in VxWorks (STREAMS Opt.)

SYNOPSIS
```
STATUS strmDebugInit ()
```

DESCRIPTION This routine includes the STREAMS debugging facility in VxWorks. It is called in **usrNetwork.c** to ensure the inclusion of this library into VxWorks.

RETURNS N / A

SEE ALSO **strmShow**

strmDriverAdd( )

NAME *strmDriverAdd*( ) – add a STREAMS driver to the STREAMS subsystem (STREAMS Opt.)

SYNOPSIS
```
STATUS strmDriverAdd
    (
    char *              pStrmDriverName,  /* driver name                  */
    struct streamtab *  pStreamtab,       /* ptr to driver streamtab      */
    int                 cloneFlag,        /* TRUE = clone device          */
    struct streamtab *  pBuddyStreamtab,  /* ptr to driver buddy streamtab */
    int                 flags,            /* open style SVR3 or SVR4      */
    int                 sqlvl             /* synchronization level        */
    )
```

DESCRIPTION This routine adds a STREAMS driver into the STREAMS subsystem. The input parameters are the driver name, the streamtab of the driver, the clone flag (to specify clone devices or non-clone devices), the streamtab for the writer buddy, the open style type supported by driver, and the synchronization level. For more information about writer buddies, see the *WindNet STREAMS Optional Component Supplement*.

RETURNS OK if successful, otherwise ERROR.

ERRNO

S_strmLib_INVALID_OPEN_STYLE
Invalid *flags* parameter. Must be **SVR3_STYLE_OPEN** or **SVR4_STYLE_OPEN**.

S_strmLib_INVALID_SYNC_LEVEL
Invalid synchronization-level parameter.

SEE ALSO **strmLib**

strmDriverModShow( )

NAME *strmDriverModShow*( ) – list configuration info for modules and devices (STREAMS Opt.)

SYNOPSIS
```
void strmDriverModShow
    (
    int  format
    )
```

DESCRIPTION This routine displays configuration information about the STREAMS modules and devices. It displays the following information under the respective column heads.

Name	name of the device
Type	STREAMS module or STREAMS device
Major	device major number, applicable only to devices
idnum	module ID number
idname	name as specified in the **module_info** structure
minpsz	minimum packet size allowed in the module or driver
maxpsz	maximum packet size allowed in the module or driver
lowat	low water mark
hiwat	high water mark

RETURNS N / A

SEE ALSO **strmShow**

strmMessageShow( )

NAME *strmMessageShow*( ) – display info about all messages in a stream (STREAMS Opt.)

SYNOPSIS
```
void strmMessageShow
    (
    queue_t *  q  /* pointer to a queue in the stream */
    )
```

DESCRIPTION This routine displays information about all messages in a stream. The information shown is the message type, band, and size, and the address of the message block.

RETURNS N / A

SEE ALSO **strmShow**

strmMkfifo( )

NAME *strmMkfifo*( ) – create a STREAMS FIFO (STREAMS Opt.)

SYNOPSIS
```
int strmMkfifo()
```

DESCRIPTION This routine creates an I/O mechanism called a first-in-first-out (FIFO) and returns a file descriptor. The file associated with the file descriptor is a stream. The data written to the file descriptor can be read on a first-in-first-out basis by making read calls to the same file descriptor.

RETURNS A file descriptor if successful, otherwise -1.

ERRNO Errors from *open*( ).

SEE ALSO **strmLib**

strmModuleAdd( )

NAME *strmModuleAdd*( ) – add a STREAMS module to the STREAMS subsystem (STREAMS Opt.)

SYNOPSIS
```
STATUS strmModuleAdd
    (
    char *              pModuleName,      /* module name             */
    struct streamtab *  pStreamtab,       /* ptr to Module streamtab */
    struct streamtab *  pBuddyStreamtab,  /* ptr to Buddy streamtab  */
    int                 flags,            /* open style SVR3 or SVR4 */
    int                 sqlvl             /* synchronization level   */
    )
```

DESCRIPTION This routine installs a STREAMS module into the STREAMS subsystem. The input parameters are the module name, the streamtab of the driver, the streamtab of the writer buddy, the open style type supported by the module, and the synchronization level.

For more information about writer buddies, see the *WindNet STREAMS Optional Component Supplement*.

RETURNS OK if successful, otherwise ERROR.

ERRNO

S_strmLib_INVALID_OPEN_STYLE
Invalid *flags* parameter. Must be **SVR3_STYLE_OPEN** or **SVR4_STYLE_OPEN**.

S_strmLib_INVALID_SYNC_LEVEL
Invalid *sqlvl* level parameter.

SEE ALSO **strmLib**

strmMsgStatShow( )

NAME *strmMsgStatShow*( ) – display statistics about system-wide usage of message blocks (STREAMS Opt.)

SYNOPSIS
```
void strmMsgStatShow ( void )
```

DESCRIPTION This routine displays statistics about the system-wide usage of STREAMS message blocks. The statistics shown are:

– the size of the message block,

– the number of message blocks available for this size,

– the number of message blocks of this size in use,

– the total number of message blocks allocated for this size,

– the total memory used by all the message blocks of this size,

– the number of calls to *allocb*( ) that have failed for this message-block size,

– the number of *bufcall*( ) requests pending due to *allocb*( ) failure for message blocks of this size.

This routine also prints a brief summary of:

– the number of message blocks allocated for all sizes,

– the number of data blocks allocated for all sizes,

– the total memory used by STREAMS buffers system-wide.

RETURNS N / A

SEE ALSO **strmShow**

strmOpenStreamsShow( )

NAME *strmOpenStreamsShow*( ) – display all open streams in the STREAMS subsystem (STREAMS Opt.)

SYNOPSIS

```
void strmOpenStreamsShow
    (
    char *  msg  /* message to display */
    )
```

DESCRIPTION This routine displays information about all open streams in the system. If *msg* is not NULL, it is displayed before the open streams information. This routine displays the following information: address of the stream, flags, major number associated with the stream, and minor number associated with the stream

RETURNS N / A

SEE ALSO **strmShow**

strmPipe( )

NAME *strmPipe*( ) – create an intertask channel (STREAMS Opt.)

SYNOPSIS

```
int strmPipe
    (
    int  *fds  /* pointer to an array of file descriptors */
    )
```

DESCRIPTION This routine creates an I / O mechanism called a pipe and returns two file descriptors, **fds[0]** and **fds[1]**. The files associated with **fds[0]** and **fds[1]** are streams and are opened for reading and writing.

A read from **fds[0]** accesses the data written to **fds[1]** on a first-in-first-out (FIFO) basis. A read from **fds[1]** accesses the data written to **fds[0]**, also on a FIFO basis.

RETURNS 0 if successful, otherwise -1.

ERRNO Errors from *open*( ).

SEE ALSO **strmLib**

strmQueueShow( )

NAME *strmQueueShow*( ) – display all queues in a particular stream (STREAMS Opt.)

SYNOPSIS
```
int strmQueueShow
    (
    STHP    sth,  /* pointer to the Stream Head */
    char *  msg   /* message to display         */
    )
```

DESCRIPTION This routine displays all queues on the stream *sth*. If *msg* is not NULL, it is displayed before the rest of the information.

RETURNS N / A

SEE ALSO **strmShow**

strmQueueStatShow( )

NAME *strmQueueStatShow*( ) – display statistics about queues system-wide (STREAMS Opt.)

SYNOPSIS
```
void strmQueueStatShow ( void )
```

DESCRIPTION This routine displays statistics about the system-wide usage of queues. The statistics shown are:

– the number of queues allocated,

– the number of queues in use,

– the total number of queues configured,

– the maximum number of queues that can be allocated,

– the number of queue allocation failures.

RETURNS N / A

SEE ALSO **strmShow**

strmSleep( )

NAME *strmSleep*( ) – suspend task execution pending occurrence of an event (STREAMS Opt.)

SYNOPSIS
```
int strmSleep
    (
    ulong_t  event  /* sleep event */
    )
```

DESCRIPTION This routine suspends execution of a task to await an event specified in the parameter *event*. STREAMS drivers and modules are only allowed to invoke *strmSleep*( ) during their open and close procedures. STREAMS components that invoke *strmSleep*( ) can be awakened by *strmWakeup*( ) after the awaited event has occurred.

RETURNS N / A

SEE ALSO **strmLib**, *strmWakeup*( )

strmSockDevNameGet( )

NAME *strmSockDevNameGet*( ) – get the transport-provider device name (STREAMS Opt.)

SYNOPSIS
```
char * strmSockDevNameGet
    (
    int  family,  /* Address family of the transport provider */
    int  type     /* Socket type of the transport provider    */
    )
```

DESCRIPTION This routine returns the transport-provider device name based on the address family and socket type passed to it. This routine does a linear search of the socket table and returns the device name when a match is found.

RETURNS A transport-protocol device name if successful, NULL otherwise.

ERRNO **ENXIO** on failure.

SEE ALSO **strmSockLib**

strmSockProtoAdd( )

NAME *strmSockProtoAdd*( ) – add transport-protocol entry to STREAMS sockets (STREAMS Opt.)

SYNOPSIS

```
STATUS strmSockProtoAdd
    (
    int     family,  /* address family of added protocol */
    int     type,    /* socket type of added protocol    */
    char *  devName  /* transport provider device name   */
    )
```

DESCRIPTION This routine enables transport independence for the STREAMS socket library by adding transport-provider entries to the table. It must be called for all transport providers that use the STREAMS socket interface. This routine must be executed before applications call *socket*( ). The table consists of three fields: the address family (**AF_INET**, **AF_CCITT**), the socket type (**SOCK_STREAM**, **SOCK_RAW**), and the transport-provider device name. The socket library decides on an appropriate transport provider based on the address-family and socket-type parameters passed to the *socket*( ).

WARNING: Two different transport providers having the same address family and socket type cannot be added to the table.

EXAMPLE

```
strmSockProtoAdd (AF_INET, SOCK_DGRAM, "/dev/udp")
```

RETURNS OK, or ERROR if the transport-provider entry cannot be made.

ERRNO **S_strmLib_DUPLICATE_PROVIDER**

SEE ALSO **strmSockLib**

strmSockProtoDelete( )

NAME *strmSockProtoDelete*( ) – remove a protocol entry from the table (STREAMS Opt.)

SYNOPSIS

```
STATUS strmSockProtoDelete
    (
    int  family,  /* address family */
    int  type     /* socket type    */
    )
```

DESCRIPTION This routine removes a protocol entry from the table. It does a linear search of the table and removes an entry when there is a match.

RETURNS OK if successful, ERROR otherwise.

SEE ALSO **strmSockLib**

strmStatShow( )

NAME *strmStatShow*( ) – display statistics about streams (STREAMS Opt.)

SYNOPSIS

```
void strmStatShow ( void )
```

DESCRIPTION This routine displays the following statistics about streams:

– the number of streams allocated,
– the number of streams in use,
– the total number of streams configured,
– the maximum number of streams that can be configured,
– the number of stream allocation failures.

RETURNS N / A

SEE ALSO **strmShow**

strmSyncWriteAccess( )

NAME *strmSyncWriteAccess*( ) – access a shared data structure for synchronous writing (STREAMS Opt.)

SYNOPSIS

```
void strmSyncWriteAccess
    (
    queue_t                     *pQueue, /* queue to get for write        */
    mblk_t                      *pMsg,   /* msg block to pass to callback */
    void (*pFuncToWrite)(PCELL, mblk_t *)  /* callback for write operation */
    )
```

DESCRIPTION This routine is provided to STREAMS drivers and modules to gain exclusive write access to a shared data structure.

EXAMPLE The read-side and write-side queues of an IP module operate independently; however, when an IP instance updates its routing table, it requires temporary exclusive access to the routing table shared among all IP instances.

RETURNS N / A

SEE ALSO **strmLib**

strmTimeout( )

NAME *strmTimeout*( ) – execute a routine in a specified length of time (STREAMS Opt.)

SYNOPSIS

```
int strmTimeout
    (
    void     (*pFunc)(),  /* timeout callback function           */
    caddr_t  pArg,        /* argument to callback function       */
    long     ticks        /* time when callback should be called */
    )
```

DESCRIPTION This routine schedules the function specified by *pFunc* to be called after a specified time interval. The *pArg* parameter is passed as the only argument to *pFunc*.

RETURNS The timer ID.

SEE ALSO **strmLib**, *strmUntimeout*( )

*strmUntimeout***( )**

NAME *strmUntimeout***( )** – cancel the previous *strmTimeout***( )** call (STREAMS Opt.)

SYNOPSIS

```
void strmUntimeout
    (
    int  timeoutId  /* timer ID to be cancelled */
    )
```

DESCRIPTION This routine cancels the pending *strmTimeout***( )** request specified by *timeoutId*.

RETURNS N / A

SEE ALSO **strmLib**, *strmTimeout***( )**

*strmUnWeld***( )**

NAME *strmUnWeld***( )** – set the **q_next** pointers of streams queues to NULL (STREAMS Opt.)

SYNOPSIS

```
STATUS strmUnWeld
    (
    queue_t *  pQdst1,      /* pQdts1->q_next is set to NULL        */
    queue_t *  pQdst2,      /* pQdts2->q_next is set to NULL        */
    void       (*pFunc)(),  /* callback to notify unweld completion */
    u32        arg0,        /* argument 1 to callback               */
    u32        arg1         /* argument 2 to callback               */
    )
```

DESCRIPTION This routine removes queue links performed by *strmWeld***( )** operations. It assigns NULL to the **q_next** fields of the queue parameters *pQdst1* and *pQdst2*. The routine *strmUnWeld***( )** does not set the **q_next** pointers to NULL synchronously at its invocation. An optional callback function can be specified as the argument *pFunc* to *strmUnWeld***( )**. The callback function is invoked after the **q_next** pointers have been set to NULL.

RETURNS OK if successful, otherwise ERROR.

SEE ALSO **strmLib**

strmWakeup( )

NAME *strmWakeup*( ) – resume suspended task execution (STREAMS Opt.)

SYNOPSIS

```
void strmWakeup
    (
    ulong_t  event  /* tasks to be woken up for this event */
    )
```

DESCRIPTION This routine awakens tasks sleeping on an event and makes them eligible for scheduling.

SEE ALSO **strmLib**, *strmSleep*( )

strmWeld( )

NAME *strmWeld*( ) – connect the **q_next** pointers of arbitrary streams (STREAMS Opt.)

SYNOPSIS

```
STATUS strmWeld
    (
    queue_t *  pQdst1,      /* pQdts1->q_next is set to pQsrc1    */
    queue_t *  pQsrc1,      /* new value for pQdst1->q_next       */
    queue_t *  pQdst2,      /* pQdts2->q_next is set to pQsrc2    */
    queue_t *  pQsrc2,      /* new value for pQdst2->q_next       */
    void       (*pFunc)(),  /* callback to notify weld completion */
    u32        arg0,        /* argument 1 to callback             */
    u32        arg1         /* argument 2 to callback             */
    )
```

DESCRIPTION This routine connects **q_next** pointers of arbitrary streams to facilitate construction of STREAMS configurations not supported by normal STREAMS services. The routine *strmWeld*( ) does not connect the **q_next** pointers synchronously at its invocation. An optional callback function can be specified as the argument *pFunc* to *strmWeld*( ). The callback function is invoked after the **q_next** pointers have been connected.

EXAMPLES A loopback configuration can be created by setting the driver's write-side **q_next** pointer to its read-side queue. The *strmWeld*( ) call would set *pQdst1* to point to the driver write-side queue, set *pQsrc2* to its read-side queue, and set the rest of the parameters to NULL.

RETURNS OK if successful, otherwise ERROR.

SEE ALSO **strmLib**

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**

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*( )

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.

In other than the "C" locale, additional implementation-defined subject sequence forms may be accepted. VxWorks supports only the "C" locale.

If the 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*, if *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**

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 *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 premitted. 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 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 *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 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 premitted. 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 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.

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**

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 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**.

To search the global VxWorks symbol table, specify **sysSymTbl** as *symTblId*.

RETURNS OK, or ERROR if *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 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**.

To search the global VxWorks symbol table, specify **sysSymTbl** as *symTblId*.

RETURNS OK, or ERROR if *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 **INCLUDE_SYM_TBL** is defined in **configAll.h**, it 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 **INCLUDE_SYM_TBL_SYNC** is defined in **configAll.h** or in **config.h**.

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**

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 man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man 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 manual entries for **sysLib** and *sysAuxClkConnect*( )

sysAuxClkDisable( )

NAME *sysAuxClkDisable*( ) – turn off auxiliary clock interrupts

SYNOPSIS
```
void sysAuxClkDisable (void)
```

DESCRIPTION This routine disables auxiliary clock interrupts.

NOTE: This is a generic man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man pages for your BSP.

RETURNS N / A

SEE ALSO **sysLib**, *sysAuxClkEnable*( ), and BSP-specific manual entries for **sysLib** and *sysAuxClkDisable*( )

sysAuxClkEnable( )

NAME *sysAuxClkEnable*( ) – turn on auxiliary clock interrupts

SYNOPSIS
```
void sysAuxClkEnable (void)
```

DESCRIPTION This routine enables auxiliary clock interrupts.

NOTE: This is a generic man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man pages for your BSP.

RETURNS N / A

SEE ALSO **sysLib**, *sysAuxClkConnect*( ), *sysAuxClkDisable*( ), *sysAuxClkRateSet*( ), and BSP-specific manual entries for **sysLib** and *sysAuxClkEnable*( )

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 man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man pages for your BSP.

RETURNS The number of ticks per second of the auxiliary clock.

SEE ALSO **sysLib**, *sysAuxClkEnable*( ), *sysAuxClkRateSet*( ), and BSP-specific manual entries for **sysLib** and *sysAuxClkRateGet*( )

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 man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man 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 manual entries for **sysLib** and *sysAuxClkRateSet*( )

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 man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man pages for your BSP.

RETURNS A pointer to the BSP version/revision string.

SEE ALSO **sysLib**, and BSP-specific manual entries for **sysLib** and *sysBspRev*( )

sysBusIntAck( )

NAME *sysBusIntAck*( ) – acknowledge a bus interrupt

SYNOPSIS
```
int sysBusIntAck
    (
    int  intLevel  /* interrupt level to acknowledge */
    )
```

DESCRIPTION This routine acknowledges a specified VMEbus interrupt level.

NOTE: This is a generic man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man pages for your BSP.

RETURNS NULL.

SEE ALSO **sysLib**, *sysBusIntGen*( ), and BSP-specific manual entries for **sysLib** and *sysBusIntAck*( )

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 man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man 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 manual entries for **sysLib** and *sysBusIntGen*( )

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 man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man 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 manual entries for **sysLib** and *sysBusTas*( )

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 man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man pages for your BSP.

RETURNS OK, or ERROR if the address space is unknown or the mapping is not possible.

SEE ALSO **sysLib**, *sysLocalToBusAdrs*( ), and BSP-specific manual entries for **sysLib** and *sysBusToLocalAdrs*( )

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 man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man pages 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 manual entries for **sysLib** and *sysClkConnect*( )

sysClkDisable( )

NAME *sysClkDisable*( ) – turn off system clock interrupts

SYNOPSIS

```
void sysClkDisable (void)
```

DESCRIPTION This routine disables system clock interrupts.

NOTE: This is a generic man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man pages for your BSP.

RETURNS N / A

SEE ALSO **sysLib**, *sysClkEnable*( ), and BSP-specific manual entries for **sysLib** and *sysClkDisable*( )

sysClkEnable( )

NAME *sysClkEnable*( ) – turn on system clock interrupts

SYNOPSIS
```
void sysClkEnable (void)
```

DESCRIPTION This routine enables system clock interrupts.

NOTE: This is a generic man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man pages for your BSP.

RETURNS N / A

SEE ALSO **sysLib**, *sysClkConnect*( ), *sysClkDisable*( ), *sysClkRateSet*( ), and BSP-specific manual entries for **sysLib** and *sysClkEnable*( )

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 man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man pages for your BSP.

RETURNS The number of ticks per second of the system clock.

SEE ALSO **sysLib**, *sysClkEnable*( ), *sysClkRateSet*( ), and BSP-specific manual entries for **sysLib** and *sysClkRateGet*( )

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**.

NOTE: This is a generic man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man 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*( ), and BSP-specific manual entries for **sysLib** and *sysClkRateSet*( )

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 man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man 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 manual entries for **sysLib** and *sysHwInit*( )

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 man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man pages for your BSP.

RETURNS OK, or ERROR if *intLevel* is out of range.

SEE ALSO **sysLib**, *sysIntEnable*( ), and BSP-specific manual entries for **sysLib** and *sysIntDisable*( )

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 man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man pages for your BSP.

RETURNS OK, or ERROR if *intLevel* is out of range.

SEE ALSO **sysLib**, *sysIntDisable*( ), and BSP-specific manual entries for **sysLib** and *sysIntEnable*( )

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 man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man pages for your BSP.

RETURNS OK, or ERROR if the address space is unknown or not mapped.

SEE ALSO **sysLib**, *sysBusToLocalAdrs*( ), and BSP-specific manual entries for **sysLib** and *sysLocalToBusAdrs*( )

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 man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man 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 manual entries for **sysLib** and *sysMailboxConnect*( )

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 man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man pages for your BSP.

RETURNS OK, always.

SEE ALSO **sysLib**, *sysMailboxConnect*( ), and BSP-specific manual entries for **sysLib** and *sysMailboxEnable*( )

sysMemTop( )

NAME *sysMemTop*( ) – get the address of the top of logical memory

SYNOPSIS
```
char *sysMemTop (void)
```

DESCRIPTION This routine returns the address of the top of memory.

NOTE: This is a generic man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man pages for your BSP.

RETURNS The address of the top of memory.

SEE ALSO **sysLib**, and BSP-specific manual entries for **sysLib** and *sysMemTop*( )

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 man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man pages for your BSP.

RETURNS A pointer to a string containing the board name.

SEE ALSO **sysLib**, and BSP-specific manual entries for **sysLib** and *sysModel*( )

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 man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man 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 manual entries for **sysLib** and *sysNvRamGet*( )

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 man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man 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 manual entries for **sysLib** and *sysNvRamSet*( )

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** in **config.h**. 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 man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man pages for your BSP.

RETURNS The address of the top of physical memory.

SEE ALSO **sysLib**, *sysMemTop***( )**, and BSP-specific manual entries for **sysLib** and *sysPhysMemTop***( )**

*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 man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man pages for your BSP.

RETURNS The processor number for the CPU board.

SEE ALSO **sysLib**, *sysProcNumSet***( )**, and BSP-specific manual entries for **sysLib** and *sysProcNumGet***( )**

*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 man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man pages for your BSP.

RETURNS N / A

SEE ALSO **sysLib**, *sysProcNumGet***( )**, and BSP-specific entries for **sysLib** and *sysProcNumSet***( )**

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 man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man pages for your BSP.

RETURNS

N / A

SEE ALSO

sysLib, and BSP-specific manual entries for **sysLib** and *sysScsiBusReset*( )

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** in **config.h**, 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 man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man pages for your BSP.

RETURNS

OK or ERROR.

SEE ALSO

sysLib, and BSP-specific manual entries for **sysLib** and *sysScsiConfig***( )**

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 in **config.h**, the DMA is also initialized.

NOTE: This is a generic man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man 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 manual entries for **sysLib** and *sysScsiInit*( )

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 man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man 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 manual entries for **sysLib** and *sysSerialChanGet*( )

sysSerialHwInit( )

NAME *sysSerialHwInit*( ) – initialize the BSP serial devices to a quiesent state

SYNOPSIS
```
void sysSerialHwInit (void)
```

DESCRIPTION This routine initializes the BSP serial device descriptors and puts the devices in a quiesent state. It is called from *sysHwInit*( ) with interrupts locked.

NOTE: This is a generic man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man pages for your BSP.

RETURNS N / A

SEE ALSO **sysLib**, and BSP-specific manual entries for **sysLib** and *sysSerialHwInit*( )

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 man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man pages for your BSP.

RETURNS N / A

SEE ALSO **sysLib**, and BSP-specific manual entries for **sysLib** and *sysSerialHwInit2*( )

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 man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man pages for your BSP.

RETURNS N / A

SEE ALSO **sysLib**, and BSP-specific manual entries for **sysLib** and *sysSerialReset*( )

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 man page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the man pages for your BSP.

RETURNS Does not return.

SEE ALSO **sysLib**, and BSP-specific manual entries for **sysLib** and *sysToMonitor*( )

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.

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.

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.

SEE ALSO **tapeFsLib**, *tapeFsReadyChange***( )**

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 */
    )
```

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.

ERRNOS

S_intLib_NOT_ISR_CALLABLE
This routine cannot be called from an ISR.

EINTR
An interrupted system call occurred.

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 TCB memory resources. On deletion, all routines specified by *taskDeleteHookAdd***( )** are 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.

ERRNOS

S_intLib_NOT_ISR_CALLABLE
This routine cannot be called from an ISR.

S_objLib_OBJ_DELETED
This task has already been deleted.

S_objLib_OBJ_UNAVAILABLE
You have specified NOWAIT and the task cannot be deleted at this time.

S_objLib_OBJ_ID_ERROR
This is an incorrect task ID.

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.

CAVEATS

This routine is intended as a debugging aid, and is generally inappropriate for applications. Disregarding a task's deletion protection could leave the the system in an unstable state or lead to system deadlock.

The system does not protect against simultaneous *taskDeleteForce*( ) calls. Such a situation could leave the system in an unstable state.

RETURNS

OK, or ERROR if the task cannot be deleted.

ERRNOS

Possible errnos generated by this routine include:

S_intLib_NOT_ISR_CALLABLE
This routine cannot be called from an ISR.

S_objLib_OBJ_DELETED
This task has already been deleted.

S_objLib_OBJ_UNAVAILABLE
You have specified NOWAIT and the task cannot be deleted at this time.

S_objLib_OBJ_ID_ERROR
This is an incorrect task ID.

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*( )

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 **INCLUDE_TASK_HOOKS** is defined in **configAll.h**.

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 **INCLUDE_SHOW_ROUTINES** is defined in **configAll.h**.

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 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: Shell*, **windsh**, *Tornado User's Guide: Shell*

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.

RETURNS OK, or ERROR if the task ID is invalid.

ERRNOS Possible errnos generated by this routine include:

S_objLib_OBJ_ID_ERROR
This is an incorrect task ID.

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_TCB**s 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.

ERRNOS

Possible errnos generated by this routine include:

S_intLib_NOT_ISR_CALLABLE
This routine cannot be called from an ISR.

S_objLib_OBJ_ID_ERROR
This is an incorrect task ID.

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**

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.

ERRNOS Possible errnos generated by this routine include:

S_objLib_OBJ_ID_ERROR
This is an incorrect task ID.

S_intLib_NOT_ISR_CALLABLE
This routine cannot be called from an ISR.

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.

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.

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.

ERRNOS **S_objLib_OBJ_ID_ERROR**
This is an incorrect task ID.

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.

ERRNOS

S_taskLib_ILLEGAL_PRIORITY
You have requested a priority outside the specified range.

S_objLib_OBJ_ID_ERROR
This is an incorrect task ID.

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
```

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 reinitialized with the same ID, priority, options, original entry point, stack size, and parameters it had when it was terminated. Self-restarting of a calling task is performed by 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.

ERRNOS

Possible errnos generated by this routine include:

S_intLib_NOT_ISR_CALLABLE
This routine cannot be called from an ISR.

S_objLib_OBJ_DELETED
This task has already been deleted.

S_objLib_OBJ_UNAVAILABLE
You have specified NOWAIT and the task cannot be restarted at this time.

S_objLib_OBJ_ID_ERROR
This is an incorrect task ID.

S_smObjLib_NOT_INITIALIZED
Not enough memory in the specified partition to spawn this task.

S_memLib_NOT_ENOUGH_MEMORY
There is not enough memory to spawn this task.

S_memLib_BLOCK_ERROR
Cannot get exclusive access to the memory partition.

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.

ERRNOS Possible errnos generated by this routine include:

S_objLib_OBJ_ID_ERROR
This is an incorrect task ID.

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
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
```

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. These routines are included automatically when **INCLUDE_SHOW_ROUTINES** is defined in **configAll.h**.

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 specified priority and options; it returns a system-assigned ID. See *taskInit*( ) and *taskActivate*( ) for the routine's building blocks.

A task may be assigned a name as a debugging aid. This name appears 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 NULL, an ASCII name is assigned to the task of the form "t*n*" where *n* is an integer that 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.

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.

ERRNOS

S_intLib_NOT_ISR_CALLABLE
This routine cannot be called from an ISR.

S_objLib_OBJ_ID_ERROR
This is an incorrect task ID.

S_smObjLib_NOT_INITIALIZED
Not enough memory in the specified partition to spawn this task.

S_memLib_NOT_ENOUGH_MEMORY
There is not enough memory to spawn this task.

S_memLib_BLOCK_ERROR
Cannot get exclusive access to the memory partition.

SEE ALSO

taskLib, *taskInit***( )**, *taskActivate***( )**, *sp***( )**, *VxWorks Programmer's Guide: Basic OS*

taskSRSet( )

NAME *taskSRSet*( ) – set the task status register (MC680x0, MIPS, i386/i486)

SYNOPSIS
```
STATUS taskSRSet
    (
    int     tid,  /* task ID */
    UINT16  sr    /* new SR  */
    )
```

DESCRIPTION This routine sets the status register of a non-running task (i.e., the TCB must not be that of the calling task). Debugging facilities call this routine to set the trace bit in the status register of a task being single-stepped.

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).
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.

-> printf ("shell status = <%s>\n", xx)
shell status = <READY>
```

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.

ERRNOS

S_objLib_OBJ_ID_ERROR
This is an incorrect task ID.

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.

ERRNOS
S_objLib_OBJ_ID_ERROR
This is an incorrect task ID.

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.

ERRNOS
S_intLib_NOT_ISR_CALLABLE
This routine cannot be called from an ISR.

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*.

CAVEATS 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.

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*( )

tcicInit( )

NAME *tcicInit*( ) – initialize the TCIC chip

SYNOPSIS

```
STATUS tcicInit
    (
    int      ioBase,    /* IO base address  */
    int      intVec,    /* interrupt vector */
    int      intLevel,  /* interrupt level  */
    FUNCPTR  showRtn    /* show routine     */
    )
```

DESCRIPTION This routine initializes the TCIC chip.

RETURNS OK, or ERROR if the TCIC chip cannot be found.

SEE ALSO **tcic**

tcicShow( )

NAME *tcicShow*( ) – show all configurations of the TCIC chip

SYNOPSIS

```
void tcicShow
    (
    int  sock  /* socket no. */
    )
```

DESCRIPTION This routine shows all configurations of the TCIC chip.

RETURNS N / A

SEE ALSO **tcicShow**

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 **netShow**

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 **netShow**

tcw( )

NAME *tcw*( ) – return the contents of the **tcw** register (i960)

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

DESCRIPTION This command extracts the contents of the **tcw** register from the TCB of a specified task. If *taskId* is omitted or 0, the current default task is assumed.

RETURNS The contents of the **tcw** register.

SEE ALSO **dbgArchLib**, *VxWorks Programmer's Guide: Target Shell*

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*

telnetd( )

NAME *telnetd*( ) – VxWorks telnet daemon

SYNOPSIS
```
void telnetd (void)
```

DESCRIPTION This routine enables remote users to log in to VxWorks over the network via the telnet protocol. It is spawned by *telnetInit*( ), which should be called at boot time.

Remote telnet requests will cause **stdin**, **stdout**, and **stderr** to be stolen away from the console. When the remote user disconnects, **stdin**, **stdout**, and **stderr** are restored, and the shell is restarted.

The telnet daemon requires the existence of a pseudo-terminal device, which is created by *telnetInit*( ) before *telnetd*( ) is spawned. The *telnetd*( ) routine creates two additional processes, **tTelnetInTask** and **tTelnetOutTask**, whenever a remote user is logged in. These processes exit when the remote connection is terminated.

RETURNS N / A

SEE ALSO **telnetLib**

telnetInit( )

NAME *telnetInit*( ) – initialize the telnet daemon

SYNOPSIS
```
void telnetInit (void)
```

DESCRIPTION This routine initializes the telnet facility, which supports remote login to the VxWorks shell via the telnet protocol. It creates a pty device and spawns the telnet daemon. It is called automatically when **INCLUDE_TELNET** is defined in **configAll.h**.

RETURNS N / A

SEE ALSO **telnetLib**

testproc_error( )

NAME *testproc_error*( ) – indicate that a **testproc** operation encountered an error

SYNOPSIS
```
void testproc_error
    (
    SNMP_PKT_T *  pPkt,      /* internal representation of the snmp packet */
    VB_T *        pVarBind,  /* var bind being processed                   */
    INT_32_T      error      /* error code                                 */
    )
```

DESCRIPTION This routine indicates that **testproc** encountered an error which will prevent a **set** operation from being successful.

RETURNS N / A

SEE ALSO **snmpProcLib**

testproc_good( )

NAME *testproc_good*( ) – indicate successful completion of a **testproc** procedure

SYNOPSIS

```
void testproc_good
    (
    SNMP_PKT_T *  pPkt,     /* internal representation of the snmp packet */
    VB_T *        pVarBind  /* var bind being processed                   */
    )
```

DESCRIPTION This routine indicates that **testproc** has successfully completed the checks to be performed prior to carrying out a **set** operation for a given variable binding.

RETURNS N / A

SEE ALSO **snmpProcLib**

testproc_started( )

NAME *testproc_started*( ) – indicate that a **testproc** operation has begun

SYNOPSIS

```
void testproc_started
    (
    SNMP_PKT_T *  pPkt,     /* internal representation of the snmp packet */
    VB_T *        pVarBind  /* var bind being processed                   */
    )
```

DESCRIPTION This routine indicates that **testproc** for the specified variable binding has been started by the user.

pPkt is the internal representation of the SNMP packet. *pvarBind* is a pointer to the variable-binding being processed.

RETURNS N / A

SEE ALSO **snmpProcLib**

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 filename. *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**

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 without creating a new task. It will simply report whether a new TFTP task was successfully spawned. The argument *stackSize* can be specified to change the default stack size for the TFTP server task. The default size is set in the global variable **tftpdTaskStackSize**.

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***( )**.

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**

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 gets 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:

```
int  dataFd;
int  errorFd;
int  num;
char buf [512];
if (tftpXfer ("congo", 0, "/usr/fred", "get", "ascii", &dataFd,
              &errorFd) == ERROR)
    return (ERROR);
while ((num = read (dataFd, buf, sizeof (buf))) > 0)
    {
    ....
    }
close (dataFd);
if (read (errorFd, buf, sizeof (buf)) > 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
```

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***( )**

*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 *timer* points to.

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    */
    )
```

NOTE: Non-POSIX.

RETURNS

0 (OK), or -1 (ERROR) if the timer is invalid or cannot bind the signal handler.

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 there are already too many timers 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 task ID 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,  /* 1st of up to 8 args to call <func> 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.
```

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 **INCLUDE_TIMEX** is defined in **configAll.h**, 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*

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*

tsp( )

NAME

tsp( ) – return the contents of register **sp** (i960)

SYNOPSIS

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

DESCRIPTION

This command extracts the contents of register **sp**, the stack pointer, from the TCB of a specified task. If *taskId* is omitted or 0, the current default task is assumed.

Note: The name *tsp*( ) is used because *sp*( ) (the logical name choice) conflicts with the routine *sp*( ) for spawning a task with default parameters.

RETURNS

The contents of the **sp** register.

SEE ALSO

dbgArchLib, *VxWorks Programmer's Guide: Target Shell*

tt( )

NAME

tt( ) – print a stack trace of a task

SYNOPSIS

```
STATUS tt
    (
    int  task  /* task whose stack is to be traced */
    )
```

DESCRIPTION

This routine prints a list of the nested routine calls that the specified task is in. Each routine call and its parameters are shown.

If *task* 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)
```

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.

CAVEAT

In order to do the trace, some assumptions are made. In general, the trace will work for all C language routines and for assembly language routines 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 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**

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 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**

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) causes no character to be entered in the current line, but causes 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 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**.

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** 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 are in the input buffer, **FIONREAD** returns the value 10.

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**

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**

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 **netShow**

ulattach( )

NAME *ulattach*( ) – attach a ULIP interface to a list of network interfaces (VxSim)

SYNOPSIS
```
STATUS ulattach
    (
    int  unit  /* ULIP unit number */
    )
```

DESCRIPTION This routine is called by *ulipInit***( )**. It inserts a pointer to the ULIP interface data structure into a linked list of available network interfaces.

RETURNS OK or ERROR.

SEE ALSO **if_ulip**, *VxSim User's Guide*

---

*ulipDelete***( )**

NAME *ulipDelete***( )** – delete a ULIP interface (VxSim)

SYNOPSIS

```
STATUS ulipDelete
    (
    int  unit  /* ULIP unit number */
    )
```

DESCRIPTION This routine detaches the ULIP unit and frees up resources taken by this ULIP interface.

RETURNS OK, or ERROR if the unit number is invalid or the interface is uninitialized.

SEE ALSO **if_ulip**, *VxSim User's Guide*

---

*ulipInit***( )**

NAME *ulipInit***( )** – initialize the ULIP interface (VxSim)

SYNOPSIS

```
STATUS ulipInit
    (
    int   unit,       /* ULIP unit number (0 - NULIP-1)           */
    char  *myAddr,    /* IP address of the interface              */
    char  *peerAddr,  /* IP address of the remote peer interface  */
    int   procnum     /* processor number to map to ULIP interface */
    )
```

DESCRIPTION This routine initializes the ULIP interface and sets the Internet address as a function of the processor number.

RETURNS OK, or ERROR if the device cannot be opened or there is insufficient memory.

SEE ALSO **if_ulip**, *VxSim User's Guide*

ultraattach( )

NAME *ultraattach*( ) – publish the **ultra** network interface and initialize the driver and device

SYNOPSIS

```
STATUS ultraattach
    (
    int  unit,     /* unit number                                  */
    int  ioAddr,   /* address of ultra's shared memory             */
    int  ivec,     /* interrupt vector to connect to               */
    int  ilevel,   /* interrupt level                              */
    int  memAddr,  /* address of ultra's shared memory             */
    int  memSize,  /* size of ultra's shared memory                */
    int  config    /* 0: RJ45 + AUI(Thick)  1: RJ45 + BNC(Thin) */
    )
```

DESCRIPTION This routine attaches an **ultra** Ethernet interface to the network if the device exists. It makes the interface available by filling in the network interface record. The system will initialize the interface when it is ready to accept packets.

RETURNS OK or ERROR.

SEE ALSO **if_ultra**, **ifLib**, **netShow**

ultraShow( )

NAME *ultraShow*( ) – display statistics for the **ultra** network interface

SYNOPSIS

```
void ultraShow
    (
    int   unit,  /* interface unit */
    BOOL  zap    /* zero totals    */
    )
```

DESCRIPTION This routine displays statistics about the **elc** Ethernet network interface. Prameters:

unit interface unit; should be 0.

zap if 1, all collected statistics are cleared to zero.

RETURNS N / A

SEE ALSO **if_ultra**

undoproc_error( )

NAME *undoproc_error*( ) – indicate that an **undproc** operation encountered an error

SYNOPSIS

```
void undoproc_error
    (
    SNMP_PKT_T *  pPkt,      /* internal representation of the snmp packet */
    VB_T *        pVarBind,  /* var bind being processed                   */
    INT_32_T      error      /* error value                                */
    )
```

DESCRIPTION This routine indicates that **undoproc** encountered an error for a specified variable binding.

RETURNS N / A

SEE ALSO **snmpProcLib**

undoproc_good( )

NAME *undoproc_good*( ) – indicates successful completion of an **undoproc** operation

SYNOPSIS

```
void undoproc_good
    (
    SNMP_PKT_T *  pPkt,     /* internal representation of the snmp packet */
    VB_T *        pVarBind  /* var bind being processed                   */
    )
```

DESCRIPTION This routine indicates that an **undoproc** operation completed successfully.

RETURNS N / A

SEE ALSO **snmpProcLib**

undoproc_started( )

NAME *undoproc_started*( ) – indicate that an **undoproc** operation has begun

SYNOPSIS

```
void undoproc_started
    (
    SNMP_PKT_T *  pPkt,     /* internal representation of the snmp packet */
    VB_T *        pVarBind  /* var bind being processed                   */
    )
```

DESCRIPTION This routine indicates that **undoproc** has been started by the user for the specified variable binding.

RETURNS N / A

SEE ALSO **snmpProcLib**

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 (VxSim)

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.

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 (VxSim)

SYNOPSIS

```
void unixDiskInit
    (
    char  *unixFile,  /* UNIX file name  */
    char  *volName,   /* dosFs name      */
    int   diskSize    /* number of bytes */
    )
```

DESCRIPTION This routine provides some convenience for a user wanting to create a UNIX disk-based dosFs file system under VxWorks. The user only specifes the UNIX file to use, the dosFs volume name, and the size of the volume in bytes, if the UNIX file needs to be created.

RETURNS N / A

SEE ALSO **unixDrv**

unixDrv( )

NAME *unixDrv*( ) – install UNIX disk driver (VxSim)

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.

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 must free 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.

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

SYNOPSIS

```
STATUS usrAtaConfig
    (
    int     ctrl,     /* 0: primary address, 1: secondary address */
    int     drive,    /* drive number of hard disk (0 or 1)       */
    char *  fileName  /* mount point                              */
    )
```

DESCRIPTION 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:**.

fileName the mount point, for example, **/ata0/**.

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 **src/config/usrAta.c**, *VxWorks Programmer's Guide: I/O System, Local File Systems, Intel i386/i486/Pentium*

usrAtaPartition( )

NAME *usrAtaPartition*( ) – get an offset to the first partition of the drive

SYNOPSIS

```
int usrAtaPartition
    (
    int           ctrl,   /* 0: primary address, 1: secondary address */
    int           drive,  /* drive number of hard disk (0 or 1)       */
    DOS_PART_TBL  *pPart  /* pointer to the partition table           */
    )
```

DESCRIPTION This routine gets an offset to the first partition of the drive.

For the *drive* parameter, 0 is **C:** and 1 is **D:**.

RETURNS Offset to the partition.

SEE ALSO *usrAtaConfig*( ), **src/config/usrAta.c**

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 *VxWorks Programmer’s Guide: I/O System, Local File Systems, Intel i386/i486 Appendix*

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 *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 IDs and to configure a **scsiPhysDev** structure for each device found. Of course this does not 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.

This code can be customized on a per-BSP basis using the **SYS_SCSI_CONFIG** macro. If defined, 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 is provided in **target/src/config/usrScsi.c**. The code contains sample configurations for a hard disk, a floppy disk, and a tape unit.

RETURNS OK or ERROR.

SEE ALSO *VxWorks Programmer's Guide: I/O System, Local File Systems*

usrSmObjInit( )

NAME *usrSmObjInit*( ) – initialize shared memory objects

SYNOPSIS

```
STATUS usrSmObjInit
    (
    char *  bootString  /* boot parameter string */
    )
```

DESCRIPTION This routine initializes the shared memory objects facility. It sets up the shared memory objects facility if called from processor 0. Then it initializes a shared memory descriptor and calls *smObjAttach*( ) to attach this CPU to the shared memory object facility.

When the shared memory pool resides on the local CPU dual ported memory, **SM_OBJ_MEM_ADRS** must be set to NONE in **configAll.h** and the shared memory objects pool is allocated from the VxWorks system pool.

RETURNS OK, or ERROR if unsuccessful.

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*. It is an error for *nbytes* to be odd.

NOTE: Due to speed considerations, this routine should only be used when absolutely necessary. Use *swab*( ) for aligned swaps.

RETURNS N / A

SEE ALSO **bLib**, *swab***( )**

*utime***( )**

NAME *utime***( )** – update time on a file

SYNOPSIS
```
int utime
    (
    char *            file,
    struct utimbuf *  newTimes
    )
```

RETURNS OK or ERROR.

SEE ALSO **dirLib**, *stat***( )**, *fstat***( )**, *ls***( )**

*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.

SEE ALSO **memLib**

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 *va_start*( ), or if *va_end*( ) 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**

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 1992.
Boot line:
enp(0,0)host:/usr/wpwr/target/config/mz7170/vxWorks e=90.0.0.50 h=90.0.0.4
u=target
```

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, /* # 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:

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.

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 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 physical memory already mapped into the virtual memory space.

This routine should not be called from interrupt level.

AVAILABILITY This routine is 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.

NOTE: This routine should be used for debugging purposes only.

Note that this routine cannot report non-standard architecture-dependent states.

AVAILABILITY This routine is 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 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 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 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 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 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 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.

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, /* # 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 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 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 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 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.

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 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 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 if both **INCLUDE_MMU_FULL** and **INCLUDE_SHOW_ROUTINES** are defined in **configAll.h**.

AVAILABILITY This routine is 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 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.

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 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.

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 a component of the unbundled virtual memory support option, VxVMI.

RETURNS OK, or ERROR if the text segment cannot be write-protected.

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 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**)

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, or 4                    */
    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);
    }
```

RETURNS

OK, or ERROR if the probe caused a bus error or was misaligned.

SEE ALSO

vxLib

vxMemProbeAsi( )

NAME

vxMemProbeAsi( ) – probe address in ASI space for bus error (SPARC)

SYNOPSIS

```
STATUS vxMemProbeAsi
    (
    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    */
    int     adrsAsi  /* ASI field of address to be probed */
    )
```

DESCRIPTION

This routine probes the specified address to see if it is readable or writable, as specified by *mode*. The address will be read / written as 1, 2, 4, or 8 bytes as specified by *length* (values other than 1, 2, 4, or 8 return ERROR). If the probe is a **VX_READ** (0), then the value read will be returned in the location pointed to by *pVal*. If the probe is a **VX_WRITE** (1), then the value written will be taken from the location pointed to by *pVal*. In either case, *pVal* should point to a value of the appropriate length, 1, 2, 4, or 8 bytes, as specified by *length*.

The fifth parameter *adrsAsi* is the ASI parameter used to modify the *adrs* parameter.

EXAMPLE

```
testMem (adrs)
   char *adrs;
   {
   char testW = 1;
   char testR;
   if (vxMemProbeAsi (adrs, VX_WRITE, 1, &testW) == OK)
       printf ("value %d written to adrs %x\n", testW, adrs);
   if (vxMemProbeAsi (adrs, VX_READ, 1, &testR) == OK)
       printf ("value %d read from adrs %x\n", testR, adrs);
   }
```

RETURNS

OK, or ERROR if the probe caused a bus error or was misaligned.

SEE ALSO

vxLib

vxPowerDown( )

NAME *vxPowerDown*( ) – place the processor in reduced-power mode (PowerPC)

SYNOPSIS
```
STATUS 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*( )

vxPowerModeGet( )

NAME *vxPowerModeGet*( ) – get the power management mode (PowerPC)

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)

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*( ).

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 and PPC860 only).

VX_POWER_MODE_DOZE (0x4)
Only the decrementer, data cache, and bus snooping are active while the kernel is idle (PPC603 and PPC860).

VX_POWER_MODE_NAP (0x8)
Only the decrementer is active while the kernel is idle (PPC603, PPC604 and PPC860).

VX_POWER_MODE_SLEEP (0x10)
All CPU units are inactive while the kernel is idle (PPC603 and PPC860 – not recommended for the PPC603 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 only).

VX_POWER_MODE_DOWN (0x80)
Only a hard reset causes an exit from power-down low power mode (PPC860 only – not recommended).

RETURNS OK, or ERROR if *mode* is incorrect or not supported by the processor.

SEE ALSO **vxLib**, *vxPowerModeGet*( ), *vxPowerDown*( )

vxSSDisable( )

NAME *vxSSDisable*( ) – disable the superscalar dispatch (MC68060)

SYNOPSIS
```
void vxSSDisable (void)
```

DESCRIPTION This function resets the ESS bit of the Processor Configuration Register (PCR) to disable the superscalar dispatch.

RETURNS N / A

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 instruction is executed on the specified address. The architecture test-and-set instruction is:

68K: **tas**

SPARC: **ldstub**

i960: **atmod**

This routine is equivalent to *sysBusTas***( )** in **sysLib**.

BUGS (MIPS) Only **Kseg0** and **Kseg1** addresses are accepted; other addresses always return FALSE.

RETURNS TRUE if the value had not been set (but is now), or FALSE if the value was set already.

SEE ALSO **vxLib**, *sysBusTas***( )**

---

*VXWBSem::VXWBSem***( )**

NAME *VXWBSem::VXWBSem***( )** – create and initialize a binary semaphore (WFC Opt.)

SYNOPSIS

```
VXWBSem (int opts, SEM_B_STATE iState)
```

DESCRIPTION This routine allocates and initializes a binary semaphore. The semaphore is initialized to the state *iState*: either **SEM_FULL** (1) or **SEM_EMPTY** (0).

The *opts* 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** and **SEM_Q_FIFO**, respectively.

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 *VXWSem::take***( )**, 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 is blocked, pending the availability of the semaphore. If a timeout is specified and the timeout expires, the pended task is removed from the queue of pended tasks and enters the ready state with an ERROR status. A pended task is ineligible for CPU allocation. Any number of tasks may be pended simultaneously on the same binary semaphore.

When a task gives a binary semaphore, using *VXWSem::give***( )**, 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 *VXWSem::give***( )**, the unblocked task preempts 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.

Then guard a critical section or resource by taking the semaphore with *VXWSem::take***( )**, and exit the section or release the resource by giving the semaphore with *VXWSem::give***( )**.

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 *VXWSem::take***( )**, and an interrupt service routine can unblock that task with *VXWSem::give***( )**.

A *semFlush***( )** on a binary semaphore atomically unblocks all pended tasks in the semaphore queue; that is, all tasks are 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 **VXWMSem** offer protection from unexpected task deletion.

RETURNS N / A.

SEE ALSO **vxwSemLib**

*VXWCSem::VXWCSem***( )**

NAME *VXWCSem::VXWCSem***( )** – create and initialize a counting semaphore (WFC Opt.)

SYNOPSIS

```
VXWCSem (int opts, int count)
```

DESCRIPTION This routine allocates and initializes a counting semaphore. The semaphore is initialized to the specified initial count.

The *opts* 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** and **SEM_Q_FIFO**, respectively.

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 *VXWSem::take***( )**, subsequent action depends on the state of the count:

(1) If the count is non-zero, it is decremented and the calling task continues executing.

(2) If the count is zero, the task is blocked, pending the availability of the semaphore. If a timeout is specified and the timeout expires, the pended task is removed from the queue of pended tasks and enters the ready state with an ERROR status. A pended task is ineligible for CPU allocation. Any number of tasks may be pended simultaneously on the same counting semaphore.

When a task gives a semaphore using *VXWSem::give***( )**, the next available task 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 *VXWSem::give***( )**, the unblocked task preempts the calling task.

A *VXWSem::flush***( )** on a counting semaphore atomically unblocks all pended tasks in the semaphore queue. Thus, all tasks are made ready before any task actually executes. The count of the semaphore remains unchanged.

INTERRUPT USAGE Counting semaphores may be given but not taken from interrupt level.

CAVEATS There is no mechanism to give back or reclaim semaphores automatically when tasks are suspended or deleted. Such a mechanism, although desirable, is not feasible. Without explicit knowledge of the state of the guarded resource or region, reckless automatic reclamation of a semaphore could leave the resource in a partial state. Thus, if a task ceases execution unexpectedly, as with a bus error, currently owned semaphores are not given back, effectively leaving a resource permanently unavailable. The mutual-exclusion semaphores provided by **VXWMSem** offer protection from unexpected task deletion.

RETURNS N / A

SEE ALSO **vxwSemLib**

*VXWList::add***( )**

NAME *VXWList::add***( )** – add a node to the end of list (WFC Opt.)

SYNOPSIS
```
void add (NODE *pNode)
```

DESCRIPTION This routine adds a specified node to the end of the list.

RETURNS N / A

SEE ALSO **vxwLstLib**

*VXWList::concat***( )**

NAME *VXWList::concat***( )** – concatenate two lists (WFC Opt.)

SYNOPSIS
```
void concat (VXWList &aList)
```

DESCRIPTION This routine concatenates the specified list to the end of the current list. The specified list is left empty. Either list (or both) can be empty at the beginning of the operation.

RETURNS N / A

SEE ALSO **vxwLstLib**

*VXWList::count***( )**

NAME *VXWList::count***( )** – report the number of nodes in a list (WFC Opt.)

SYNOPSIS
```
int count () const
```

DESCRIPTION This routine returns the number of nodes in a specified list.

RETURNS The number of nodes in the list.

SEE ALSO **vxwLstLib**

*VXWList::extract***( )**

NAME *VXWList::extract***( )** – extract a sublist from list (WFC Opt.)

SYNOPSIS
```
LIST extract (NODE *pStart, NODE *pEnd)
```

DESCRIPTION This routine extracts the sublist that starts with *pStart* and ends with *pEnd*. It returns the extracted list.

RETURNS The extracted sublist.

SEE ALSO **vxwLstLib**

*VXWList::find***( )**

NAME *VXWList::find***( )** – find a node in list (WFC Opt.)

SYNOPSIS
```
int find (NODE *pNode) const
```

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 **vxwLstLib**

*VXWList::first***( )**

NAME *VXWList::first***( )** – find first node in list (WFC Opt.)

SYNOPSIS
```
NODE * first () const
```

DESCRIPTION This routine finds the first node in its list.

RETURNS A pointer to the first node in the list, or NULL if the list is empty.

SEE ALSO **vxwLstLib**

VXWList::get( )

NAME *VXWList::get*( ) – delete and return the first node from list (WFC Opt.)

SYNOPSIS
```
NODE * get ()
```

DESCRIPTION This routine gets the first node from its 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 **vxwLstLib**

VXWList::insert( )

NAME *VXWList::insert*( ) – insert a node in list after a specified node (WFC Opt.)

SYNOPSIS
```
void insert (NODE *pPrev, NODE *pNode)
```

DESCRIPTION This routine inserts a specified node into the 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 **vxwLstLib**

VXWList::last( )

NAME *VXWList::last*( ) – find the last node in list (WFC Opt.)

SYNOPSIS
```
NODE * last () const
```

DESCRIPTION This routine finds the last node in its list.

RETURNS A pointer to the last node in the list, or NULL if the list is empty.

SEE ALSO **vxwLstLib**

*VXWList::next***( )**

NAME *VXWList::next***( )** – find the next node in list (WFC Opt.)

SYNOPSIS
```
NODE * next (NODE *pNode) const
```

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 **vxwLstLib**

*VXWList::nStep***( )**

NAME *VXWList::nStep***( )** – find a list node *nStep* steps away from a specified node (WFC Opt.)

SYNOPSIS
```
NODE * nStep (NODE *pNode, int nStep) const
```

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 **vxwLstLib**

*VXWList::nth***( )**

NAME *VXWList::nth***( )** – find the Nth node in a list (WFC Opt.)

SYNOPSIS
```
NODE * nth (int nodeNum) const
```

DESCRIPTION This routine returns a pointer to the node specified by *nodeNum* where the first node in the list is numbered 1. 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 **vxwLstLib**

*VXWList::previous***( )**

NAME *VXWList::previous***( )** – find the previous node in list (WFC Opt.)

SYNOPSIS
```
NODE * previous (NODE *pNode) const
```

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 **vxwLstLib**

*VXWList::remove***( )**

NAME *VXWList::remove***( )** – delete a specified node from list (WFC Opt.)

SYNOPSIS
```
void remove (NODE *pNode)
```

DESCRIPTION This routine deletes a specified node from its list.

RETURNS N / A

SEE ALSO **vxwLstLib**

*VXWList::VXWList***( )**

NAME *VXWList::VXWList***( )** – initialize a list (WFC Opt.)

SYNOPSIS
```
VXWList ()
```

DESCRIPTION This constructor initializes a list as an empty list.

RETURNS N / A

SEE ALSO **vxwLstLib**

VXWList::VXWList( )

NAME *VXWList::VXWList*( ) – initialize a list as a copy of another (WFC Opt.)

SYNOPSIS
```
VXWList (const VXWList &);
```

DESCRIPTION This constructor builds a new list as a copy of an existing list.

RETURNS N / A

SEE ALSO **vxwLstLib**

VXWList::~VXWList( )

NAME *VXWList::~VXWList*( ) – free up a list (WFC Opt.)

SYNOPSIS
```
~VXWList ()
```

DESCRIPTION This destructor frees up memory used for nodes.

RETURNS N / A

SEE ALSO **vxwLstLib**

VXWMemPart::addToPool( )

NAME *VXWMemPart::addToPool*( ) – add memory to a memory partition (WFC Opt.)

SYNOPSIS
```
STATUS addToPool (char *pool, unsigned poolSize)
```

DESCRIPTION This routine adds memory to its memory partition. The new memory added need not be contiguous with memory previously assigned to the partition.

RETURNS OK or ERROR.

SEE ALSO **vxwMemPartLib**

*VXWMemPart::alignedAlloc***( )**

NAME *VXWMemPart::alignedAlloc***( )** – allocate aligned memory from partition (WFC Opt.)

SYNOPSIS
```
void * alignedAlloc (unsigned nBytes, unsigned alignment)
```

DESCRIPTION This routine allocates a buffer of size *nBytes* from its partition. Additionally, it ensures that the allocated buffer begins on a memory address evenly divisible by *alignment*. The *alignment* parameter must be a power of 2.

RETURNS A pointer to the newly allocated block, or NULL if the buffer cannot be allocated.

SEE ALSO **vxwMemPartLib**

*VXWMemPart::alloc***( )**

NAME *VXWMemPart::alloc***( )** – allocate a block of memory from partition (WFC Opt.)

SYNOPSIS
```
void * alloc (unsigned nBytes)
```

DESCRIPTION This routine allocates a block of memory from its partition. The size of the block allocated is equal to or greater than *nBytes*.

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

SEE ALSO **vxwMemPartLib**, *VXWMemPart::free***( )**

*VXWMemPart::findMax***( )**

NAME *VXWMemPart::findMax***( )** – find the size of the largest available free block (WFC Opt.)

SYNOPSIS
```
int findMax () const
```

DESCRIPTION This call finds for the largest block in the memory partition free list and returns its size.

RETURNS The size, in bytes, of the largest available block.

SEE ALSO **vxwMemPartLib**

*VXWMemPart::free***( )**

NAME *VXWMemPart::free***( )** – free a block of memory in partition (WFC Opt.)

SYNOPSIS
```
STATUS free (char *pBlock)
```

DESCRIPTION This routine returns to the partition's free memory list a block of memory previously allocated with *VXWMemPart::alloc***( )**.

RETURNS OK, or ERROR if the block is invalid.

SEE ALSO **vxwMemPartLib**, *VXWMemPart::alloc***( )**

*VXWMemPart::info***( )**

NAME *VXWMemPart::info***( )** – get partition information (WFC Opt.)

SYNOPSIS
```
STATUS info (MEM_PART_STATS *pPartStats) const
```

DESCRIPTION This routine takes 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 **vxwMemPartLib**, *VXWMemPart::show***( )**

*VXWMemPart::options***( )**

NAME *VXWMemPart::options***( )** – set the debug options for memory partition (WFC Opt.)

SYNOPSIS
```
STATUS options (unsigned options)
```

DESCRIPTION This routine sets the debug options for its 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.

SEE ALSO **vxwMemPartLib**

*VXWMemPart::realloc***( )**

NAME *VXWMemPart::realloc***( )** – reallocate a block of memory in partition (WFC Opt.)

SYNOPSIS
```
void * realloc (char *pBlock, int nBytes)
```

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 *VXWMemPart::alloc***( )**.

RETURNS A pointer to the new block of memory, or NULL if the call fails.

SEE ALSO **vxwMemPartLib**

*VXWMemPart::show***( )**

NAME *VXWMemPart::show***( )** – show partition blocks and statistics (WFC Opt.)

SYNOPSIS

```
STATUS show (int type = 0) const
```

DESCRIPTION This routine displays statistics about the available and allocated memory in its 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.

SEE ALSO **vxwMemPartLib**

*VXWMemPart::VXWMemPart***( )**

NAME *VXWMemPart::VXWMemPart***( )** – create a memory partition (WFC Opt.)

SYNOPSIS

```
VXWMemPart (char *pool, unsigned poolSize)
```

DESCRIPTION This constructor creates a new memory partition containing a specified memory pool. 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 N / A.

SEE ALSO **vxwMemPartLib**

VXWModule::flags( )

NAME *VXWModule::flags*( ) – get the flags associated with this module (WFC Opt.)

SYNOPSIS
```
int flags () const
```

DESCRIPTION This routine returns the flags associated with its module.

RETURNS The option flags.

SEE ALSO **vxwLoadLib**

VXWModule::info( )

NAME *VXWModule::info*( ) – get information about object module (WFC Opt.)

SYNOPSIS
```
STATUS info (MODULE_INFO * pModuleInfo) const
```

DESCRIPTION This routine fills in a **MODULE_INFO** structure with information about the object module.

RETURNS OK or ERROR.

SEE ALSO **vxwLoadLib**

VXWModule::name( )

NAME *VXWModule::name*( ) – get the name associated with module (WFC Opt.)

SYNOPSIS
```
char * name () const
```

DESCRIPTION This routine returns a pointer to the name associated with its module.

RETURNS A pointer to the module name.

SEE ALSO **vxwLoadLib**

*VXWModule::segFirst***( )**

NAME *VXWModule::segFirst***( )** – find the first segment in module (WFC Opt.)

SYNOPSIS
```
SEGMENT_ID segFirst () const
```

DESCRIPTION This routine returns information about the first segment of a module descriptor.

RETURNS A pointer to the segment ID.

SEE ALSO **vxwLoadLib**, *VXWModule::segGet***( )**

*VXWModule::segGet***( )**

NAME *VXWModule::segGet***( )** – get (delete and return) the first segment from module (WFC Opt.)

SYNOPSIS
```
SEGMENT_ID segGet ()
```

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 **vxwLoadLib**, *VXWModule::segFirst***( )**

*VXWModule::segNext***( )**

NAME *VXWModule::segNext***( )** – find the next segment in module (WFC Opt.)

SYNOPSIS
```
SEGMENT_ID segNext (SEGMENT_ID segmentId) const
```

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 **vxwLoadLib**

VXWModule::VXWModule( )

NAME *VXWModule::VXWModule*( ) – build module object from module ID (WFC Opt.)

SYNOPSIS
```
VXWModule (MODULE_ID aModuleId)
```

DESCRIPTION Use this constructor to manipulate a module that was not loaded using C++ interfaces. The argument *id* is the module identifier returned and used by the C interface to the VxWorks target-resident load facility.

RETURNS N / A.

SEE ALSO **vxwLoadLib**, **loadLib**

VXWModule::VXWModule( )

NAME *VXWModule::VXWModule*( ) – load an object module at specified memory addresses (WFC Opt.)

SYNOPSIS
```
VXWModule (int fd, int symFlag, char **ppText,
char **ppData = 0, char **ppBss =  0)
```

DESCRIPTION This constructor 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 caller using *VXWMemPart::alloc*( ), 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 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. In this case, NULL is returned after the module is loaded.

ADDING SYMBOLS TO THE SYMBOL TABLE

The symbols defined in the module to be loaded may be optionally added to the target-resident 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***( )**.

In addition, 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.

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 is 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, this constructor 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 internally use another storage class known as *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 on the host resolves these and places them in BSS as a final step in creating a fully linked object module. However, the VxWorks target-resident dynamic loader is most often used to load partially linked object modules. When the VxWorks loader encounters a variable labeled as common, memory for the variable is allocated, and the variable is entered in the system symbol table (if specified) at that address. Note that most static loaders have an option that forces resolution of the common storage while leaving the module relocatable.

RETURNS

N / A.

SEE ALSO

vxwLoadLib, *Tornado User's Guide: Cross-Development*

*VXWModule::VXWModule***( )**

NAME

*VXWModule::VXWModule***( )** – load an object module into memory (WFC Opt.)

SYNOPSIS

```
VXWModule (int fd, int symFlag)
```

DESCRIPTION

This constructor loads an object module from the file descriptor *fd*, and places the code, data, and BSS into memory allocated from the system memory pool.

RETURNS

N / A.

SEE ALSO

vxwLoadLib

*VXWModule::VXWModule***( )**

NAME

*VXWModule::VXWModule***( )** – create and initialize an object module (WFC Opt.)

SYNOPSIS

```
VXWModule (char *name, int format, int flags)
```

DESCRIPTION

This constructor creates an object module descriptor. It is usually called from another constructor.

The arguments specify the name of the object module file, the object module format, and a collection of options *flags*.

Space for the new module is dynamically allocated.

RETURNS N / A.

SEE ALSO **vxwLoadLib**

VXWModule::~VXWModule( )

NAME *VXWModule::~VXWModule*( ) – unload an object module (WFC Opt.)

SYNOPSIS
```
~VXWModule ()
```

DESCRIPTION This destructor unloads the object module from the target system. For a.out and ECOFF format modules, unloading does the following:

(1) It frees the space allocated for text, data, and BSS segments, unless *VXWModule::VXWModule*( ) was called with specific addresses, in which case the application is responsible for freeing 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.

Unloading modules with this interface has no effect on breakpoints in other modules.

RETURNS N / A.

SEE ALSO **vxwLoadLib**

VXWMSem::giveForce( )

NAME *VXWMSem::giveForce*( ) – give a mutual-exclusion semaphore without restrictions (WFC Opt.)

SYNOPSIS
```
STATUS giveForce ()
```

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 gives the semaphore to the next task in the pend queue, or makes the semaphore full if no tasks are pending. In effect, execution continues as if the task owning the semaphore had actually given the semaphore.

CAVEATS Use this routine should only as a debugging aid, when the condition of the semaphore is known.

RETURNS OK.

SEE ALSO **vxwSemLib**, *VXWSem::give***( )**

*VXWMSem::VXWMSem***( )**

NAME *VXWMSem::VXWMSem***( )** – create and initialize a mutual-exclusion semaphore (WFC Opt.)

SYNOPSIS

```
VXWMSem (int opts)
```

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
Queue pended tasks on the basis of their priority.

SEM_Q_FIFO
Queue pended tasks on a first-in-first-out basis.

SEM_DELETE_SAFE
Protect a task that owns the semaphore from unexpected deletion. This option enables an implicit *taskSafe***( )** for each *VXWSem::take***( )**, and an implicit *taskUnsafe***( )** for each *VXWSem::give***( )**.

SEM_INVERSION_SAFE
Protect the system from priority inversion. With this option, the task owning the semaphore executes at the highest priority of the tasks pended on the semaphore, if that is higher than its current priority. This option must be accompanied by the **SEM_Q_PRIORITY** queuing mode.

Mutual-exclusion semaphores offer convenient options suited for situations that require 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 as described for *VXWBSem::VXWBSem***( )**, 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 *VXWSem::flush***( )** 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;" that is, 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 increments with each *VXWSem::take***( )** and decrements with each *VXWSem::give***( )**.

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 executes at the priority of the highest

priority task blocked on that resource. When execution is complete, the task gives up the resource and 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 the resources it owns. The task returns to its normal priority only after relinquishing all of its mutual-exclusion semaphores that have the inversion-safety option enabled.

SEMAPHORE DELETION

The *VXWSem::~VXWSem***( )** destructor 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 is 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 *VXWSem::take***( )**, and a *taskUnsafe***( )** with each *VXWSem::give***( )**. 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.

RETURNS

N / A

SEE ALSO

vxwSemLib, *taskSafe***( )**, *taskUnsafe***( )**

VXWMsgQ::info( )

NAME *VXWMsgQ::info*( ) – get information about message queue (WFC Opt.)

SYNOPSIS

```
STATUS info (MSG_Q_INFO *pInfo) const
```

DESCRIPTION This routine gets information about the state and contents of its 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 the message queue is empty, there may be tasks blocked on receiving. If the 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 is returned for that array.

No more than *msgListMax* message pointers and lengths are returned, although *numMsgs* is always returned with the actual number of messages queued.

For example, if the caller supplies a *msgPtrList* and *msgLenList* with room for 10 messages and sets *msgListMax* to 10, but there are 20 messages queued, then the pointers and lengths of the first 10 messages in the queue are returned in *msgPtrList* and *msgLenList*, but *numMsgs* is returned with the value 20.

A list of the task IDs of tasks blocked on the message queue can be obtained by setting *taskIdList* to the address of an array to receive the list, and setting *taskIdListMax* to the maximum number of elements in that array. If *taskIdList* is NULL, then no task IDs are returned. No more than *taskIdListMax* task IDs are returned, although *numTasks* is always returned with the actual number of tasks blocked.

For example, if the caller supplies a *taskIdList* with room for 10 task IDs and sets *taskIdListMax* to 10, but there are 20 tasks blocked on the message queue, then the IDs of the first 10 tasks in the blocked queue are returned in *taskIdList*, but *numTasks* is returned with the value 20.

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 *VXWMsgQ::send***( )** and *VXWMsgQ::receive***( )** (or their equivalents in other language bindings) 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 is an accurate snapshot of the state of the message queue at the time of the call. This information is generally used for debugging purposes only.

WARNING: The current implementation of this routine locks out interrupts while obtaining the information. This can compromise the overall interrupt latency of the system. Generally this routine is used for debugging purposes only.

RETURNS OK or ERROR.

SEE ALSO **vxwMsgQLib**

*VXWMsgQ::numMsgs***( )**

NAME *VXWMsgQ::numMsgs***( )** – report the number of messages queued (WFC Opt.)

SYNOPSIS

```
int numMsgs () const
```

DESCRIPTION This routine returns the number of messages currently queued to the message queue.

RETURNS The number of messages queued, or ERROR.

ERRNO **S_objLib_OBJ_ID_ERROR**
msgQId is invalid.

SEE ALSO **vxwMsgQLib**

VXWMsgQ::receive( )

NAME *VXWMsgQ::receive*( ) – receive a message from message queue (WFC Opt.)

SYNOPSIS
```
int receive (char * buffer, UINT nBytes, int timeout)
```

DESCRIPTION This routine receives a message from its message queue. The received message is copied into the specified *buffer*, which is *nBytes* in length. If the message is longer than *nBytes*, 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 *VXWMsgQ::receive*( ) is called. The *timeout* parameter can also have the following special values:

NO_WAIT
return immediately, even if the message has not been sent.

WAIT_FOREVER
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_objLib_OBJ_DELETED**
the message queue was deleted while waiting to receive a message.

S_objLib_OBJ_UNAVAILABLE
timeout is set to **NO_WAIT**, and no messages are available.

S_objLib_OBJ_TIMEOUT
no messages were received in *timeout* ticks.

S_msgQLib_INVALID_MSG_LENGTH
nBytes is less than 0.

SEE ALSO **vxwMsgQLib**

*VXWMsgQ::send***( )**

NAME *VXWMsgQ::send***( )** – send a message to message queue (WFC Opt.)

SYNOPSIS

```
STATUS send (char * buffer, UINT nBytes, int timeout, int pri)
```

DESCRIPTION This routine sends the message in *buffer* of length *nBytes* to its message queue. If a task is already waiting to receive messages on the queue, the message is delivered to the first waiting task. If no task is waiting, the message is saved in the message queue.

The *timeout* parameter specifies the number of ticks to wait for free space if the message queue is full. The *timeout* parameter can also have the following special values:

NO_WAIT
return immediately, even if the message has not been sent.

WAIT_FOREVER
never time out.

The *pri* parameter specifies the priority of the message being sent. The possible values are:

MSG_PRI_NORMAL
normal priority; add the message to the tail of the list of queued messages.

MSG_PRI_URGENT
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 or ERROR.

ERRNO

S_objLib_OBJ_DELETED
the message queue was deleted while waiting to a send message.

S_objLib_OBJ_UNAVAILABLE
timeout is set to **NO_WAIT**, and the queue is full.

S_objLib_OBJ_TIMEOUT
the queue is full for *timeout* ticks.

S_msgQLib_INVALID_MSG_LENGTH
nBytes is larger than the *maxMsgLength* set for the message queue.

S_msgQLib_NON_ZERO_TIMEOUT_AT_INT_LEVEL
called from an ISR, with *timeout* not set to **NO_WAIT**.

SEE ALSO **vxwMsgQLib**

VXWMsgQ::show( )

NAME *VXWMsgQ::show*( ) – show information about a message queue (WFC Opt.)

SYNOPSIS

```
STATUS show (int level) const
```

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
```

If *level* is 1, more detailed information is displayed. If messages are queued, they are displayed as follows:

```
Messages queued:
  #    address length value
  1 0x123eb204    4   0x00000001 0x12345678
```

If tasks are blocked on the queue, they are displayed as follows:

```
Receivers blocked:
   NAME      TID    PRI DELAY
---------- -------- --- -----
tExcTask     3fd678   0   21
```

RETURNS OK or ERROR.

SEE ALSO **vxwMsgQLib**

VXWMsgQ::VXWMsgQ( )

NAME *VXWMsgQ::VXWMsgQ*( ) – create and initialize a message queue (WFC Opt.)

SYNOPSIS
```
VXWMsgQ (int maxMsgs, int maxMsgLen, int opts)
```

DESCRIPTION This constructor creates a message queue capable of holding up to *maxMsgs* messages, each up to *maxMsgLen* bytes long. The queue can be created with the following options specified as *opts*:

MSG_Q_FIFO
queue pended tasks in FIFO order.

MSG_Q_PRIORITY
queue pended tasks in priority order.

RETURNS N / A.

ERRNO

S_memLib_NOT_ENOUGH_MEMORY
unable to allocate memory for message queue and message buffers.

S_intLib_NOT_ISR_CALLABLE
called from an interrupt service routine.

SEE ALSO **vxwMsgQLib**, **vxwSmLib**

VXWMsgQ::VXWMsgQ( )

NAME *VXWMsgQ::VXWMsgQ*( ) – build message-queue object from ID (WFC Opt.)

SYNOPSIS
```
VXWMsgQ (MSG_Q_ID id)
```

DESCRIPTION Use this constructor to manipulate a message queue that was not created using C++ interfaces. The argument *id* is the message-queue identifier returned and used by the C interface to the VxWorks message queue facility.

RETURNS N / A.

SEE ALSO **vxwMsgQLib**, **msgQLib**

*VXWMsgQ::~VXWMsgQ***( )**

NAME *VXWMsgQ::~VXWMsgQ***( )** – delete message queue (WFC Opt.)

SYNOPSIS

```
virtual ~VXWMsgQ ()
```

DESCRIPTION This destructor deletes a message queue. Any task blocked on either a *VXWMsgQ::send***( )** or *VXWMsgQ::receive***( )** is unblocked and receives an error from the call with **errno** set to **S_objLib_OBJECT_DELETED**.

RETURNS N / A.

ERRNO

S_objLib_OBJ_ID_ERROR
msgQId is invalid.

S_intLib_NOT_ISR_CALLABLE
called from an interrupt service routine.

SEE ALSO **vxwMsgQLib**

*VXWRingBuf::flush***( )**

NAME *VXWRingBuf::flush***( )** – make ring buffer empty (WFC Opt.)

SYNOPSIS

```
void flush ()
```

DESCRIPTION This routine initializes the ring buffer to be empty. Any data in the buffer is lost.

RETURNS N / A

SEE ALSO **vxwRngLib**

*VXWRingBuf::freeBytes***( )**

NAME *VXWRingBuf::freeBytes***( )** – determine the number of free bytes in ring buffer (WFC Opt.)

SYNOPSIS
```
int freeBytes () const
```

DESCRIPTION This routine determines the number of bytes currently unused in the ring buffer.

RETURNS The number of unused bytes in the ring buffer.

SEE ALSO **vxwRngLib**

*VXWRingBuf::get***( )**

NAME *VXWRingBuf::get***( )** – get characters from ring buffer (WFC Opt.)

SYNOPSIS
```
int get (char * buffer, int maxbytes)
```

DESCRIPTION This routine copies bytes from the ring buffer into *buffer*. It copies as many bytes as are available in the ring, up to *maxbytes*. The bytes copied are then 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 **vxwRngLib**

*VXWRingBuf::isEmpty***( )**

NAME *VXWRingBuf::isEmpty***( )** – test whether ring buffer is empty (WFC Opt.)

SYNOPSIS
```
BOOL isEmpty () const
```

DESCRIPTION This routine reports on whether the ring buffer is empty.

RETURNS TRUE if empty, FALSE if not.

SEE ALSO **vxwRngLib**

*VXWRingBuf::isFull***( )**

NAME *VXWRingBuf::isFull***( )** – test whether ring buffer is full (no more room) (WFC Opt.)

SYNOPSIS
```
BOOL isFull () const
```

DESCRIPTION This routine reports on whether the ring buffer is completely full.

RETURNS TRUE if full, FALSE if not.

SEE ALSO **vxwRngLib**

*VXWRingBuf::moveAhead***( )**

NAME *VXWRingBuf::moveAhead***( )** – advance ring pointer by *n* bytes (WFC Opt.)

SYNOPSIS
```
void moveAhead (int n)
```

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 *VXWRingBuf::putAhead***( )**.

RETURNS N / A

SEE ALSO **vxwRngLib**

*VXWRingBuf::nBytes***( )**

NAME *VXWRingBuf::nBytes***( )** – determine the number of bytes in ring buffer (WFC Opt.)

SYNOPSIS
```
int nBytes () const
```

DESCRIPTION This routine determines the number of bytes currently in the ring buffer.

RETURNS The number of bytes filled in the ring buffer.

SEE ALSO **vxwRngLib**

VXWRingBuf::put( )

NAME *VXWRingBuf::put*( ) – put bytes into ring buffer (WFC Opt.)

SYNOPSIS
```
int put (char * buffer, int nBytes)
```

DESCRIPTION This routine puts bytes from *buffer* into the ring buffer. The specified number of bytes is 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 **vxwRngLib**

VXWRingBuf::putAhead( )

NAME *VXWRingBuf::putAhead*( ) – put a byte ahead in a ring buffer without moving ring pointers (WFC Opt.)

SYNOPSIS
```
void putAhead (char byte, int offset)
```

DESCRIPTION This routine writes a byte into the ring, but does not move the ring buffer pointers. Thus the byte is not yet be available to *VXWRingBuf::get*( ) 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 *VXWRingBuf::put*( ) would put a byte, except that the input pointer is not updated.

Bytes written ahead in the ring buffer with this routine can be made available all at once by subsequently moving the ring buffer pointers with the routine *VXWRingBuf::moveAhead*( ).

Before calling *VXWRingBuf::putAhead*( ), the caller must verify that at least *offset* + 1 bytes are available in the ring buffer.

RETURNS N / A

SEE ALSO **vxwRngLib**

VXWRingBuf::VXWRingBuf( )

NAME *VXWRingBuf::VXWRingBuf*( ) – create an empty ring buffer (WFC Opt.)

SYNOPSIS
```
VXWRingBuf (int nbytes)
```

DESCRIPTION This constructor creates a ring buffer of size *nbytes*, and initializes it. Memory for the buffer is allocated from the system memory partition.

RETURNS N / A.

SEE ALSO **vxwRngLib**

VXWRingBuf::VXWRingBuf( )

NAME *VXWRingBuf::VXWRingBuf*( ) – build ring-buffer object from existing ID (WFC Opt.)

SYNOPSIS
```
VXWRingBuf (RING_ID aRingId)
```

DESCRIPTION Use this constructor to build a ring-buffer object from an existing ring buffer. This lets you to use the C++ ring-buffer interfaces even if the ring buffer was created by a C routine.

RETURNS N / A.

SEE ALSO **vxwRngLib**, **rngLib**

VXWRingBuf::~VXWRingBuf( )

NAME *VXWRingBuf::~VXWRingBuf*( ) – delete ring buffer (WFC Opt.)

SYNOPSIS
```
~VXWRingBuf ()
```

DESCRIPTION This destructor deletes a specified ring buffer. Any data in the buffer at the time it is deleted is lost.

RETURNS N / A

SEE ALSO **vxwRngLib**

*VXWSem::flush***( )**

NAME *VXWSem::flush***( )** – unblock every task pended on a semaphore (WFC Opt.)

SYNOPSIS

```
STATUS flush ()
```

DESCRIPTION This routine atomically unblocks all tasks pended on a specified semaphore; that is, all tasks are 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 *VXWMSem::VXWMSem***( )**.

RETURNS OK, or ERROR if the operation is not supported.

SEE ALSO **vxwSemLib**, *VXWCSem::VXWCsem***( )**, *VXWBSem::VXWBsem***( )**, *VXWMSem::VXWMsem***( )**, *VxWorks Programmer's Guide: Basic OS*

*VXWSem::give***( )**

NAME *VXWSem::give***( )** – give a semaphore (WFC Opt.)

SYNOPSIS

```
STATUS 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. The behavior of *VXWSem::give***( )** is discussed fully in the constructor description for the specific semaphore type being used.

RETURNS OK.

SEE ALSO **vxwSemLib**, *VXWCSem::VXWCsem***( )**, *VXWBSem::VXWBsem***( )**, *VXWMSem::VXWMsem***( )**, *VxWorks Programmer's Guide: Basic OS*

*VXWSem::id***( )**

NAME *VXWSem::id***( )** – reveal underlying semaphore ID (WFC Opt.)

SYNOPSIS
```
SEM_ID id () const
```

DESCRIPTION This routine returns the semaphore ID corresponding to a semaphore object. The semaphore ID is used by the C interface to VxWorks semaphores.

RETURNS Semaphore ID.

SEE ALSO **vxwSemLib**, **semLib**

*VXWSem::info***( )**

NAME *VXWSem::info***( )** – get a list of task IDs that are blocked on a semaphore (WFC Opt.)

SYNOPSIS
```
STATUS info (int idList [], int maxTasks) const
```

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 *VXWSem::info***( )** returns.

RETURNS The number of blocked tasks placed in *idList*.

SEE ALSO **vxwSemLib**

*VXWSem::show***( )**

NAME *VXWSem::show***( )** – show information about a semaphore (WFC Opt.)

SYNOPSIS
```
STATUS show (int level) const
```

DESCRIPTION This routine displays (on standard output) 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}
```

If *level* is 1, more detailed information is displayed. If tasks are blocked on the queue, they are displayed in the order in which they will unblock, as follows:

```
   NAME       TID    PRI DELAY
---------- -------- --- -----
tExcTask    3fd678   0    21
tLogTask    3f8ac0   0   611
```

RETURNS OK or ERROR.

SEE ALSO **vxwSemLib**

---

*VXWSem::take***( )**

NAME *VXWSem::take***( )** – take a semaphore (WFC Opt.)

SYNOPSIS

```
STATUS take (int timeout)
```

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 *VXWSem::take***( )** is discussed fully in the constructor description for the specific semaphore type being used.

A timeout in ticks may be specified. If a task times out, *VXWSem::take***( )** returns ERROR. Timeouts of **WAIT_FOREVER** and **NO_WAIT** indicate to wait indefinitely or not to wait at all.

When *VXWSem::take***( )** returns due to timeout, it sets the errno to **S_objLib_OBJ_TIMEOUT** (defined in **objLib.h**).

The *VXWSem::take***( )** routine must not be called from interrupt service routines.

RETURNS OK, or ERROR if the task timed out.

SEE ALSO **vxwSemLib**, *VXWCSem::VXWCsem***( )**, *VXWBSem::VXWBsem***( )**, *VXWMSem::VXWMsem***( )**, *VxWorks Programmer's Guide: Basic OS*

VXWSem::VXWSem( )

NAME *VXWSem::VXWSem*( ) – build semaphore object from semaphore ID (WFC Opt.)

SYNOPSIS
```
VXWSem (SEM_ID id)
```

DESCRIPTION Use this constructor to manipulate a semaphore that was not created using C++ interfaces. The argument *id* is the semaphore identifier returned and used by the C interface to the VxWorks semaphore facility.

RETURNS N / A.

SEE ALSO **vxwSemLib**, **semLib**

VXWSem::~VXWSem( )

NAME *VXWSem::~VXWSem*( ) – delete a semaphore (WFC Opt.)

SYNOPSIS
```
virtual ~VXWSem ()
```

DESCRIPTION This destructor terminates and deallocates any memory associated with a specified semaphore. Any pended tasks 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 N / A.

SEE ALSO **vxwSemLib**, *VxWorks Programmer's Guide: Basic OS*

VXWSmBSem::VXWSmBSem( )

NAME *VXWSmBSem::VXWSmBSem*( ) – create and initialize a binary shared-memory semaphore (WFC Opt.)

SYNOPSIS
```
VXWSmBSem (int opts, SEM_B_STATE istate, char * name = 0)
```

DESCRIPTION This routine allocates and initializes a shared memory binary semaphore. The semaphore is initialized to an initial state *istate* of either **SEM_FULL** (available) or **SEM_EMPTY** (not available). The shared semaphore structure is allocated from the shared semaphore dedicated memory partition. Use the optional *name* argument to identify the new semaphore by name.

The queuing style for blocked tasks is set by *opts*; the only supported queuing style for shared memory semaphores is first-in-first-out, selected by **SEM_Q_FIFO**.

The maximum number of shared memory semaphores (binary plus counting) that can be created is **SM_OBJ_MAX_SEM**, defined in **configAll.h**.

AVAILABILITY This routine depends on code distributed as a component of the unbundled shared memory support option, VxMP.

RETURNS N / A.

ERRNO
S_smMemLib_NOT_ENOUGH_MEMORY
S_semLib_INVALID_QUEUE_TYPE
S_semLib_INVALID_STATE
S_smObjLib_LOCK_TIMEOUT

SEE ALSO **vxwSmLib**, **vxwSemLib**, **vxwSmNameLib**

VXWSmBSem::VXWSmBSem( )

NAME *VXWSmBSem::VXWSmBSem*( ) – build a binary shared-memory semaphore object (WFC Opt.)

SYNOPSIS
```
VXWSmBSem (char * name, int waitType)
```

DESCRIPTION This routine builds a shared-memory binary semaphore object around an existing named shared-memory semaphore. The *name* argument identifies the existing semaphore; *waitType* specifies whether to wait if the desired name is not found in the shared-memory name database; see *VXWSmName::nameGet*( ).

Use this routine to take advantage of the VXWSmBSem class while working with semaphores created by some other means (for example, previously existing C code).

RETURNS N / A.

SEE ALSO **vxwSmLib**, *VXWSmName::nameGet***( )**

---

*VXWSmCSem::VXWSmCSem***( )**

NAME *VXWSmCSem::VXWSmCSem***( )** – create and initialize a shared memory counting semaphore (WFC Opt.)

SYNOPSIS

```
VXWSmCSem (int opts, int icount, char * name = 0)
```

DESCRIPTION This routine allocates and initializes a shared memory counting semaphore. The initial count value of the semaphore (the number of times the semaphore must be taken before it can be given) is specified by *icount*.

The queuing style for blocked tasks is set by *opts*; the only supported queuing style for shared memory semaphores is first-in-first-out, selected by **SEM_Q_FIFO**.

The maximum number of shared memory semaphores (binary plus counting) that can be created is **SM_OBJ_MAX_SEM**, defined in **configAll.h**.

AVAILABILITY This routine depends on code distributed as a component of the unbundled shared memory support option, VxMP.

RETURNS N / A.

ERRNO S_smMemLib_NOT_ENOUGH_MEMORY
S_semLib_INVALID_QUEUE_TYPE
S_smObjLib_LOCK_TIMEOUT

SEE ALSO **vxwSmLib**, **vxwSemLib**, **vxwSmNameLib**

VXWSmCSem::VXWSmCSem( )

NAME *VXWSmCSem::VXWSmCSem*( ) – build a shared-memory counting semaphore object (WFC Opt.)

SYNOPSIS
```
VXWSmCSem (char * name, int waitType)
```

DESCRIPTION This routine builds a shared-memory semaphore object around an existing named shared-memory counting semaphore. The *name* argument identifies the existing semaphore, and *waitType* specifies whether to wait if the desired name is not found in the shared-memory name database; see *VXWSmName::nameGet*( ).

Use this routine to take advantage of the VXWSmBSem class while working with semaphores created by some other means (for example, previously existing C code).

RETURNS N / A.

SEE ALSO **vxwSmLib**, *VXWSmName::nameGet*( )

VXWSmMemBlock::baseAddress( )

NAME *VXWSmMemBlock::baseAddress*( ) – address of shared-memory block (WFC Opt.)

SYNOPSIS
```
void * baseAddress () const
```

DESCRIPTION This routine reports the local address of a block of shared memory managed as a **VXWSmMemBlock** object.

RETURNS Local address of memory block in shared-memory system partition.

SEE ALSO **vxwSmLib**

VXWSmMemBlock::VXWSmMemBlock( )

NAME *VXWSmMemBlock::VXWSmMemBlock*( ) – allocate a block of memory from the shared memory system partition (WFC Opt.)

SYNOPSIS
```
VXWSmMemBlock (int nBytes)
```

DESCRIPTION This routine allocates, from the shared memory system partition, a block of memory whose size is equal to or greater than *nBytes*. The local address of the allocated shared memory block can be obtained from *VXWSmMemBlock::baseAddress*( ).

AVAILABILITY This routine depends on code distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS N / A.

ERRNO S_memLib_NOT_ENOUGH_MEMORY
S_smObjLib_LOCK_TIMEOUT

SEE ALSO **vxwSmLib**

VXWSmMemBlock::VXWSmMemBlock( )

NAME *VXWSmMemBlock::VXWSmMemBlock*( ) – allocate memory for an array from the shared memory system partition (WFC Opt.)

SYNOPSIS
```
VXWSmMemBlock (int nElems, int sizeOfElem)
```

DESCRIPTION This routine allocates a block of memory for an array that contains *nElems* elements of size *sizeOfElem* from the shared memory system partition. The local address of the allocated shared memory block can be obtained from *VXWSmMemBlock::baseAddress*( ).

AVAILABILITY This routine depends on code 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 **vxwSmLib**

VXWSmMemBlock::~VXWSmMemBlock( )

NAME *VXWSmMemBlock::~VXWSmMemBlock*( ) – free a shared memory system partition block of memory (WFC Opt.)

SYNOPSIS
```
virtual ~VXWSmMemBlock ()
```

DESCRIPTION This routine returns a **VXWSmMemBlock** shared-memory block to the free-memory pool in the shared-memory system partition.

AVAILABILITY This routine depends on code distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS N / A

ERRNO S_smObjLib_LOCK_TIMEOUT

SEE ALSO **vxwSmLib**, *VXWSmMemBlock::VXWSmMemBlock*( )

VXWSmMemPart::VXWSmMemPart( )

NAME *VXWSmMemPart::VXWSmMemPart*( ) – create a shared memory partition (WFC Opt.)

SYNOPSIS
```
VXWSmMemPart (char *pool, unsigned poolSize)
```

DESCRIPTION This routine creates a shared memory partition that can be used by tasks on all CPUs in the system to manage memory blocks . Because the **VXWSmMemPart** class inherits from **VXWMemPart**, you can use the general-purpose methods in that class to manage the partition (that is, to allocate and free memory blocks in the partition).

pool is a pointer to the global address of shared memory dedicated to the partition. The memory area where *pool* points 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.

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**, defined in **configAll.h**. Memory pool size is rounded down to a 16-byte boundary.

AVAILABILITY This routine depends on code distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS N / A.

ERRNO S_memLib_NOT_ENOUGH_MEMORY
S_smObjLib_LOCK_TIMEOUT

SEE ALSO **vxwSmLib**, **vxwMemPartLib**, **vxwSmNameLib**

---

*VXWSmMsgQ::VXWSmMsgQ***( )**

NAME *VXWSmMsgQ::VXWSmMsgQ***( )** – create and initialize a shared-memory message queue (WFC Opt.)

SYNOPSIS
```
VXWSmMsgQ (int maxMsgs, int maxMsgLength, int options)
```

DESCRIPTION This routine creates a shared memory message queue capable of holding up to *maxMsgs* messages, each up to *maxMsgLength* bytes long. The queue can only be created with the option **MSG_Q_FIFO** (0), thus queuing pended tasks in FIFO order.

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 fails with **errno** set to **S_smMemLib_NOT_ENOUGH_MEMORY**. This problem can be solved by incrementing the **SM_OBJ_MAX_MSG_Q** value in **configAll.h** and / or the size of memory dedicated to shared-memory objects **SM_OBJ_MEM_SIZE** in **config.h**.

AVAILABILITY This routine depends on code distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS N / A.

ERRNO S_smMemLib_NOT_ENOUGH_MEMORY
S_intLib_NOT_ISR_CALLABLE
S_msgQLib_INVALID_QUEUE_TYPE
S_smObjLib_LOCK_TIMEOUT

SEE ALSO **vxwSmLib**, **vxwMsgQLib**, **vxwSmNameLib**

VXWSmName::nameGet( )

NAME *VXWSmName::nameGet*( ) – get name and type of a shared memory object (VxMP Opt.) (WFC Opt.)

SYNOPSIS
```
STATUS nameGet (char * name, int * pType, int waitType)
```

DESCRIPTION This routine searches the shared memory name database for an object matching the this VXWSmName instance. 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 depends on 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 **vxwSmNameLib**

VXWSmName::nameGet( )

NAME *VXWSmName::nameGet*( ) – get name of a shared memory object (VxMP Opt.) (WFC Opt.)

SYNOPSIS
```
STATUS nameGet (char * name, int waitType)
```

DESCRIPTION This routine searches the shared memory name database for an object matching the this VXWSmName instance. If the object is found, its name is copied to the address pointed to by *name*. 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 depends on 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 **vxwSmNameLib**

---

*VXWSmName::nameSet***( )**

NAME *VXWSmName::nameSet***( )** – define a name string in the shared-memory name database (VxMP Opt.) (WFC Opt.)

SYNOPSIS
```
virtual STATUS nameSet (char * name) = 0;
```

DESCRIPTION This routine adds a name of the type appropriate for each derived class to the database of memory object names.

The *name* parameter is an arbitrary null-terminated string with a maximum of 20 characters, including EOS.

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 depends on 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 **vxwSmNameLib**

VXWSmName::~VXWSmName( )

NAME *VXWSmName::~VXWSmName*( ) – remove an object from the shared memory objects name database (VxMP Opt.) (WFC Opt.)

SYNOPSIS
```
virtual ~VXWSmName ();
```

DESCRIPTION This routine removes an object from the shared memory objects name database.

AVAILABILITY This routine depends on code distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS OK, or ERROR if the database is not initialized, or the name-database lock times out.

ERRNO S_smNameLib_NOT_INITIALIZED
S_smObjLib_LOCK_TIMEOUT

SEE ALSO **vxwSmNameLib**

VXWSymTab::add( )

NAME *VXWSymTab::add*( ) – create and add a symbol to a symbol table, including a group number (WFC Opt.)

SYNOPSIS
```
STATUS add (char *name, char *value, SYM_TYPE type, UINT16 group)
```

DESCRIPTION This routine allocates a symbol *name* and adds it to its symbol table with the specified parameters *value*, *type*, and *group*. The *group* parameter specifies the group number assigned to a module when it is loaded on the target; see the manual entry for **moduleLib**.

RETURNS OK, or ERROR if there is insufficient memory for the symbol to be allocated.

SEE ALSO **vxwSymLib**, **moduleLib**

VXWSymTab::each( )

NAME *VXWSymTab::each*( ) – call a routine to examine each entry in a symbol table (WFC Opt.)

SYNOPSIS

```
SYMBOL * each (FUNCPTR routine, int routineArg)
```

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 must have the following type signature:

```
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 must return TRUE if *VXWSymTab::each*( ) is to continue calling it for each entry, or FALSE if it is done and *VXWSymTab::each*( ) can exit.

RETURNS A pointer to the last symbol reached, or NULL if all symbols are reached.

SEE ALSO **vxwSymLib**

VXWSymTab::findByName( )

NAME *VXWSymTab::findByName*( ) – look up a symbol by name (WFC Opt.)

SYNOPSIS

```
STATUS findByName (char *name, char **pValue, SYM_TYPE *pType) const
```

DESCRIPTION This routine searches its 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.

RETURNS OK, or ERROR if the symbol cannot be found.

SEE ALSO **vxwSymLib**

VXWSymTab::findByNameAndType( )

NAME *VXWSymTab::findByNameAndType*( ) – look up a symbol by name and type (WFC Opt.)

SYNOPSIS
```
STATUS findByNameAndType (char *name, char **pValue, SYM_TYPE *pType,
SYM_TYPE goalType, SYM_TYPE mask)  const
```

DESCRIPTION This routine searches its symbol table for a symbol matching both name and type (*name* and *goalType*). 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.

RETURNS OK, or ERROR if the symbol is not found.

SEE ALSO **vxwSymLib**

VXWSymTab::findByValue( )

NAME *VXWSymTab::findByValue*( ) – look up a symbol by value (WFC Opt.)

SYNOPSIS
```
STATUS findByValue (UINT value, char *name, int *pValue,
SYM_TYPE *pType)  const
```

DESCRIPTION This routine searches its 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 value, and the type are copied to *name*, *pValue*, and *pType*.

RETURNS OK, or ERROR if *value* is less than the lowest value in the table.

SEE ALSO **vxwSymLib**

VXWSymTab::findByValueAndType( )

NAME *VXWSymTab::findByValueAndType*( ) – look up a symbol by value and type (WFC Opt.)

SYNOPSIS
```
STATUS findByValueAndType (UINT value, char *name, int *pValue,
SYM_TYPE *pType, SYM_TYPE  goalType,
SYM_TYPE mask)             const
```

DESCRIPTION This routine searches a symbol table for a symbol matching both value and type (*value* and *goalType*). 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.

RETURNS OK, or ERROR if *value* is less than the lowest value in the table.

SEE ALSO **vxwSymLib**

---

*VXWSymTab::remove***( )**

NAME *VXWSymTab::remove***( )** – remove a symbol from a symbol table (WFC Opt.)

SYNOPSIS
```
STATUS remove (char *name, SYM_TYPE type)
```

DESCRIPTION This routine removes a symbol of matching name and type from its 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 **vxwSymLib**

---

*VXWSymTab::VXWSymTab***( )**

NAME *VXWSymTab::VXWSymTab***( )** – create a symbol table (WFC Opt.)

SYNOPSIS
```
VXWSymTab (int hashSizeLog2, BOOL sameNameOk, PART_ID symPartId)
```

DESCRIPTION This constructor 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 N / A

SEE ALSO **vxwSymLib**

VXWSymTab::VXWSymTab( )

NAME *VXWSymTab::VXWSymTab*( ) – create a symbol-table object (WFC Opt.)

SYNOPSIS

```
VXWSymTab (SYMTAB_ID aSymTabId)
```

DESCRIPTION This constructor creates a symbol table object based on an existing symbol table. For example, the following statement creates a symbol-table object for the VxWorks system symbol table (assuming you have configured a target-resident symbol table into your VxWorks system):

```
VXWSymTab sSym;
sSym = VXWSymTab (sysSymTbl);
```

SEE ALSO **vxwSymLib**

VXWSymTab::~VXWSymTab( )

NAME *VXWSymTab::~VXWSymTab*( ) – delete a symbol table (WFC Opt.)

SYNOPSIS

```
~VXWSymTab ()
```

DESCRIPTION This routine deletes a symbol table; it deallocates all memory associated with its symbol table, including the hash table, and marks the table as invalid.

Deletion of a table that still contains symbols throws an 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 table still contains symbols.

SEE ALSO **vxwSymLib**

*VXWTask::activate***( )**

NAME *VXWTask::activate***( )** – activate a task (WFC Opt.)

SYNOPSIS

```
STATUS activate ()
```

DESCRIPTION This routine activates tasks created by the form of the constructor that does not automatically activate a task. Without activation, a task is ineligible for CPU allocation by the scheduler.

RETURNS OK, or ERROR if the task cannot be activated.

SEE ALSO **vxwTaskLib**, *VXWTask::VXWTask***( )**

*VXWTask::deleteForce***( )**

NAME *VXWTask::deleteForce***( )** – delete a task without restriction (WFC Opt.)

SYNOPSIS

```
STATUS deleteForce ()
```

DESCRIPTION This routine deletes a task even if the task is protected from deletion. It is similar to *VXWTask::~VXWTask***( )**. Upon deletion, all routines specified by *taskDeleteHookAdd***( )** are called in the context of the deleting task.

CAVEATS This routine is intended as a debugging aid, and is generally inappropriate for applications. Disregarding a task's deletion protection could leave the the system in an unstable state or lead to system deadlock.

The system does not protect against simultaneous *VXWTask:deleteForce***( )** calls. Such a situation could leave the system in an unstable state.

RETURNS OK, or ERROR if the task cannot be deleted.

SEE ALSO **vxwTaskLib**, *taskDeleteHookAdd***( )**, *VXWTask::~VXWTask***( )**

VXWTask::envCreate( )

NAME *VXWTask::envCreate*( ) – create a private environment (WFC Opt.)

SYNOPSIS
```
STATUS envCreate (int envSource)
```

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 **vxwTaskLib**, **envLib**

VXWTask::errNo( )

NAME *VXWTask::errNo*( ) – retrieve error status value (WFC Opt.)

SYNOPSIS
```
int errNo () const
```

DESCRIPTION This routine gets the error status for the task.

RETURNS The error status value contained in **errno**.

SEE ALSO **vxwTaskLib**

VXWTask::errNo( )

NAME *VXWTask::errNo*( ) – set error status value (WFC Opt.)

SYNOPSIS
```
STATUS errNo (int errorValue)
```

DESCRIPTION This routine sets the error status value for its task.

RETURNS OK.

SEE ALSO **vxwTaskLib**

VXWTask::id( )

NAME *VXWTask::id*( ) – reveal task ID (WFC Opt.)

SYNOPSIS
```
int id () const
```

DESCRIPTION This routine reveals the task ID for its task. The task ID is necessary to call C routines that affect or inquire on a task.

RETURNS task ID

SEE ALSO **vxwTaskLib**, **taskLib**

VXWTask::info( )

NAME *VXWTask::info*( ) – get information about a task (WFC Opt.)

SYNOPSIS
```
STATUS info (TASK_DESC *pTaskDesc) const
```

DESCRIPTION This routine fills in a specified task descriptor (**TASK_DESC**) for its 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

SEE ALSO **vxwTaskLib**

VXWTask::isReady( )

NAME *VXWTask::isReady*( ) – check if task is ready to run (WFC Opt.)

SYNOPSIS
```
BOOL isReady () const
```

DESCRIPTION This routine tests the status field of its task to determine whether the task is ready to run.

RETURNS TRUE if the task is ready, otherwise FALSE.

SEE ALSO **vxwTaskLib**

VXWTask::isSuspended( )

NAME *VXWTask::isSuspended*( ) – check if task is suspended (WFC Opt.)

SYNOPSIS
```
BOOL isSuspended () const
```

DESCRIPTION This routine tests the status field of its task to determine whether the task is suspended.

RETURNS TRUE if the task is suspended, otherwise FALSE.

SEE ALSO **vxwTaskLib**

VXWTask::kill( )

NAME *VXWTask::kill*( ) – send a signal to task (WFC Opt.)

SYNOPSIS
```
int kill (int signo)
```

DESCRIPTION This routine sends a signal *signo* to its task.

RETURNS OK (0), or ERROR (-1) if the signal number is invalid.

ERRNO **EINVAL**

SEE ALSO **vxwTaskLib**

VXWTask::name( )

NAME *VXWTask::name*( ) – get the name associated with a task ID (WFC Opt.)

SYNOPSIS
```
char * name () const
```

DESCRIPTION This routine returns a pointer to the name of its task, if it has a name; otherwise it returns NULL.

RETURNS A pointer to the task name, or NULL.

SEE ALSO **vxwTaskLib**

VXWTask::options( )

NAME *VXWTask::options*( ) – examine task options (WFC Opt.)

SYNOPSIS
```
STATUS options (int *pOptions) const
```

DESCRIPTION This routine gets the current execution options of its task. The option bits returned indicate the following modes:

VX_FP_TASK
execute with floating-point coprocessor support.

VX_PRIVATE_ENV
include private environment support (see **envLib**).

VX_NO_STACK_FILL
do not fill the stack for use by *checkstack*( ).

VX_UNBREAKABLE
do not allow breakpoint debugging.

For definitions, see **taskLib.h**.

RETURNS OK.

SEE ALSO **vxwTaskLib**

VXWTask::*options*( )

NAME *VXWTask*::*options*( ) – change task options (WFC Opt.)

SYNOPSIS
```
STATUS options (int mask, int newOptions)
```

DESCRIPTION This routine changes the execution options of its 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.

SEE ALSO **vxwTaskLib**

VXWTask::*priority*( )

NAME *VXWTask*::*priority*( ) – examine the priority of task (WFC Opt.)

SYNOPSIS
```
STATUS priority (int *pPriority) const
```

DESCRIPTION This routine reports the current priority of its task. The current priority is copied to the integer pointed to by *pPriority*.

RETURNS OK.

SEE ALSO **vxwTaskLib**

VXWTask::*priority*( )

NAME *VXWTask*::*priority*( ) – change the priority of a task (WFC Opt.)

SYNOPSIS
```
STATUS priority (int newPriority)
```

DESCRIPTION This routine changes its task's priority to a specified priority. Priorities range from 0, the highest priority, to 255, the lowest priority.

RETURNS OK.

SEE ALSO **vxwTaskLib**

*VXWTask::registers***( )**

NAME *VXWTask::registers***( )** – set a task's registers (WFC Opt.)

SYNOPSIS

```
STATUS registers (const REG_SET *pRegs)
```

DESCRIPTION This routine loads a specified register set *pRegs* into the 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.

SEE ALSO **vxwTaskLib**, *VXWTask::suspend***( )**

*VXWTask::registers***( )**

NAME *VXWTask::registers***( )** – get task registers from the TCB (WFC Opt.)

SYNOPSIS

```
STATUS registers (REG_SET *pRegs) const
```

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.

SEE ALSO **vxwTaskLib**, *VXWTask::suspend***( )**

VXWTask::restart( )

NAME *VXWTask::restart*( ) – restart task (WFC Opt.)

SYNOPSIS
```
STATUS restart ()
```

DESCRIPTION This routine "restarts" its task. The task is first terminated, and then reinitialized with the same ID, priority, options, original entry point, stack size, and parameters it had when it was terminated. Self-restarting of a calling task is performed by the exception task.

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 could not be restarted.

SEE ALSO **vxwTaskLib**

VXWTask::resume( )

NAME *VXWTask::resume*( ) – resume task (WFC Opt.)

SYNOPSIS
```
STATUS resume ()
```

DESCRIPTION This routine resumes its task. Suspension is cleared, and the task operates in the remaining state.

RETURNS OK, or ERROR if the task cannot be resumed.

SEE ALSO **vxwTaskLib**

VXWTask::show( )

NAME *VXWTask::show*( ) – display the contents of task registers (WFC Opt.)

SYNOPSIS

```
void show () const
```

DESCRIPTION This routine displays the register contents of its task on standard output.

EXAMPLE The following shell command line displays the register of a task **vxwT28**:

```
-> vxwT28.show ()
```

The example prints on standard output a display like the following (68000 family):

```
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
```

RETURNS N / A

SEE ALSO **vxwTaskLib**

VXWTask::show( )

NAME *VXWTask::show*( ) – display task information from TCBs (WFC Opt.)

SYNOPSIS

```
STATUS show (int level) const
```

DESCRIPTION This routine displays the contents of its task's task control block (TCB). 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*( )

Field	Meaning
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 a task named **t28**:

```
  NAME         ENTRY    TID    PRI  STATUS      PC       SP     ERRNO  DELAY
---------- --------- -------- --- --------- -------- -------- ------ -----
t28        _appStart 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
```

RETURNS N / A

SEE ALSO **vxwTaskLib**, *VXWTaskstatusString***( )**, *Tornado User's Guide: The Tornado Shell*

*VXWTask::sigqueue***( )**

NAME *VXWTask::sigqueue***( )** – send a queued signal to task (WFC Opt.)

SYNOPSIS

```
int sigqueue (int signo, const union sigval value)
```

DESCRIPTION The routine *sigqueue***( )** sends to its task the signal specified by *signo* with the signal-parameter value specified by *value*.

RETURNS OK (0), or ERROR (-1) if the signal number is invalid, or if there are no queued-signal buffers available.

ERRNO **EINVAL**, **EAGAIN**

SEE ALSO **vxwTaskLib**

*VXWTask::SRSet***( )**

NAME *VXWTask::SRSet***( )** – set the task status register (MC680x0, MIPS, i386/i486) (WFC Opt.)

SYNOPSIS
```
STATUS SRSet (UINT16 sr)
```
SYNOPSIS (X86)
```
STATUS SRSet (UINT sr)
```
SYNOPSIS (MIPS)
```
STATUS SRSet (UINT32 sr)
```

DESCRIPTION This routine sets the status register of a task that is not running; that is, you must not call this *->SRSet***( )**. Debugging facilities use this routine to set the trace bit in the status register of a task that is being single-stepped.

RETURNS OK.

SEE ALSO **vxwTaskLib**

*VXWTask::statusString***( )**

NAME *VXWTask::statusString***( )** – get task status as a string (WFC Opt.)

SYNOPSIS
```
STATUS statusString (char *pString) const
```

DESCRIPTION This routine deciphers the WIND task status word in the TCB for its 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).
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.

RETURNS OK.

SEE ALSO **vxwTaskLib**

VXWTask::suspend( )

NAME *VXWTask::suspend*( ) – suspend task (WFC Opt.)

SYNOPSIS

```
STATUS suspend ()
```

DESCRIPTION This routine suspends its 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 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.

SEE ALSO **vxwTaskLib**

VXWTask::tcb( )

NAME *VXWTask::tcb*( ) – get the task control block (WFC Opt.)

SYNOPSIS

```
WIND_TCB * tcb () const
```

DESCRIPTION This routine returns a pointer to the task control block (**WIND_TCB**) for its task. Although all task state information is contained in the TCB, users must not modify it directly. To change registers, for instance, use *VXWTask::registers*( ).

RETURNS A pointer to a **WIND_TCB**.

SEE ALSO **vxwTaskLib**

*VXWTask::varAdd***( )**

NAME *VXWTask::varAdd***( )** – add a task variable to task (WFC Opt.)

SYNOPSIS

```
STATUS varAdd (int * pVar)
```

DESCRIPTION This routine adds a specified variable *pVar* (4-byte memory location) to its task's context. After calling this routine, the variable is private to the task. The task can access and modify the variable, but the modifications are not visible to other tasks, and other tasks' modifications to that variable do 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 has its own stack, and thus separate stack variables, they all share the same static and global variables. To make a variable *not* shareable, the routine can call *VXWTask::varAdd***( )** 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 are spawned with a main 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
VXWTask   me;         // task object for self
void operator
    (
    int opNum         // number of this operator task
    )
    {
    me = VXWTask (0); // task object for running task
    if (me.varAdd ((int *)&opGlobal) != OK)
        {
        printErr ("operator%d: can't VXWTask::varAdd opGlobal\n", opNum);
        me.suspend ();
        }
    if ((opGlobal = (OP_GLOBAL *) malloc (sizeof (OP_GLOBAL))) == NULL)
        {
        printErr ("operator%d: can't malloc opGlobal\n", opNum);
        me.suspend ();
        }
    ...
    }
```

RETURNS OK, or ERROR if memory is insufficient for the task variable descriptor.

SEE ALSO **vxwTaskLib**, *VXWTask::varDelete***( )**, *VXWTask::varGet***( )**, *VXWTask::varSet***( )**

*VXWTask::varDelete***( )**

NAME *VXWTask::varDelete***( )** – remove a task variable from task (WFC Opt.)

SYNOPSIS

```
STATUS varDelete (int * pVar)
```

DESCRIPTION This routine removes a specified task variable, *pVar*, from its task's context. The private value of that variable is lost.

RETURNS OK, or ERROR if the task variable does not exist for the task.

SEE ALSO **vxwTaskLib**, *VXWTask::varAdd***( )**, *VXWTask::varGet***( )**, *VXWTask:varSet***( )**

*VXWTask::varGet***( )**

NAME *VXWTask::varGet***( )** – get the value of a task variable (WFC Opt.)

SYNOPSIS

```
int varGet (int * pVar) const
```

DESCRIPTION This routine returns the private value of a task variable for its task. The 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 does not own the task variable.

SEE ALSO **vxwTaskLib**, *VXWTask::varAdd***( )**, *VXWTask::varDelete***( )**, *VXWTask::varSet***( )**

VXWTask::varInfo( )

NAME *VXWTask::varInfo*( ) – get a list of task variables (WFC Opt.)

SYNOPSIS
```
int varInfo (TASK_VAR varList [], int maxVars) const
```

DESCRIPTION This routine provides the calling task with a list of all of the task variables of its task. The unsorted array of task variables is copied to *varList*.

CAVEATS Kernel rescheduling is disabled 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 **vxwTaskLib**

VXWTask::varSet( )

NAME *VXWTask::varSet*( ) – set the value of a task variable (WFC Opt.)

SYNOPSIS
```
STATUS varSet (int * pVar, int value)
```

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 does not own the task variable.

SEE ALSO **vxwTaskLib**, *VXWTask::varAdd*( ), *VXWTask::varDelete*( ), *VXWTask::varGet*( )

*VXWTask::VXWTask***( )**

NAME *VXWTask::VXWTask***( )** – initialize a task object (WFC Opt.)

SYNOPSIS

```
VXWTask (int tid)
```

DESCRIPTION This constructor creates a task object from the task ID of an existing task. Because of the VxWorks convention that a task ID of 0 refers to the calling task, this constructor can be used to derive a task object for the calling task, as follows:

```
myTask = VXWTask (0);
```

RETURNS N / A

SEE ALSO **vxwTaskLib**, **taskLib**, *VXWTask::~VXWTask***( )**, *sp***( )**

*VXWTask::VXWTask***( )**

NAME *VXWTask::VXWTask***( )** – create and spawn a task (WFC Opt.)

SYNOPSIS

```
VXWTask (char       * name,
int          priority,
int          options,
int          stackSize,
FUNCPTR      entryPoint,
int arg1 =   0,
int arg2 =   0,
int arg3 =   0,
int arg4 =   0,
int arg5 =   0,
int arg6 =   0,
int arg7 =   0,
int arg8 =   0,
int arg9 =   0,
int arg10 =  0)
```

DESCRIPTION This constructor creates and activates a new task with a specified priority and options.

A task may be assigned a name as a debugging aid. This name appears 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 is 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 is called after the C environment is set up. The specified routine is called with the ten arguments provided. Should the specified main routine return, a call to *exit***( )** is made automatically.

Note that ten (and only ten) arguments must be passed for the spawned function.

Bits in the options argument may be set to run with the following modes:

VX_FP_TASK
execute with floating-point coprocessor support.

VX_PRIVATE_ENV
include private environment support.

VX_NO_STACK_FILL
do not fill the stack for use by *checkStack***( )**.

VX_UNBREAKABLE
do not allow breakpoint debugging.

See the definitions in **taskLib.h**.

RETURNS N / A

SEE ALSO **vxwTaskLib**, *VXWTask::~VXWTask***( )**, *VXWTask::activate***( )**, *sp***( )**, *VxWorks Programmer’s Guide: Basic OS*

*VXWTask::VXWTask***( )**

NAME *VXWTask::VXWTask***( )** – initialize a task with a specified stack (WFC Opt.)

SYNOPSIS

```
VXWTask (WIND_TCB   * pTcb,
char *       name,
int          priority,
int          options,
char *       pStackBase,
int          stackSize,
FUNCPTR      entryPoint,
int arg1 =   0,
int arg2 =   0,
int arg3 =   0,
int arg4 =   0,
int arg5 =   0,
int arg6 =   0,
int arg7 =   0,
int arg8 =   0,
int arg9 =   0,
int arg10 =  0)
```

DESCRIPTION This constructor initializes user-specified regions of memory for a task stack and control block instead of allocating them from memory. This constructor uses 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 other constructors, a task may be given a name. If no name is specified, this constructor creates a task without a name (rather than assigning a default name).

Other arguments are the same as in the previous constructor. This constructor does not activate the task. This must be done by calling *VXWTask::activate***( )**.

Normally, tasks should be started using the previous constructor rather than this one, 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.

SEE ALSO **vxwTaskLib**, *VXWTask::activate***( )**

*VXWTask::~VXWTask***( )**

NAME *VXWTask::~VXWTask***( )** – delete a task (WFC Opt.)

SYNOPSIS

```
virtual ~VXWTask ()
```

DESCRIPTION This destructor causes the task to cease to exist and deallocates the stack and **WIND_TCB** memory resources. Upon deletion, all routines specified by *taskDeleteHookAdd***( )** are called in the context of the deleting task.

RETURNS N / A

SEE ALSO **vxwTaskLib**, **excLib**, *taskDeleteHookAdd***( )**, *VXWTask::VXWTask***( )**, *VxWorks Programmer's Guide: Basic OS*

*VXWWd::cancel***( )**

NAME *VXWWd::cancel***( )** – cancel a currently counting watchdog (WFC Opt.)

SYNOPSIS

```
STATUS 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 **vxwWdLib**, *VXWWd::start***( )**

*VXWWd::start***( )**

NAME *VXWWd::start***( )** – start a watchdog timer (WFC Opt.)

SYNOPSIS

```
STATUS start (int delay, FUNCPTR pRoutine, int parameter)
```

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 *VXWWd::start***( )** again; only the most recent *VXWWd::start***( )** on a given watchdog ID has any effect. (If your application requires multiple watchdog routines, use *VXWWd::VXWWd***( )** to generate separate a watchdog for each.) To cancel a watchdog timer before the specified tick count is reached, call *VXWWd::cancel***( )**.

Watchdog timers execute only once, but some applications require periodically executing timers. To achieve this effect, the timer routine itself must call *VXWWd::start***( )** 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 **vxwWdLib**, *VXWWd::cancel***( )**

*VXWWd::VXWWd***( )**

NAME *VXWWd::VXWWd***( )** – construct a watchdog timer (WFC Opt.)

SYNOPSIS

```
VXWWd ( )
```

DESCRIPTION This routine creates a watchdog timer.

RETURNS N / A

SEE ALSO **vxwWdLib**, *VXWWd::~VXWWd***( )**

*VXWWd::VXWWd***( )**

NAME *VXWWd::VXWWd***( )** – construct a watchdog timer (WFC Opt.)

SYNOPSIS

```
VXWWd (WDOG_ID aWdId)
```

DESCRIPTION This routine creates a watchdog timer from an existing **WDOG_ID**.

RETURNS N / A

SEE ALSO **vxwWdLib**, *VXWWd::~VXWWd***( )**

VXWWd::~VXWWd( )

NAME *VXWWd::~VXWWd*( ) – destroy a watchdog timer (WFC Opt.)

SYNOPSIS
```
~VXWWd ( )
```

DESCRIPTION This routine destroys a watchdog timer. The watchdog will be removed from the timer queue if it has been started.

RETURNS N / A

SEE ALSO **vxwWdLib**, *VXWWd::VXWWd*( )

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**

wd33c93CtrlCreate( )

NAME *wd33c93CtrlCreate*( ) – create and partially initialize a WD33C93 SBIC structure

SYNOPSIS

```
WD_33C93_SCSI_CTRL *wd33c93CtrlCreate
    (
    UINT8    *sbicBaseAdrs,   /* base address of SBIC                   */
    int      regOffset,       /* addr offset between consecutive regs. */
    UINT     clkPeriod,       /* period of controller clock (nsec)     */
    int      devType,         /* SBIC device type                      */
    FUNCPTR  sbicScsiReset,   /* SCSI bus reset function               */
    FUNCPTR  sbicDmaBytesIn,  /* SCSI DMA input function               */
    FUNCPTR  sbicDmaBytesOut  /* SCSI DMA output function              */
    )
```

DESCRIPTION This routine creates an SBIC data structure and must be called before using an SBIC chip. It should be called once and only once for a specified SBIC. Since it allocates memory for a structure needed by all routines in **wd33c93Lib**, it must be called before any other routines in the library. After calling this routine, at least one call to *wd33c93CtrlInit*( ) should be made before any SCSI transaction is initiated using the SBIC.

Note that only the non-multiplexed processor interface is supported.

The input parameters are as follows:

sbicBaseAdrs
the address where the CPU accesses the lowest register of the SBIC.

regOffset
the address offset (in bytes) to access consecutive registers. (This must be a power of 2; for example, 1, 2, 4, etc.)

clkPeriod
the period, in nanoseconds, of the signal-to-SBIC clock input used only for select command timeouts.

devType
a constant corresponding to the type (part number) of this controller; possible options are enumerated in **wd33c93.h** under the heading "SBIC device type."

sbicScsiReset
a board-specific routine to assert the RST line on the SCSI bus, which causes all connected devices to return to a known quiescent state.

spcDmaBytesIn and *spcDmaBytesOut*
board-specific routines to handle DMA input and output. If these are NULL (0), SBIC program transfer mode is used. DMA is implemented only during SCSI data in/out phases. The interface to these DMA routines must be of the form:

```
STATUS xxDmaBytes{In, Out}
    (
    SCSI_PHYS_DEV  *pScsiPhysDev,  /* ptr to phys dev info    */
    UINT8          *pBuffer,       /* ptr to the data buffer */
    int            bufLength       /* number of bytes to xfer */
    )
```

RETURNS A pointer to the SBIC control structure, or NULL if memory is insufficient or parameters are invalid.

SEE ALSO **wd33c93Lib1**, **wd33c93.h**

---

*wd33c93CtrlCreateScsi2***( )**

NAME *wd33c93CtrlCreateScsi2***( )** – create and partially initialize an SBIC structure

SYNOPSIS

```
WD_33C93_SCSI_CTRL *wd33c93CtrlCreateScsi2
    (
    UINT8    *sbicBaseAdrs,       /* base address of the SBIC             */
    int      regOffset,           /* address offset between SBIC registers */
```

```
    UINT     clkPeriod,           /* period of the SBIC clock (nsec)       */
    FUNCPTR  sysScsiBusReset,     /* function to reset SCSI bus            */
    int      sysScsiResetArg,     /* argument to pass to above function    */
    UINT     sysScsiDmaMaxBytes,  /* maximum byte count using DMA          */
    FUNCPTR  sysScsiDmaStart,     /* function to start SCSI DMA transfer   */
    FUNCPTR  sysScsiDmaAbort,     /* function to abort SCSI DMA transfer   */
    int      sysScsiDmaArg        /* argument to pass to above functions   */
    )
```

DESCRIPTION

This routine creates an SBIC data structure and must be called before using an SBIC chip. It must be called exactly once for a specified SBIC. Since it allocates memory for a structure needed by all routines in **wd33c93Lib2**, it must be called before any other routines in the library. After calling this routine, at least one call to *wd33c93CtrlInit***( )** must be made before any SCSI transaction is initiated using the SBIC.

NOTE: Only the non-multiplexed processor interface is supported.

A detailed description of the input parameters follows:

sbicBaseAdrs
the address at which the CPU would access the lowest (AUX STATUS) register of the SBIC.

regOffset
the address offset (bytes) to access consecutive registers. (This must be a power of 2, for example, 1, 2, 4, etc.)

clkPeriod
the period in nanoseconds of the signal to SBIC CLK input.

sysScsiBusReset and *sysScsiResetArg*
the board-specific routine to pulse the SCSI bus RST signal. The specified argument is passed to this routine when it is called. It may be used to identify the SCSI bus to be reset, if there is a choice. The interface to this routine is of the form:

```
    void xxBusReset
        (
        int arg;                    /* call-back argument */
        )
```

sysScsiDmaMaxBytes, *sysScsiDmaStart*, *sysScsiDmaAbort*, and *sysScsiDmaArg*
board-specific routines to handle DMA transfers to and from the SBIC; if the maximum DMA byte count is zero, programmed I/O is used. Otherwise, non-NULL function pointers to DMA start and abort routines must be provided. The specified argument is passed to these routines when they are called; it may be used to identify the DMA channel to use, for example. Note that DMA is implemented only during SCSI data in/out phases. The interface to these DMA routines must be of the form:

```
STATUS xxDmaStart
    (
    int arg;                    /* call-back argument           */
    UINT8 *pBuffer;             /* ptr to the data buffer       */
    UINT bufLength;             /* number of bytes to xfer      */
    int direction;              /* 0 = SCSI->mem, 1 = mem->SCSI */
    )
STATUS xxDmaAbort
    (
    int arg;                    /* call-back argument */
    )
```

RETURNS A pointer to the SBIC structure, or NULL if not enough memory or invalid parameters.

SEE ALSO **wd33c93Lib2**

---

*wd33c93CtrlInit***( )**

NAME *wd33c93CtrlInit***( )** – initialize the user-specified fields in an SBIC structure

SYNOPSIS

```
STATUS wd33c93CtrlInit
    (
    int   *pSbic,             /* ptr to SBIC info                        */
    int   scsiCtrlBusId,      /* SCSI bus ID of this SBIC                */
    UINT  defaultSelTimeOut,  /* default dev. select timeout (microsec) */
    int   scsiPriority        /* priority of task when doing SCSI I/O   */
    )
```

DESCRIPTION This routine initializes an SBIC structure, after the structure is created with either *wd33c93CtrlCreate***( )** or *wd33c93CtrlCreateScsi2***( )**. This structure must be initialized before the SBIC can be used. It may be called more than once; however, it should be called only while there is no activity on the SCSI interface.

Before returning, this routine pulses RST (reset) on the SCSI bus, thus resetting all attached devices.

The input parameters are as follows:

pSbic
: a pointer to the **WD_33C93_SCSI_CTRL** structure created with *wd33c93CtrlCreate***( )** or *wd33c93CtrlCreateScsi2***( )**.

scsiCtrlBusId
: the SCSI bus ID of the SBIC, in the range 0 – 7. The ID is somewhat arbitrary; the value 7, or highest priority, is conventional.

defaultSelTimeOut
: the timeout, in microseconds, for selecting a SCSI device attached to this controller. This value is used as a default if no timeout is specified in *scsiPhysDevCreate***( )**. The recommended value zero (0) specifies **SCSI_DEF_SELECT_TIMEOUT** (250 millisec). The maximum timeout possible is approximately 2 seconds. Values exceeding this revert to the maximum. For more information about chip timeouts, see the manuals *Western Digital WD33C92/93 SCSI-Bus Interface Controller*, *Western Digital WD33C92A/93A SCSI-Bus Interface Controller*.

scsiPriority
: the priority to which a task is set when performing a SCSI transaction. Valid priorities are 0 to 255. Alternatively, the value -1 specifies that the priority should not be altered during SCSI transactions.

RETURNS OK, or ERROR if a parameter is out of range.

SEE ALSO **wd33c93Lib**, *scsiPhysDevCreate***( )**, *Western Digital WD33C92/93 SCSI-Bus Interface Controller*, *Western Digital WD33C92A/93A SCSI-Bus Interface Controller*

*wd33c93Show***( )**

NAME *wd33c93Show***( )** – display the values of all readable WD33C93 chip registers

SYNOPSIS
```
int wd33c93Show
    (
    int  *pScsiCtrl  /* ptr to SCSI controller info */
    )
```

DESCRIPTION This routine displays the state of the SBIC registers in a user-friendly manner. It is useful primarily for debugging. It should not be invoked while another running process is accessing the SCSI controller.

EXAMPLE
```
-> wd33c93Show
REG #00 (Own ID         ) = 0x07
REG #01 (Control        ) = 0x00
REG #02 (Timeout Period ) = 0x20
REG #03 (Sectors        ) = 0x00
REG #04 (Heads          ) = 0x00
REG #05 (Cylinders MSB  ) = 0x00
REG #06 (Cylinders LSB  ) = 0x00
REG #07 (Log. Addr. MSB ) = 0x00
REG #08 (Log. Addr. 2SB ) = 0x00
REG #09 (Log. Addr. 3SB ) = 0x00
```

```
REG #0a (Log. Addr. LSB ) = 0x00
REG #0b (Sector Number  ) = 0x00
REG #0c (Head Number    ) = 0x00
REG #0d (Cyl. Number MSB) = 0x00
REG #0e (Cyl. Number LSB) = 0x00
REG #0f (Target LUN     ) = 0x00
REG #10 (Command Phase  ) = 0x00
REG #11 (Synch. Transfer) = 0x00
REG #12 (Xfer Count MSB ) = 0x00
REG #13 (Xfer Count 2SB ) = 0x00
REG #14 (Xfer Count LSB ) = 0x00
REG #15 (Destination ID ) = 0x03
REG #16 (Source ID      ) = 0x00
REG #17 (SCSI Status    ) = 0x42
REG #18 (Command        ) = 0x07
```

RETURNS OK, or ERROR if *pScsiCtrl* and *pSysScsiCtrl* are both NULL.

SEE ALSO **wd33c93Lib**

---

wdbNetromPktDevInit( )

NAME *wdbNetromPktDevInit*( ) – initialize a NETROM packet device for the WDB agent

SYNOPSIS

```
void wdbNetromPktDevInit
    (
    WDB_NETROM_PKT_DEV  *pPktDev,       /* packet device to initialize    */
    caddr_t             dpBase,         /* address of dualport memory     */
    int                 width,          /* number of bytes in a ROM word  */
    int                 index,          /* pod zero's index in a ROM word */
    int                 numAccess,      /* to pod zero per byte read      */
    void                (*stackRcv)(),  /* callback when packet arrives   */
    int                 pollDelay       /* poll task delay                */
    )
```

DESCRIPTION This routine initializes a NETROM packet device. It is typically called from **usrWdb.c** when the WDB agents NETROM communication path is selected. The *dpBase* parameter is the address of NetROM's dualport RAM. The *width* parameter is the width of a word in ROM space, and can be 1, 2, or 4 to select 8-bit, 16-bit, or 32-bit width respectivly (use the macro **WDB_NETROM_WIDTH** in **configAll.h** for this parameter). The *index* parameter refers to which byte of the ROM contains pod zero. The *numAccess* parameter should be set to the number of accesses to POD zero that are required to read a byte. It is typically

one, but some boards actually read a word at a time. This routine spawns a task which polls the NetROM for incomming packets every *pollDelay* clock ticks.

RETURNS N / A

SEE ALSO **wdbNetromPktDrv**

wdbSlipPktDevInit( )

NAME *wdbSlipPktDevInit*( ) – initialize a SLIP packet device for a WDB agent

SYNOPSIS

```
void wdbSlipPktDevInit
    (
    WDB_SLIP_PKT_DEV  *pPktDev,      /* SLIP packetizer device         */
    SIO_CHAN *        pSioChan,      /* underlying serial channel      */
    void              (*stackRcv)()  /* callback when a packet arrives */
    )
```

DESCRIPTION This routine initializes a SLIP packet device on one of the BSP's serial channels. It is typically called from **usrWdb.c** when the WDB agent's lightweight SLIP communication path is selected.

RETURNS N / A

SEE ALSO **wdbSlipPktDrv**

wdbUlipPktDevInit( )

NAME *wdbUlipPktDevInit*( ) – initialize the WDB agent's communication functions for ULIP

SYNOPSIS

```
void wdbUlipPktDevInit
    (
    WDB_ULIP_PKT_DEV * pDev,         /* ULIP packet device to initialize   */
    char *             ulipDev,      /* name of UNIX device to use         */
    void               (*stackRcv)() /* func to call when a packet arrives */
    )
```

DESCRIPTION This routine initializes a ULIP device for use by the WDB debug agent. It provides a communication path to the debug agent which can be used with both a task and an

external mode agent. It is typically called by **usrWdb.c** when the WDB agent's lightweight ULIP communication path is selected.

RETURNS N / A

SEE ALSO **wdbUlipPktDrv**

wdbVioDrv( )

NAME *wdbVioDrv*( ) – initialize the tty driver for a WDB agent

SYNOPSIS

```
STATUS wdbVioDrv
    (
    char  *name
    )
```

DESCRIPTION This routine initializes the VxWorks virtual I / O driver and creates a virtual I / O device of the specified name.

This routine should be called exactly once, before any reads, writes, or opens. Normally, it is called by *usrRoot*( ) in **usrConfig.c**, and the device name created is "/vio".

After this routine has been called, individual virtual I / O channels can be open by appending the channel number to the virtual I / O device name. For example, to get a file descriptor for virtual I / O channel 0x1000017, call *open*( ) as follows:

```
fd = open ("/vio/0x1000017", O_RDWR, 0)
```

RETURNS OK, or ERROR if the driver cannot be installed.

SEE ALSO **wdbVioDrv**

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. This facility is included automatically when **INCLUDE_SHOW_ROUTINES** is defined in **configAll.h**.

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*( )

wim( )

NAME *wim*( ) – return the contents of the window invalid mask register (SPARC)

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

DESCRIPTION This command extracts the contents of the window invalid mask register from the TCB of a specified task. If *taskId* is omitted or 0, the default task is assumed.

RETURNS The contents of the window invalid mask register.

SEE ALSO **dbgArchLib**, *VxWorks Programmer's Guide: Target Shell*

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 could not be logged.

SEE ALSO **wvLib**, **dbgLib**, *e*( ), *wvEvtLogEnable*( )

wvEvtLogDisable( )

NAME *wvEvtLogDisable*( ) – stop event logging (WindView)

SYNOPSIS
```
STATUS wvEvtLogDisable (void)
```

DESCRIPTION This routine turns off all event logging.

If in **CONTINUOUS_MODE** the event buffer is emptied and the communication between the target and host is closed.

This routine has an effect only if **INCLUDE_INSTRUMENTATION** is defined in **configAll.h**, and only if event logging has been started with *wvEvtLogEnable*( ) or from the WindView GUI.

This routine is not callable from interrupt level.

RETURNS OK, or ERROR if called from interrupt level or if event logging is not on.

SEE ALSO **wvLib**, *wvEvtLogEnable*( )

wvEvtLogEnable( )

NAME *wvEvtLogEnable*( ) – start event logging (WindView)

SYNOPSIS
```
STATUS wvEvtLogEnable
    (
    int  eventLevel  /* event logging mode */
    )
```

DESCRIPTION This routine starts event logging at one of three event logging modes. When logging is started via *wvEvtLogEnable*( ), the event buffer is initialized and the connection is established between the target and host.

eventLevel indicates the event logging mode: **CONTEXT_SWITCH**, **TASK_STATE**, or **OBJECT_STATUS**. You cannot switch between these modes while event logging is on; that is, a call to *wvEvtLogEnable*( ) must be followed by a call to *wvEvtLogDisable*( ).

To instrument objects to be logged in the **OBJECT_STATUS** event logging mode, use *wvObjInst*( ) to specify objects and / or *wvSigInst*( ) to instrument signals. When event logging is started, the instrumented objects will generate events. *wvObjInst*( ) can be used at any time to instrument a specified object or set of objects.

This routine has effect only if **INCLUDE_INSTRUMENTATION** is defined in **configAll.h**.

This routine is not callable from interrupt level.

RETURNS OK, or ERROR if called from interrupt level or if instrumentation is not initialized correctly.

SEE ALSO **wvLib**, *wvObjInstModeSet***( )**, *wvObjInst***( )**, *wvEvtLogDisable***( )**, *wvSigInst***( )**

*wvEvtTaskInit***( )**

NAME *wvEvtTaskInit***( )** – set parameters for the event task (WindView)

SYNOPSIS

```
void wvEvtTaskInit
    (
    int  stackSize,  /* event buffer task stack size */
    int  priority    /* event buffer task priority   */
    )
```

DESCRIPTION This routine sets the stack size and priority for the WindView event buffer task, **tEvtTask**. This routine must be called before *wvInstInit***( )**. If the event task's priority is not high enough, then there is a possibility that the event buffer will not be emptied. In this case, event logging will turn itself off.

This routine is usually called from **usrConfig.c**.

RETURN N / A.

SEE ALSO **wvLib**, *wvInstInit***( )**

*wvHostInfoInit***( )**

NAME *wvHostInfoInit***( )** – initialize host connection information (WindView)

SYNOPSIS

```
STATUS wvHostInfoInit
    (
    char *    pIpAddress,  /* host ip address         */
    ushort_t  port         /* host port to connect to */
    )
```

DESCRIPTION This routine initializes the host connection information, and it can be called any time afer the *wvInstInit***( )** call in **usrConfig.c** and before event logging is enabled with *wvEvtLogEnable***( )**.

If the default port number is desired, set *port* to 0 or **DEFAULT_PORT**.

If the *pIpAddress* is NULL then the booting host IP address is the default host address.

RETURN OK, or ERROR if the host address cannot be calculated.

SEE ALSO **wvHostLib**, **wvLib**

---

*wvHostInfoShow***( )**

NAME *wvHostInfoShow***( )** – show host connection information (WindView)

SYNOPSIS
```
STATUS wvHostInfoShow (void)
```

DESCRIPTION This routine prints the WindView host connection information. If this information has not been initialized with *wvHostInfoInit***( )**, a message is printed to that effect.

RETURN OK, or ERROR if the host information has not been initialized.

SEE ALSO **wvHostLib**, *wvHostInfoInit***( )**, **wvLib**

---

*wvInstInit***( )**

NAME *wvInstInit***( )** – initialize instrumentation (WindView)

SYNOPSIS
```
void * wvInstInit
    (
    char *  buffer,      /* pointer to user-supplied buffer */
    size_t  bufferSize,  /* size of event buffer            */
    int     mode         /* event upload mode               */
    )
```

DESCRIPTION This routine initializes kernel instrumentation on the target. It takes a *buffer* address and the size of the buffer (*bufferSize*). If *buffer* is NULL, then a buffer of *bufferSize* is allocated from system memory for the use of the event upload mechanism. If *buffer* is non-NULL, then the specified address is used. This address must be strictly aligned.

A small area for the buffer state variables is carved out of this area. The remainder is used for the event buffer.

mode indicates whether the event upload is to take place in **CONTINUOUS_MODE** or **POST_MORTEM_MODE**. If the event upload mode is set to **CONTINUOUS_MODE**, initialization consists of initializing the event buffer and spawning the event buffer task, **tEvtTask**. As the event buffer is filled, its events are transferred to the host for display by WindView.

If the event upload mode is set to **POST_MORTEM_MODE**, the initialization consists of initializing the event buffer. Events are not transferred to the host; instead, the buffer is overwritten and the contents of the buffer can be used for debugging in the case of a system crash.

When **POST_MORTEM_MODE** is used, the routines in **evtBufferLib** can be used to examine the event buffer. Event logging (using the *wvEvtLogEnable***( )** routine or WindView GUI) will initialize the event buffer in **POST_MORTEM_MODE**, losing previously collected events.

RETURNS Address of the event buffer, or NULL if an error occurs.

SEE ALSO **wvLib**, **evtBufferLib**

*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 event logging is on in object status mode.

objType can be set to one of the following to indicate tasks, semaphores, message queues, or watchdogs: **OBJ_TASK**, **OBJ_SEM**, **OBJ_MSG**, or **OBJ_WD**. *objId* specifies the identifier of the particular object to be instrumented. If *objId* is NULL, then all objects of *objType* have instrumentation turned on or off depending on the value of *mode*.

If *mode* is **INSTRUMENT_ON**, instrumentation is turned on; if it is any other value (including **INSTRUMENT_OFF**) then object *objId* will have instrumentation turned off.

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_INSTRUMENTATION** is defined in **configAll.h**.

RETURNS OK, or ERROR if an error occurs.

SEE ALSO **wvLib**, *wvEvtLogEnable***( )**, *wvSigInst***( )**, *wvObjInstModeSet***( )**

---

*wvObjInstModeSet***( )**

NAME *wvObjInstModeSet***( )** – set mode for object instrumentation (WindView)

SYNOPSIS

```
STATUS wvObjInstModeSet
    (
    int  mode  /* object instrumentation mode */
    )
```

DESCRIPTION This routine causes created object to be either instrumented or not depending on the value of *mode*. *mode* can be **INSTRUMENT_ON** or **INSTRUMENT_OFF**. All objects created after wvObjInstModeSet (**INSTRUMENT_ON**) and before wvObjInstModeSet (**INSTRUMENT_OFF**) will be 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.

The effect of this routine will only be seen if event logging mode is object status event logging mode.

This routine has effect only if **INCLUDE_INSTRUMENTATION** is defined in **configAll.h**.

RETURNS Previous value of *mode* or ERROR if error occurs.

SEE ALSO **wvLib**, *wvEvtLogEnable***( )**, *wvObjInst***( )**, *wvSigInst***( )**

wvOff( )

NAME *wvOff*( ) – stop event logging (WindView)

SYNOPSIS
```
void wvOff (void)
```

DESCRIPTION This command stops WindView event logging. It simply calls *wvEvtLogDisable*( ).

RETURNS N / A.

SEE ALSO **wvLib**, *wvEvtLogDisable*( )

wvOn( )

NAME *wvOn*( ) – start event logging (WindView)

SYNOPSIS
```
void wvOn
    (
    int  eventLevel  /* event logging mode */
    )
```

DESCRIPTION This command starts WindView event logging. It simply calls *wvEvtLogEnable*( ).

RETURNS N / A.

SEE ALSO **wvLib**, *wvEvtLogEnable*( )

wvServerInit( )

NAME *wvServerInit*( ) – start the WindView command server on the target (WindView)

SYNOPSIS
```
STATUS wvServerInit
    (
    int  serverStackSize,  /* server task stack size */
    int  serverPriority    /* server task priority   */
    )
```

DESCRIPTION This routine spawns the RPC server task for WindView with the indicated stack size and priority.

This routine is invoked automatically from **usrConfig.c** when **INCLUDE_WINDVIEW** is defined in **configAll.h**. It should be performed after *usrNetInit***( )** has run.

RETURNS The task ID of the server task, or ERROR if it could not be started.

SEE ALSO **wvLib**

---

*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_INSTRUMENTATION** is defined in **configAll.h** and event logging mode is object status event logging mode.

RETURNS OK, or ERROR if an error occurs.

SEE ALSO **wvLib**, *wvEvtLogEnable***( )**

---

*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:

– timestamp routine, which returns a timestamp when called (must be called with interrupts locked)

– timestamp routine, which returns a timestamp when called (locks interrupts)

– enable-timer routine, which enables the timestamp timer

– disable-timer routine, which disables the timestamp timer

– 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.

– period-of-timer routine, which returns the period of the timer

– frequency-of-timer routine, which returns the frequency of the timer

If any of these routines are set to NULL, the behavior of instrumented code is undefined.

RETURNS N / A

SEE ALSO **wvTmrLib**

---

y( )

NAME *y*( ) – return the contents of the **y** register (SPARC)

SYNOPSIS

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

DESCRIPTION This command extracts the contents of the **y** register from the TCB of a specified task. If *taskId* is omitted or 0, the default task is assumed.

RETURNS The contents of the y register.

SEE ALSO **dbgArchLib**, *VxWorks Programmer's Guide: Target Shell*

z8530DevInit( )

NAME *z8530DevInit*( ) – intialize a **Z8530_DUSART**

SYNOPSIS

```
void z8530DevInit
    (
    Z8530_DUSART *  pDusart
    )
```

DESCRIPTION The BSP must have already initialized all the device addresses, etc in **Z8530_DUSART** structure. This routine initializes some **SIO_CHAN** function pointers and then resets the chip in a quiescent state.

SEE ALSO **z8530Sio**

z8530Int( )

NAME *z8530Int*( ) – handle all interrupts in one vector

SYNOPSIS

```
void z8530Int
    (
    Z8530_DUSART *  pDusart
    )
```

DESCRIPTION On some boards, all SCC interrupts for both ports share a single interrupt vector. This is the ISR for such boards. We determine from the parameter which SCC interrupted, then look at the code to find out which channel and what kind of interrupt.

SEE ALSO **z8530Sio**

z8530IntEx( )

NAME *z8530IntEx*( ) – handle error interrupts

SYNOPSIS
```
void z8530IntEx
    (
    Z8530_CHAN *  pChan
    )
```

DESCRIPTION This routine handles miscellaneous interrupts on the SCC

RETURNS N / A

SEE ALSO **z8530Sio**

z8530IntRd( )

NAME *z8530IntRd*( ) – handle a reciever interrupt

SYNOPSIS
```
void z8530IntRd
    (
    Z8530_CHAN *  pChan
    )
```

DESCRIPTION This routine handles read interrupts from the SCC

RETURNS N / A

SEE ALSO **z8530Sio**

z8530IntWr( )

NAME *z8530IntWr*( ) – handle a transmitter interrupt

SYNOPSIS

```
void z8530IntWr
    (
    Z8530_CHAN *  pChan
    )
```

DESCRIPTION This routine handles write interrupts from the SCC.

RETURNS N / A

SEE ALSO **z8530Sio**

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.

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.

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).

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.

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.

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 */
    )
```

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.

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*.

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.

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 entry for a discussion of addressing zbufs by segment and offset.

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.

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**

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 previous to the zbuf segment *zbufSeg*. If *zbufSeg* is NULL, or is the first segment in *zbufId*, NULL is returned.

RETURNS The zbuf segment ID of the segment previous to *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.

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 */
    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.

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 in **configAll.h**.

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.

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.

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.

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 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.

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.

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*.

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

Wind River Systems
Corporate Headquarters
1010 Atlantic Avenue
Alameda, CA 94501-1153
USA
800/545-WIND toll free
510/748-4100 phone
510/814-2010 fax

Wind River Systems, S.A.R.L.
19, Avenue de Norvège
Immeuble B4, Bâtiment 3
Z.A. de Courtaboeuf 1
91953 Les Ulis Cédex
FRANCE
33-1-60-92-63-00 phone
33-1-60-92-63-15 fax

Wind River Systems GmbH
Freisinger Straße 34
Postfach 1320
D-85737 Ismaning
GERMANY
49-89-96-24-45-0 phone
49-89-96-24-45-55 fax

Wind River Systems UK Ltd
Unit 5, Ashted Lock
Aston Science Park
Birmingham B7 4AZ
UNITED KINGDOM
44-121-628-1888 phone
44-121-628-1889 fax

Wind River Systems Scandinavia
Turebergs Torg 1
S-191 47 Sollentuna
SWEDEN
46-8 92 15 80 phone
46-8 92 15 65 fax

Wind River Systems
Japan/Asia-Pacific
Pola Ebisu Bldg. 11F
3-9-19 Higashi
Shibuya-ku
Tokyo 150
JAPAN
81-3-5467-5900 phone
81-3-5467-5877 fax

Copyright © 1998 Wind River Systems, Inc. All rights reserved.
Printed in U.S.A.

DOC-12068-ZD-00