VxWorks®

Reference Manual

5.4

Edition 1

WindRiver®
SYSTEMS

An ISO 9001 Registered Company

Corporate Headquarters
Wind River Systems, Inc.
500 Wind River Way
Alameda, CA 94501-1153
USA

toll free (US): 800/545-WIND
telephone: 510/748-4100
facsimile: 510/749-2010

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 K.K.
Ebisu Prime Square Tower 5th Fl.
1-1-39 Hiroo
Shibuya-ku
Tokyo 150-0012
JAPAN

telephone: 81-3-5778-6001
facsimile: 81-3-5778-6002

CUSTOMER SUPPORT

	Telephone	E-mail	Fax
Corporate:	800/872-4977 toll free, U.S. & Canada 510/748-4100 direct	support@wrs.com	510/749-2164
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, 5.4
Edition 1
7 May 99
Part #: DOC-12910-ND-00

Contents

1 Libraries

This section provides reference pages for VxWorks libraries. Each entry lists the routines found in the library, including a one-line synopsis of each and a general description of their use.

Entries for libraries that are specific to board support packages (BSPs) are provided in online format only. However, this section contains entries for the serial, Ethernet, and SCSI drivers available with VxWorks BSPs, plus a generic entry for the BSP-specific library **sysLib**.

2 Subroutines

This section provides reference pages for each of the subroutines found in VxWorks libraries documented in section 1.

Keyword Index

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

1
Libraries

aic7880Lib

NAME

aic7880Lib – Adaptec 7880 SCSI Host Adapter Library File

ROUTINES

***aic7880CtrlCreate*( )** – create a control structure for the AIC 7880
***aic7880ScbCompleted*( )** – successfully completed execution of a client thread
***aic7880EnableFast20*( )** – enable double speed SCSI data transfers
***aic7880dFifoThresholdSet*( )** – set the data FIFO threshold.
***aic7880GetNumOfBuses*( )** – perform a PCI bus scan
***aic7880ReadConfig*( )** – read from PCI config space
***aic7880WriteConfig*( )** – read to PCI config space

DESCRIPTION

This is the I/O driver for the Adaptec AIC 7880 PCI Bus Master Single Chip SCSI Host Adapter. It is designed to work with **scsi2Lib**. This driver runs in conjunction with the HIM (Hardware Interface Module) supplied by Adaptec. The AIC 7880 SCSI Host Adapter driver supports the following features:

– Fast, Double Speed 20 MHz data transfers.
– 16 bit Wide Synchronous Data transfers.
– Tagged Command Queueing.
– Data FIFO threshold selection.
– Disconnect / Reconnect support.
– Multiple Initiator support.
– Multiple Controller support.

In general, the SCSI system and this driver will automatically choose the best combination of these features to suit the target devices used. However, the default choices may be over-ridden by using the function ***scsiTargetOptionsSet*( )** (see **scsiLib**).

OPERATIONS OVERVIEW

The host processor initiates a SCSI I/O operation by programming a data structure called SCB (SCSI Command Block). The SCB contains all the relevant information needed by the Host Adapter to carry out the requested SCSI operation. SCSI SCB's are passed to the HIM by this module which are then sent to the AIC-7880 for execution. The AIC-7880 Sequencer or PhaseEngine comprises the on-chip intelligence that allows the AIC-7880 to execute SCB commands. The Sequencer is programmable and uses its own microcode program which is downloaded to AIC-7880 by the host at initialization.

The following is an example of how an SCB is delivered to the AIC-7880

– Memory is allocated for the SCB structure and it is programmed with the necessary information required to execute a SCSI transaction.

– The SCB is then sent to HIM.

– The HIM pauses the Sequencer.

– The Sequencer has internal registers that point to the area in system memory where the SCB resides.

– The HIM unpauses the Sequencer.

– The AIC-7880 Sequencer uses DMA to transfer the SCB into its internal memory.

– The AIC-7880 executes the SCB.

– Upon completion of the SCB command, the AIC-7880 Sequencer posts the pointer of the completed SCB into system memory.

– The AIC-7880 generates an interupt.

– The status of the completed SCB is then read by the host.

SCB PROCESSING The AIC-7880 Sequencer uses DMA to transfer the SCB into its internal memory. The Sequencer processes SCB's in the order they are received with new SCB's being started when older SCB operations are idle due to wait for selection or a SCSI bus disconnect. When operations for an Idle SCB reactivate, the sequencer scans the SCB array for the SCB corresponding to the Target/LUN reactivating. The Sequencer then restarts the SCB found until the next disconnect or SCB completion.

MAXIMUM NUMBER OF TAGGED SCB's

The number of tagged SCB's per SCSI target that is handled by the Sequencer, range from 1-32. The HIM supports only the External SCB Access mode. The default number of tags handled by the Sequencer in this mode is 32. Changing the field "Cf_MaxTagScbs" in the cfp_struct changes the maximum number of tagged SCB's.

MAXIMUM NUMBER OF SCB's

The number of SCB's that can be queued to the Sequencer, range from 1-254. This value can be changed before calling the HIM routine "PH_GetConfig ()". Changing the field "Cf_NumberScbs" in "cfp_struct" changes the maximum number of SCB's to be used. The default max number of SCB's is 254.

SYNCHRONOUS TRANSFER SUPPORT

If double speed SCSI mode is enabled, this driver supports transfer periods of 50, 64 and 76 ns. In standard fast SCSI mode transfer periods of 100, 125, 150, 175, 200, 225, 250 and 275 are supported. Synchronous transfer parameters for a target can be set using the SCSI library function "scsiTargetOptionsSet".

DOUBLE SPEED SCSI MODE

To enable/disable double speed SCSI mode the routine "aic7880EnableFast20" needs to be invoked with the following two parameters:

(1) A pointer to the appropriate SCSI Controller structure

(2) A BOOLEAN value which enables or disable double speed SCSI mode.

With double speed SCSI mode enabled the host adapter may be capable of transferring data at theoritcal transfer rates of 20 MB/s for an 8-bit device and 40 MB/s for a 16-bit device. Double Speed SCSI is disabled by default.

DATA FIFO THRESHOLD

To set the data FIFO threshold the routine "aic7880dFifoThresholdSet" needs to be invoked with the following two parameters:

(1) A pointer to the appropriate SCSI Controller structure

(2) The data FIFO threhold value.

For more information about the data FIFO threshold value refer the ***aic7880dFifoThresholdSet*( )** routine

In order to initialize the driver from the BSP the following needs to be done in the BSP specific routine ***sysScsiInit*( )** in file **sysScsi.c**.

– Find the SCSI Host Adapter.
– Create the SCSI Controller Structure.
– Connect the interrupt to Interupt Service Routine (ISR).
– Enable the SCSI interupt

The following example shows the SCSI initialization sequence that need to be done in the BSP.

```
STATUS sysScsiInit ()
    {
    int   busNo;          /* PCI bus number          */
    int   devNo;          /* PCI device number       */
    UWORD found = FALSE;  /* host adapter found      */
    int   numHa = 0;      /* number of host adapters */
    for (busNo=0; busNo < MAX_NO_OF_PCI_BUSES && !found; busNo++)
        for (devNo = 0; devNo < MAX_NO_OF_PCI_DEVICES; devNo++)
        {
        if ((found = sysScsiHostAdapterFind (busNo, devNo)) == HA_FOUND)
            {
            numHa++;
            /* Create the SCSI controller */
            if ((pSysScsiCtrl = (SCSI_CTRL *) aic7880CtrlCreate
                 (busNo, devNo, SCSI_DEF_CTRL_BUS_ID)) == NULL)
                {
                logMsg ("Could not create SCSI controller\n",
                         0, 0, 0, 0, 0, 0);
                return (ERROR);
                }
            /* connect the SCSI controller's interrupt service routine */
            if ((intConnect (INUM_TO_IVEC (SCSI_INT_VEC), aic7880Intr,
                                 (int) pSysScsiCtrl)) == ERROR)
```

```
              return (ERROR);
          /* enable SCSI interupts */
          sysIntEnablePIC (SCSI_INT_LVL);
          }
    return (OK);
    }
```

SEE ALSO
scsiLib, **scsi2Lib**, **cacheLib**, *AIC-7880 Design In Handbook, AIC-7880 Data Book, Adaptec Hardware Interface Module (HIM) Specification, VxWorks Programmer's Guide: I/O System*

aioPxLib

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

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

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

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

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

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

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

***aio_read*( )**
initiates an asynchronous read

***aio_write*( )**
initiates an asynchronous write

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

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

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

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

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

AIO CONTROL BLOCK

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

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

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

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

```
struct
    {
    int                 aio_fildes;
    off_t               aio_offset;
    volatile void *     aio_buf;
    size_t              aio_nbytes;
    int                 aio_reqprio;
```

```
struct sigevent     aio_sigevent;
int                 aio_lio_opcode;
AIO_SYS             aio_sys;
} aiocb
```

aio_fildes
file descriptor for I/O.

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

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

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

aioSysDrv

NAME **aioSysDrv** – AIO system driver

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

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

SEE ALSO POSIX 1003.1b document

ambaSio

NAME **ambaSio** – ARM AMBA UART tty driver

ROUTINES ***ambaDevInit*( )** – initialise an AMBA channel
***ambaIntTx*( )** – handle a transmitter interrupt
***ambaIntRx*( )** – handle a receiver interrupt

DESCRIPTION This is the device driver for the Advanced RISC Machines (ARM) AMBA UART. This is a generic design of UART used within a number of chips containing (or for use with) ARM CPUs such as in the Digital Semiconductor 21285 chip as used in the EBSA-285 BSP.

This design contains a universal asynchronous receiver/transmitter, a baud-rate generator, and an InfraRed Data Association (IrDa) Serial InfraRed (SiR) protocol encoder. The Sir encoder is not supported by this driver. The UART contains two 16-entry deep FIFOs for receive and transmit: if a framing, overrun or parity error occurs during reception, the appropriate error bits are stored in the receive FIFO along with the received data. The FIFOs can be programmed to be one byte deep only, like a conventional UART with double buffering, but the only mode of operation supported is with the FIFOs enabled.

The UART design does not support the modem control output signals: DTR, RI and RTS. Moreover, the implementation in the 21285 chip does not support the modem control inputs: DCD, CTS and DSR.

The UART design can generate four interrupts: Rx, Tx, modem status change and a UART disabled interrupt (which is asserted when a start bit is detected on the receive line when the UART is disabled). The implementation in the 21285 chip has only two interrupts: Rx and Tx, but the Rx interrupt is a combination of the normal Rx interrupt status and the UART disabled interrupt status.

Only asynchronous serial operation is supported by the UART which supports 5 to 8 bit bit word lengths with or without parity and with one or two stop bits. The only serial word format supported by the driver is 8 data bits, 1 stop bit, no parity, The default baud rate is determined by the BSP by filling in the **AMBA_CHAN** structure before calling ***ambaDevInit*( )**.

The exact baud rates supported by this driver will depend on the crystal fitted (and consequently the input clock to the baud-rate generator), but in general, baud rates from about 300 to about 115200 are possible.

In theory, any number of UART channels could be implemented within a chip. This driver has been designed to cope with an arbitrary number of channels, but at the time of writing, has only ever been tested with one channel.

DATA STRUCTURES

An **AMBA_CHAN** data structure is used to describe each channel, this structure is described in **h/drv/sio/ambaSio.h**.

CALLBACKS

Servicing a "transmitter ready" interrupt involves making a callback to a higher level library in order to get a character to transmit. By default, this driver installs dummy callback routines which do nothing. A higher layer library that wants to use this driver (e.g. **ttyDrv**) will install its own callback routine using the **SIO_INSTALL_CALLBACK** ioctl command. Likewise, a receiver interrupt handler makes a callback to pass the character to the higher layer library.

MODES

This driver supports both polled and interrupt modes.

USAGE

The driver is typically only called by the BSP. The directly callable routines in this modules are ***ambaDevInit*( )**, ***ambaIntTx*( )** and ***ambaIntRx*( )**.

The BSP's ***sysHwInit*( )** routine typically calls ***sysSerialHwInit*( )**, which initialises the hardware-specific fields in the **AMBA_CHAN** structure (e.g. register I/O addresses etc) before calling ***ambaDevInit*( )** which resets the device and installs the driver function pointers. After this the UART will be enabled and ready to generate interrupts, but those interrupts will be disabled in the interrupt controller.

The following example shows the first parts of the initialisation:

```
#include "drv/sio/ambaSio.h"
LOCAL AMBA_CHAN ambaChan[N_AMBA_UART_CHANS];
void sysSerialHwInit (void)
    {
    int i;
    for (i = 0; i < N_AMBA_UART_CHANS; i++)
        {
        ambaChan[i].regs = devParas[i].baseAdrs;
        ambaChan[i].baudRate = CONSOLE_BAUD_RATE;
        ambaChan[i].xtal = UART_XTAL_FREQ;
```

```
        ambaChan[i].levelRx = devParas[i].intLevelRx;
        ambaChan[i].levelTx = devParas[i].intLevelTx;
        /*
         * Initialise driver functions, getTxChar, putRcvChar and
         * channelMode, then initialise UART
         */
        ambaDevInit(&ambaChan[i]);
        }
    }
```

The BSP's ***sysHwInit2( )*** routine typically calls ***sysSerialHwInit2( )***, which connects the chips interrupts via ***intConnect( )*** (the two interrupts **ambaIntTx** and **ambaIntRx**) and enables those interrupts, as shown in the following example:

```
void sysSerialHwInit2 (void)
    {
    /* connect and enable Rx interrupt */
    (void) intConnect (INUM_TO_IVEC(devParas[0].vectorRx),
                       ambaIntRx, (int) &ambaChan[0]);
    intEnable (devParas[0].intLevelRx);
    /* connect Tx interrupt */
    (void) intConnect (INUM_TO_IVEC(devParas[0].vectorTx),
                       ambaIntTx, (int) &ambaChan[0]);
    /*
     * There is no point in enabling the Tx interrupt, as it will
     * interrupt immediately and then be disabled.
     */
    }
```

BSP

By convention all the BSP-specific serial initialisation is performed in a file called **sysSerial.c**, which is #include'ed by **sysLib.c**. **sysSerial.c** implements at least four functions, ***sysSerialHwInit( )***, ***sysSerialHwInit2( )***, ***sysSerialChanGet( )***, and ***sysSerialReset( )***. The first two have been described above, the others work as follows:

sysSerialChanGet( ) is called by usrRoot to get the serial channel descriptor associated with a serial channel number. The routine takes a single parameter which is a channel number ranging between zero and **NUM_TTY**. It returns a pointer to the corresponding channel descriptor, **SIO_CHAN** *, which is just the address of the **AMBA_CHAN** structure.

sysSerialReset( ) is called from ***sysToMonitor( )*** and should reset the serial devices to an inactive state (prevent them from generating any interrupts).

INCLUDE FILES

drv/sio/ambaSio.h sioLib.h

SEE ALSO

Advanced RISC Machines AMBA UART (AP13) Data Sheet, Digital Semiconductor 21285 Core Logic for SA-110 Microprocessor Data Sheet, " *Digital Semiconductor EBSA-285 Evaluation Board Reference Manual.*

ansiAssert

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

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

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

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

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

INCLUDE FILES **stdio.h**, **stdlib.h**, **assert.h**

SEE ALSO American National Standard X3.159-1989

ansiCtype

NAME **ansiCtype** – ANSI **ctype** documentation

ROUTINES

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

DESCRIPTION The header **ctype.h** declares several functions useful for testing and mapping characters. In all cases, the argument is an **int**, the value of which is representable as an **unsigned**

char or is equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined.

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

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

INCLUDE FILES **ctype.h**

SEE ALSO American National Standard X3.159-1989

ansiLocale

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

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

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

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

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

ansiMath

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

ROUTINES
***asin*( )** – compute an arc sine (ANSI)
***acos*( )** – compute an arc cosine (ANSI)
***atan*( )** – compute an arc tangent (ANSI)
***atan2*( )** – compute the arc tangent of y/x (ANSI)
***ceil*( )** – compute the smallest integer greater than or equal to a specified value (ANSI)

***cosh*( )** – compute a hyperbolic cosine (ANSI)
***exp*( )** – compute an exponential value (ANSI)
***fabs*( )** – compute an absolute value (ANSI)
***floor*( )** – compute the largest integer less than or equal to a specified value (ANSI)
***fmod*( )** – compute the remainder of x/y (ANSI)
***frexp*( )** – break a floating-point number into a normalized fraction and power of 2 (ANSI)
***ldexp*( )** – multiply a number by an integral power of 2 (ANSI)
***log*( )** – compute a natural logarithm (ANSI)
***log10*( )** – compute a base-10 logarithm (ANSI)
***modf*( )** – separate a floating-point number into integer and fraction parts (ANSI)
***pow*( )** – compute the value of a number raised to a specified power (ANSI)
***sin*( )** – compute a sine (ANSI)
***cos*( )** – compute a cosine (ANSI)
***sinh*( )** – compute a hyperbolic sine (ANSI)
***sqrt*( )** – compute a non-negative square root (ANSI)
***tan*( )** – compute a tangent (ANSI)
***tanh*( )** – compute a hyperbolic tangent (ANSI)

DESCRIPTION

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

The macro defined is:

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

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

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

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

INCLUDE FILES

math.h

SEE ALSO

mathALib, American National Standard X3.159-1989

ansiSetjmp

NAME **ansiSetjmp** – ANSI **setjmp** documentation

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

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

The type declared is:

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

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

SEE ALSO American National Standard X3.159-1989

ansiStdarg

NAME **ansiStdarg** – ANSI **stdarg** documentation

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

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

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

The type declared is:

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

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

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

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

SEE ALSO American National Standard X3.159-1989

ansiStdio

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

ROUTINES

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

***perror*( )** – map an error number in **errno** to an error message (ANSI)
***putc*( )** – write a character to a stream (ANSI)
***putchar*( )** – write a character to the standard output stream (ANSI)
***puts*( )** – write a string to the standard output stream (ANSI)
***putw*( )** – write a word (32-bit integer) to a stream
***rewind*( )** – set the file position indicator to the beginning of a file (ANSI)
***scanf*( )** – read and convert characters from the standard input stream (ANSI)
***setbuf*( )** – specify the buffering for a stream (ANSI)
***setbuffer*( )** – specify buffering for a stream
***setlinebuf*( )** – set line buffering for standard output or standard error
***setvbuf*( )** – specify buffering for a stream (ANSI)
***stdioInit*( )** – initialize standard I/O support
***stdioFp*( )** – return the standard input/output/error FILE of the current task
***stdioShowInit*( )** – initialize the standard I/O show facility
***stdioShow*( )** – display file pointer internals
***tmpfile*( )** – create a temporary binary file (Unimplemented) (ANSI)
***tmpnam*( )** – generate a temporary file name (ANSI)
***ungetc*( )** – push a character back into an input stream (ANSI)
***vfprintf*( )** – write a formatted string to a stream (ANSI)

DESCRIPTION

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

Types

The types declared are **size_t** and:

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

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

Macros

The macros are NULL and:

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

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

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

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

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

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

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

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

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

STREAMS

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

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

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

Environmental Limits

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

FILES

A stream is associated with an external file (which may be a physical device) by opening a file, which may involve creating a new file. Creating an existing file causes its former contents to be discarded, if necessary. If a file can support positioning requests (such as a disk file, as opposed to a terminal), then a file position indicator associated with the stream is positioned at the start (character number zero) of the file. The file position indicator is maintained by subsequent reads, writes, and positioning requests, to facilitate an orderly progression through the file. All input takes place as if characters were read by successive calls to ***fgetc*( )**; all output takes place as if characters were written by successive calls to ***fputc*( )**.

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

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

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

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

TASK TERMINATION

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

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

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

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

FIOLIB

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

SEE ALSO

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

ansiStdlib

NAME

ansiStdlib – ANSI **stdlib** documentation

ROUTINES

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

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

DESCRIPTION

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

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

Types

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

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

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

Macros

The macros defined are NULL and:

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

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

INCLUDE FILES

stdlib.h

SEE ALSO

American National Standard X3.159-1989

---

ansiString

NAME

ansiString – ANSI **string** documentation

ROUTINES

***memchr*( )** – search a block of memory for a character (ANSI)
***memcmp*( )** – compare two blocks of memory (ANSI)
***memcpy*( )** – copy memory from one location to another (ANSI)
***memmove*( )** – copy memory from one location to another (ANSI)
***memset*( )** – set a block of memory (ANSI)
***strcat*( )** – concatenate one string to another (ANSI)
***strchr*( )** – find the first occurrence of a character in a string (ANSI)

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

DESCRIPTION

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

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

SEE ALSO

American National Standard X3.159-1989

ansiTime

NAME

ansiTime – ANSI **time** documentation

ROUTINES

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

***localtime*( )** – convert calendar time into broken-down time (ANSI)
***localtime_r*( )** – convert calendar time into broken-down time (POSIX)
***mktime*( )** – convert broken-down time into calendar time (ANSI)
***strftime*( )** – convert broken-down time into a formatted string (ANSI)
***time*( )** – determine the current calendar time (ANSI)

DESCRIPTION

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

Macros

The macros defined are NULL and:

CLOCKS_PER_SEC
the number of ticks per second.

Types

The types declared are **size_t** and:

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

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

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

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

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

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

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

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

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

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

INCLUDE FILES
time.h

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

arpLib

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

ROUTINES
***arpAdd*( )** – 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

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. Use ***arpDelete*( )** to delete entries from the ARP table. Use ***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

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

DESCRIPTION

This is a driver for ATA/IDE devices on PCMCIA, ISA, and other buses. The driver can be customized via various macros to run on a variety of boards and both big-endian, and little endian CPUs.

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 or VxWorks initialization. Thus, the BIOS setting should match the modes specified by the configuration flag.

The BSP may provide a ***sysAtaInit*( )** routine for situations where an ATA controller RESET (0x1f6 or 0x3f6, bit 2 is set) clears ATA specific functionality in a chipset that is not re-enabled per the ATA-2 spec.

This BSP routine should be declared in **sysLib.c** or **sysAta.c** as follows:

```
void sysAtaInit (BOOL ctrl)
    {
    /* BSP SPECIFIC CODE HERE */
    }
```

Then the BSP should perform the following operation before ***ataDrv*( )** is called, in sysHwInit for example:

```
IMPORT VOIDFUNCPTR _func_sysAtaInit;
/* setup during initialization */
_func_sysAtaInit = (VOIDFUNCPTR) sysAtaInit;
```

It should contain chipset specific reset code, such as code which re-enables PCI write posting for an integrated PCI-IDE device, for example. This will be executed during every

***ataDrv*( )**, ***ataInit*( )**, and ***ataReset*( )** or equivalent block device routine. If the sysAtaInit routine is not provided by the BSP it is ignored by the driver, therefore it is not a required BSP routine.

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

ataShow

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

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

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

bALib

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

ROUTINES ***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)

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

ROUTINES

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

DESCRIPTION

This library contains routines to manipulate buffers of variable-length byte arrays. Operations are performed on long words when possible, even though the buffer lengths are specified in bytes. This occurs only when source and destination buffers start on addresses that are both odd or both even. If one buffer is even and the other is odd, operations must be done one byte at a time (because of alignment problems inherent in the MC68000), thereby slowing down the process.

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

INCLUDE FILES **string.h**

SEE ALSO **ansiString**

bootConfig

NAME **bootConfig** – system configuration module for boot ROMs

ROUTINES No Callable Routines

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

bootInit

NAME **bootInit** – ROM initialization module

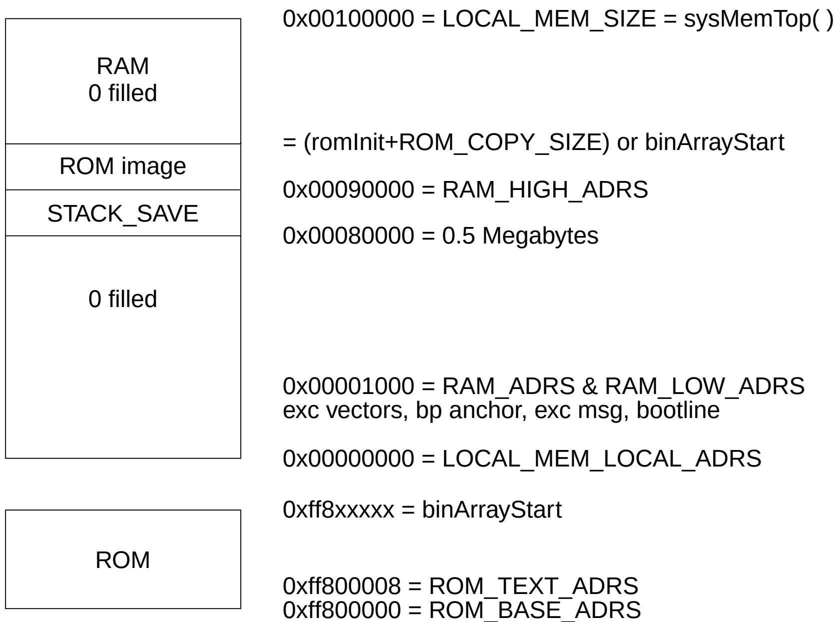
ROUTINES ***romStart*( )** – generic ROM initialization

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

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

ROM AND RAM MEMORY LAYOUT

Example memory layout for a 1-megabyte board:



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

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

bootLib

NAME **bootLib** – boot ROM subroutine library

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

DESCRIPTION

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

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

The boot line is of the form:

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

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

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

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

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

filename
: the file to be booted (required).

e the Internet address of the Ethernet interface. This field can have an optional subnet mask of the form *inet_adrs*:*subnet_mask*. If DHCP is used to obtain the configuration parameters, lease timing information may also be present. This information takes the form *lease_duration*:*lease_origin* and is appended to the end of the field. (see ***bootNetmaskExtract( )*** and ***bootLeaseExtract( )***).

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

h the Internet address of the boot host.

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

u a valid user name on the boot host.

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

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

tn the name of the system being booted

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

o "other" string for use by the application.

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

EXAMPLE

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

INCLUDE FILES **bootLib.h**

SEE ALSO **bootConfig**

bootpLib

NAME **bootpLib** – BOOTP client library

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

DESCRIPTION This library implements the client side of the Bootstrap Protocol (BOOTP). This network protocol allows the dynamic configuration of the target's boot parameters at boot time. This is in contrast to using the boot information encoded in system non-volatile RAM or ROM. Thus, at boot time, BOOTP goes over the network to get an IP address, a boot file name, and the boot host's IP address.

The actual transfer of the boot image is handled by a file transfer protocol, such as TFTP or FTP, or by an RSH command.

To access BOOTP services, you can use either the high-level interface supported by ***bootpParamsGet*( )**, or the low-level interface supported by ***bootpMsgSend*( )**.

HIGH-LEVEL INTERFACE

The ***bootpParamsGet*( )** routine provides the highest level interface to BOOTP. It accepts a parameter descriptor structure that allows the retrieval of any combination of the options described in RFC 1533 (if supported by the BOOTP server and if specified in the database). During system boot, the routine obtains the boot file, the Internet address, and the host Internet address. It also obtains the subnet mask and the Internet address of an IP router, if available.

LOW-LEVEL INTERFACE

The ***bootpMsgSend*( )** routine provides a lower-level interface to BOOTP. It accepts and returns a BOOTP message as a parameter. This interface is more flexible because it gives the caller direct access to the data in the BOOTP request/reply messages. For example, if the BOOTP message includes implementation-specific options not defined in an RFC, the caller can use ***bootpMsgSend*( )** to retrieve them from the vendor-specific field in the BOOTP message. The ***bootpParamsGet*( )** routine already provides all defined options.

EXAMPLE

The following code provides and example of how to use ***bootpParamsGet*( )**:

```
#include "bootpLib.h"
struct bootpParams  bootParams;
struct in_addr      clntAddr;
struct in_addr      hostAddr;
char                bootFile [FILENAME_SIZE];
int                 subnetMask;
struct in_addr_list routerList;
struct in_addr      gateway;
char        clntAddr [INET_ADDR_LEN];
char        bootServer [INET_ADDR_LEN];
char        bootFile [SIZE_FILE];
int         fileSize;
int         subnetMask;
char        gateway [INET_ADDR_LEN];

bzero ( (char *)&clntAddr, sizeof (struct in_addr));
bzero ( (char *)&hostAddr, sizeof (struct in_addr));
bzero (bootFile, FILENAME_SIZE);
subnetMask  = 0;
bzero ( (char *)&gateway, sizeof (struct in_addr));
/* Set all pointers in parameter descriptor to NULL. */
bzero ((char *)&bootParams, sizeof (struct bootpParams));
/* Set pointers corresponding to desired options. */
bootParams.clientAddr = &clntAddr;
bootParams.bootHostAddr = &hostAddr;
bootParams.bootfile = pBootFile;
bootParams.netmask = (struct in_addr *)&subnetMask;
routerlist.addr = &gateway;
routerlist.num = 1;
bootParams.routers = &routerlist;
if (bootpParamsGet ("ln0", 0, 0, &bootParams) == ERROR)
    return (ERROR);
```

NOTE

Certain targets (typically those with no NVRAM) construct their Ethernet address based on the target's IP address. An IP address must be entered for these targets in order to boot over the network. The remaining information can be obtained with BOOTP.

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

INCLUDE FILES **bootpLib.h**

SEE ALSO **bootLib**, RFC 951, RFC 1542, RFC 1533, *VxWorks Programmer's Guide: Network*

cacheArchLib

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

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

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

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

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

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

cacheCy604Lib

NAME

cacheCy604Lib – Cypress CY7C604/605 SPARC cache management library

ROUTINES

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

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

ROUTINES

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)

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

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

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

ROUTINES
***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)

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.

1

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

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

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

ROUTINES
***cacheLibInit*( )** – initialize the cache library for a processor architecture
***cacheEnable*( )** – enable the specified cache
***cacheDisable*( )** – disable the specified cache
***cacheLock*( )** – lock all or part of a specified cache
***cacheUnlock*( )** – unlock all or part of a specified cache
***cacheFlush*( )** – flush all or some of a specified cache
***cacheInvalidate*( )** – invalidate all or some of a specified cache
***cacheClear*( )** – clear all or some entries from a cache
***cachePipeFlush*( )** – flush processor write buffers to memory

***cacheTextUpdate*( )** – synchronize the instruction and data caches
***cacheDmaMalloc*( )** – allocate a cache-safe buffer for DMA devices and drivers
***cacheDmaFree*( )** – free the buffer acquired with ***cacheDmaMalloc*( )**
***cacheDrvFlush*( )** – flush the data cache for drivers
***cacheDrvInvalidate*( )** – invalidate data cache for drivers
***cacheDrvVirtToPhys*( )** – translate a virtual address for drivers
***cacheDrvPhysToVirt*( )** – translate a physical address for drivers

DESCRIPTION

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

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

1

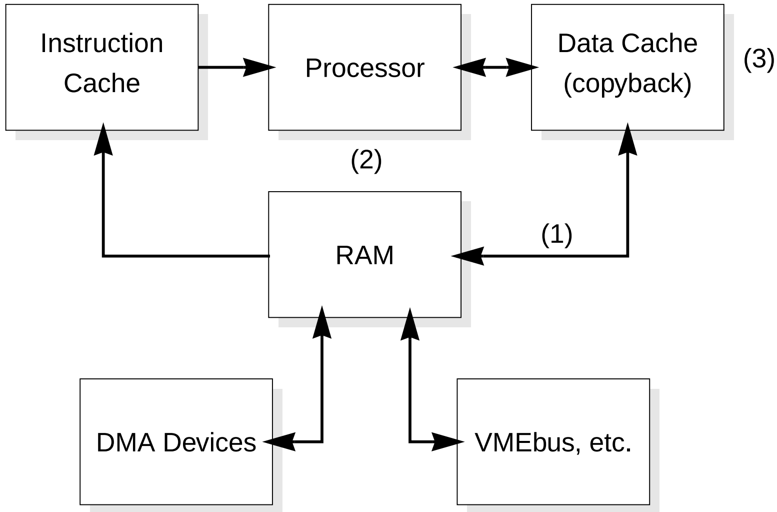
***cacheDmaMalloc*( )** routine attempts to return a "cache-safe" buffer, which is created by the MMU and a set of flush and invalidate function pointers. Examples are provided below in the section "Using the Cache Library."

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

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

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

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

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**

cacheR3kALib

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

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

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

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

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

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

INCLUDE FILES **cacheLib.h**

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

---

cacheR4kLib

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

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

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

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

INCLUDE FILES **cacheLib.h**

SEE ALSO **cacheLib**

---

cacheR33kLib

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

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

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

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

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

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

cacheR333x0Lib

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

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

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

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

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

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

cacheSun4Lib

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

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

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

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

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

INCLUDE FILES **cacheLib.h**

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

cacheTiTms390Lib

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

ROUTINES
***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

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.

1

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 may 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

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

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**

cdromFsLib

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

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

DESCRIPTION

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

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

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

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

```
cdromFsInit ();
```

2. Locate and create a SCSI physical device:

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

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

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

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

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

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

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

FILE AND DIRECTORY NAMING

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

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

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

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

IOCTL CODES SUPPORTED

FIOGETNAME
Returns the file name for a specific file descriptor.

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

FIOWHERE
Determines the current file position.

FIOSEEK
Changes the current file position.

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

FIOREADDIR
Reads the next directory entry.

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

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

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

MODIFYING A BSP TO USE CDROMFS

The following example describes mounting cdromFS on a SCSI device.

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

1. Insert the following macro definition:

```
#define INCLUDE_CDROMFS
```

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

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

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

1. Add the following declaration to the top of the file:

```
#ifdef INCLUDE_CDROMFS
#include "cdromFsLib.h"
STATUS cdromFsInit (void);
#endif
```

2. Modify the definition of ***sysScsiInit*( )** to include the following:

```
#ifdef INCLUDE_CDROMFS
cdromFsInit();
#endif
```

The call to ***cdromFsInit*( )** initializes cdromFS. This call must be made only once and must complete successfully before you can call any other **cdromFsLib** routines, such as ***cdromFsDevCreate*( )**. Typically, you make the ***cdromFSInit*( )** call at system startup. Because cdromFS is used with SCSI CD-ROM devices, it is natural to call ***cdromFSInit*( )** from within ***sysScsiInit*( )**.

3. Modify the definition of ***sysScsiConfig*( )** (if included in your BSP) to include the following:

```
/* configure a SCSI CDROM at busId 6, LUN = 0 */
#ifdef INCLUDE_CDROMFS
if ((pSpd60 = scsiPhysDevCreate (pSysScsiCtrl, 6, 0, 0, NONE, 0, 0, 0)) ==
    (SCSI_PHYS_DEV *) NULL)
    {
    SCSI_DEBUG_MSG ("sysScsiConfig: scsiPhysDevCreate failed for CDROM.\n",
                    0, 0, 0, 0, 0, 0);
    return (ERROR);
    }
else if ((pSbdCd = scsiBlkDevCreate (pSpd60, 0, 0) ) == NULL)
    {
    SCSI_DEBUG_MSG ("sysScsiConfig: scsiBlkDevCreate failed for CDROM.\n",
                    0, 0, 0, 0, 0, 0);
    return (ERROR);
    }
/*
 * Create an instance of a CD-ROM device in the I/O system.
 * A block device must already have been created.  Internally,
 * cdromFsDevCreate() calls iosDrvInstall(), which enters the
 * appropriate driver routines in the I/O driver table.
 */
if ((cdVolDesc = cdromFsDevCreate ("cdrom:", (BLK_DEV *) pSbdCd )) == NULL)
    {
    return (ERROR);
    }
#endif /* end of #ifdef INCLUDE_CDROMFS */
```

4. Before the definition of ***sysScsiConfig*( )**, declare the following global variables used in the above code fragment:

```
SCSI_PHYS_DEV *pSpd60;
BLK_DEV *pSbdCd;
CDROM_VOL_DESC_ID cdVolDesc;
```

The main goal of the above code fragment is to call ***cdromFsDevCreate*( )**. As input, ***cdromFsDevCreate*( )** expects a pointer to a block device. In the example above, the

***scsiPhysDevCreate*( )** and ***scsiBlkDevCreate*( )** calls set up a block device interface for a SCSI CD-ROM device.

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

INCLUDE FILES **cdromFsLib.h**

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

SEE ALSO **ioLib**, ISO 9660 Specification

cisLib

NAME **cisLib** – PCMCIA CIS library

ROUTINES

***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

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.

1

cisShow

NAME **cisShow** – PCMCIA CIS show library

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

DESCRIPTION This library provides a show routine for CIS tuples.

clockLib

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

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

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

A clock is a software construct that keeps time in seconds and nanoseconds. The clock has a simple interface with three routines: ***clock_settime*( )**, ***clock_gettime*( )**, and ***clock_getres*( )**. The non-POSIX routine ***clock_setres*( )** is provided (temporarily) so that **clockLib** is informed if there are changes in the system clock rate (e.g., after a call to ***sysClkRateSet*( )**).

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

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

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

INCLUDE FILES **timers.h**

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

cplusLib

NAME

cplusLib – basic run-time support for C++

ROUTINES

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

DESCRIPTION

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

– Support for C++ new and delete operators.

– Support for initialization and cleanup of static objects.

Shell utilities are provided for:

– Resolving overloaded C++ function names.

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

– Manual or automatic invocation of static constructors and destructors.

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

SEE ALSO

VxWorks Programmer's Guide: C++ Development

dbgArchLib

NAME

dbgArchLib – architecture-dependent debugger library

ROUTINES

***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)
***r0*( )** – return the contents of register **r0** (also **r1** – **r14**) (ARM)
***cpsr*( )** – return the contents of the current processor status register (ARM)
psrShow;*1*( ) – display the meaning of a specified PSR value, symbolically (ARM)

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

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

ROUTINES

***dbgHelp*( )** – display debugging help menu
***dbgInit*( )** – initialize the local debugging package
***b*( )** – set or display breakpoints
***e*( )** – set or display eventpoints (WindView)
***bh*( )** – set a hardware breakpoint
***bd*( )** – delete a breakpoint
***bdall*( )** – delete all breakpoints
***c*( )** – continue from a breakpoint

***cret*( )** – continue until the current subroutine returns
***s*( )** – single-step a task
***so*( )** – single-step, but step over a subroutine
***l*( )** – disassemble and display a specified number of instructions
***tt*( )** – display a stack trace of a task

DESCRIPTION

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

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

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

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

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

INITIALIZATION

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

BREAKPOINTS

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

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

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

EVENTPOINTS (WINDVIEW)

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

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

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

UNBREAKABLE TASKS

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

DISASSEMBLER AND STACK TRACER

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

SHELL ABORT AND EXCEPTION HANDLING

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

THE DEFAULT TASK AND TASK REFERENCING

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

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

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

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

dec21x4xEnd

NAME

dec21x4xEnd – END style DEC 21x4x PCI Ethernet network interface driver

ROUTINES

***dec21x4xEndLoad*( )** – initialize the driver and device

DESCRIPTION

This module implements a DEC 21x4x PCI Ethernet network interface driver and supports 21040, 21140 and 21143 versions of the chip.

The DEC 21x4x PCI Ethernet controller is little endian because it interfaces with a little endian PCI bus. Although PCI configuration for a device is handled in the BSP, all other device programming and initialization are handled in this module.

This driver is designed to be moderately generic. Without modification, it can operate across the range of architectures and targets supported by VxWorks. To achieve this, the driver requires a few external support routines as well as several target-specific parameters. 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, you need to modify the driver before it can operate correctly on your hardware.

On 21040, the driver configures the 10BASE-T interface by default, waits for two seconds, and checks the status of the link. If the link status indicates failure, AUI interface is configured.

On other versions of the 2114x family, the driver reads media information from a DEC serial ROM and configures the media. On targets that do not support a DEC format serial ROM, the driver calls a target-specfic media select routine using the hook, _func_dec2114xMediaSelect, to configure the media.

The driver supports big-endian or little-endian architectures (as a configurable option). The driver also and contains error recovery code that handles known device errata related to DMA activity.

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 which the chip DMAs to have to be swapped and written to, so that when the hardware swaps the accesses, the chip would see them correctly. The chip still has to 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 input and output byte stream need not be swapped.

BOARD LAYOUT

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

EXTERNAL INTERFACE

The driver provides one standard external interface, ***dec21x4xEndLoad*( )**, which a takes a string of colon separated parameters. The parameters should be specified as hexadecimal strings, optionally preceded by "0x" or a minus sign "-".

Although the parameter string is parsed using ***strtok_r( )***, each parameter is converted from string to binary by a call to strtoul(parameter, NULL, 16).

The format of the parameter string is:

"*unit number*:*device addr*:*PCI addr*:*ivec*:*ilevel*:*mem base*: *mem size*:*user flags*:*offset*"

TARGET-SPECIFIC PARAMETERS

unit number
: This represents the device instance number relative to this driver. I.e. a value of zero represents the first dec21x4x device, a value of 1 represents the second dec21x4x device.

device addr
: This is the base address at which the hardware device registers are located.

PCI addr
: This parameter defines the main memory address over the PCI bus. It is used to translate physical memory address into PCI accessible address.

ivec
: This is the interrupt vector number of the hardware interrupt generated by this Ethernet device. The driver uses intConnect, or pciIntConnect (x86 arch), to attach an interrupt handler for this interrupt.

ilevel
: This parameter defines the level of the hardware interrupt.

mem base
: This parameter specifies the base address of a DMA-able, cache free, pre-allocated memory region for use as a memory pool for transmit/receive descriptors and buffers.

If there is no pre-allocated memory available for the driver, this parameter should be -1 (NONE). In which case, the driver allocates cache safe memory for its use using ***cacheDmaAlloc( )***.

mem size
: The memory size parameter specifies the size of the pre-allocated memory region. If memory base is specified as NONE (-1), the driver ignores this parameter.

user flags
: User flags control the run-time characteristics of the Ethernet chip. Most flags specify non default CSR0 bit values. Refer to **dec21x4xEnd.h** for the bit values of the flags, and to the device hardware reference manual for details about device capabilities, and CSR 0.

Some of them are worth mentioning:

Full Duplex Mode: When set, the **DEC_USR_FD** flag allows the device to work in full duplex mode, as long as the PHY used has this capability. It is worth noting here that in this operation mode, the dec21x40 chip ignores the Collision and the Carrier Sense signals.

Transmit treshold value: The **DEC_USR_THR_XXX** flags enable the user to choose among different threshold values for the transmit FIFO. Transmission starts when the frame size within the transmit FIFO is larger than the treshold value. This should be selected taking into account the actual operating speed of the PHY. Again, see the device hardware reference manual for details.

offset
This parameter defines the offset which is used to solve alignment problem.

Device Type
Although the default device type is DEC 21040, specifying the **DEC_USR_21140** or **DEC_USR_21143** flag bit turns on DEC 21140 or **DEC_USR_21143** functionality.

Ethernet Address
The Ethernet address is retrieved from standard serial ROM on DEC 21040, DEC 21140 and DEC 21143 devices. If retrieve from ROM fails, the driver calls the BSP routine, ***sysDec21x4xEnetAddrGet*( )**. Specifying **DEC_USR_XEA** flag bit tells the driver should, by default, retrieve the Ethernet address using the BSP routine, ***sysDec21x4xEnetAddrGet*( )**.

Priority RX processing
The driver programs the chip to process the transmit and receive queues at the same priority. By specifying **DEC_USR_BAR_RX**, the device is programmed to process receives at a higher priority.

TX poll rate
By default, the driver sets the Ethernet chip into a non-polling mode. In this mode, if the transmit engine is idle, it is kick-started every time a packet needs to be transmitted. Alternately, the chip can be programmed to poll for the next available transmit descriptor if the transmit engine is in idle state. The poll rate is specified by one of **DEC_USR_TAP_xxx**.

Cache Alignment
The **DEC_USR_CAL_xxx** flags specify the address boundaries for data burst transfers.

DMA burst length
The **DEC_USR_PBL_xxx** flags specify the maximum number of long words in a DMA burst.

PCI multiple read
The **DEC_USR_RML** flag specifies that a device supports PCI memory-read-multiple.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires four external support functions, and provides a hook function:

void sysLanIntEnable (int level)
This routine provides a target-specific interface for enabling Ethernet device interrupts at a specified interrupt level.

void sysLanIntDisable (void)
This routine provides a target-specific interface for disabling Ethernet device interrupts.

STATUS sysDec21x4xEnetAddrGet (int unit, char *enetAdrs)
This routine provides a target-specific interface for accessing a device Ethernet address.

STATUS sysDec21143Init (DRV_CTRL * pDrvCtrl)
This routine performs any target-specific initialization required before the dec21143 device is initialized by the driver. The driver calls this routine every time it wants to load the device. This routine returns OK, or ERROR if it fails.

FUNCPTR _func_dec2114xMediaSelect
This driver provides a default media select routine, when _func_dec2114xMediaSelect is NULL, to read and setup physical media with configuration information from a Version 3 DEC Serial ROM. Any other media configuration can be supported by initializing <_func_dec2114xMediaSelect<, typically in ***sysHwInit*( )**, to a target-specific media select routine.

A media select routine is typically defined as:

```
STATUS decMediaSelect
    (
    DEC21X4X_DRV_CTRL *      pDrvCtrl,   /* Driver control */
    UINT *                   pCsr6Val    /* CSR6 return value */
    )
    {
        ...
    }
```

Parameter *pDrvCtrl* is a pointer to the driver control structure which this routine may use to access the Ethenet device. The driver control structure field mediaCount, is initialized to 0xff at startup, while the other media control fields (mediaDefault, mediaCurrent, and gprModeVal) are initialized to zero. This routine may use these fields in any manner, however all other driver control fields should be considered read-only and should not be modified.

This routine should reset, initialize and select an appropriate media, and write necessary the CSR6 bits (port select, PCS, SCR, and full duplex) to memory location pointed to by *pCsr6Val*. The driver will use this value to program register CSR6. This routine should return OK, and ERROR on failure.

FUNCPTR _func_dec2114xIntAck
This driver does acknowledge the LAN interrupts. However if the board hardware requires specific interrupt acknowledgement, not provided by this driver, the BSP

should define such a routine and attach it to the driver via _func_dec2114xIntAck.

SEE ALSO

ifLib, *DECchip 21040 Ethernet LAN Controller for PCI.*
Digital Semiconductor 21140A PCI Fast Ethernet LAN Controller.
Digital Semiconductor 21143 PCI/CardBus Fast Ethernet LAN Controller.
Using the Digital Semiconductor 21140A with Boot ROM, Serial ROM, and External Register: An Application Note
Using the Digital Semiconductor 21143 with Boot ROM, Serial ROM, and External Register: An Application Note

---

dec21x40End

NAME

dec21x40End – END-style DEC 21x40 PCI Ethernet network interface driver

ROUTINES

***dec21x40EndLoad*( )** – initialize the driver and device
***dec21140SromWordRead*( )** – read two bytes from the serial ROM
***dec21x40PhyLinkPoll*( )** – Poll the PHY for link status

DESCRIPTION

This module implements a DEC 21x40 PCI Ethernet network interface driver and supports both the 21040, 21140, and 21143 versions of the chip.

The DEC 21x40 PCI Ethernet controller is little endian because it interfaces with a little-endian PCI bus. Although PCI configuration for a device is handled in the BSP, all other device programming and initialization needs are handled in this module.

This driver is designed to be moderately generic. Without modification, it can operate across the full range of architectures and targets supported by VxWorks. To achieve this, the driver requires a few external support routines as well as several target-specific parameters. 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, you need to modify the driver before it can operate correctly on your hardware.

On the 21040, the driver configures the 10BASE-T interface by default, waits for two seconds, and checks the status of the link. If the link status indicates failure, AUI interface is configured.

On other versions of the 21x40 family, the driver reads media information from a DEC serial ROM and configures the media. To configure the media on targets that do not support a DEC format serial ROM, the driver calls the target-specific media-select routine referenced in the **_func_dec21x40MediaSelect** hook.

The driver supports big-endian or little-endian architectures (as a configurable option). The driver also and contains error recovery code that handles known device errata related to DMA activity.

Big-endian processors can be connected to the PCI bus through some controllers that take care of hardware byte swapping. In such cases, all the registers which the chip DMAs to have to be swapped and written to, so that when the hardware swaps the accesses, the chip would see them correctly. The chip still has to 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 input and output byte stream need not be swapped.

BOARD LAYOUT

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

EXTERNAL INTERFACE

The driver provides one standard external interface, ***dec21x40EndLoad*( )**. As input, this function expects a string of colon-separated parameters. The parameters should be specified as hexadecimal strings (optionally preceded by "0x" or a minus sign "-"). Although the parameter string is parsed using ***strtok_r*( )**, each parameter is converted from string to binary by a call to:

```
strtoul(parameter, NULL, 16).
```

The format of the parameter string is:

device_addr*:*PCI_addr*:*ivec*:*ilevel*:*num_rds*:*num_tds*:*mem_base*:*mem_size*:*user_flags

TARGET-SPECIFIC PARAMETERS

device_addr

This is the base address at which the hardware device registers are located.

PCI_addr

This parameter defines the main memory address over the PCI bus. It is used to translate a physical memory address into a PCI-accessible address.

ivec

This is the interrupt vector number of the hardware interrupt generated by this Ethernet device. The driver uses ***intConnect*( )** to attach an interrupt handler for this interrupt.

ilevel

This parameter defines the level of the hardware interrupt.

num_rds

The number of receive descriptors to use. This controls how much data the device can absorb under load. If this is specified as NONE (-1), the default of 32 is used.

num_tds

The number of transmit descriptors to use. This controls how much data the device can absorb under load. If this is specified as NONE (-1) then the default of 64 is used.

mem_base

This parameter specifies the base address of a DMA-able cache-free pre-allocated memory region for use as a memory pool for transmit/receive descriptors and buffers. If there is no pre-allocated memory available for the driver, this parameter should be -1 (NONE). In which case, the driver allocates cache safe memory for its use using ***cacheDmaAlloc*( )**.

mem_size
The memory size parameter specifies the size of the pre-allocated memory region. If memory base is specified as NONE (-1), the driver ignores this parameter.

user_flags
User flags control the run-time characteristics of the Ethernet chip. Most flags specify non default CSR0 and CSR6 bit values. Refer to **dec21x40End.h** for the bit values of the flags and to the device hardware reference manual for details about device capabilities, CSR6 and CSR0.

Device Type: Although the default device type is DEC 21040, specifying the **DEC_USR_21140** flag bit turns on DEC 21140 functionality.

Ethernet Address: The Ethernet address is retrieved from standard serial ROM on both DEC 21040, and DEC 21140 devices. If the retrieve from ROM fails, the driver calls the ***sysDec21x40EnetAddrGet*( )** BSP routine. Specifying **DEC_USR_XEA** flag bit tells the driver should, by default, retrieve the Ethernet address using the ***sysDec21x40EnetAddrGet*( )** BSP routine.

Priority RX processing: The driver programs the chip to process the transmit and receive queues at the same priority. By specifying **DEC_USR_BAR_RX**, the device is programmed to process receives at a higher priority.

TX poll rate: By default, the driver sets the Ethernet chip into a non-polling mode. In this mode, if the transmit engine is idle, it is kick-started every time a packet needs to be transmitted. Alternatively, the chip can be programmed to poll for the next available transmit descriptor if the transmit engine is in idle state. The poll rate is specified by one of **DEC_USR_TAP_xxx** flags.

Cache Alignment: The **DEC_USR_CAL_xxx** flags specify the address boundaries for data burst transfers.

DMA burst length: The **DEC_USR_PBL_xxx** flags specify the maximum number of long words in a DMA burst.

PCI multiple read: The **DEC_USR_RML** flag specifies that a device supports PCI memory-read-multiple.

Full Duplex Mode: When set, the **DEC_USR_FD** flag allows the device to work in full duplex mode, as long as the PHY used has this capability. Note that in this operation mode, the dec21x40 chip ignores the Collision and the Carrier Sense signals.

MII/Phy Checking: When set, and when a MII interface is being utilized the **DEC_USR_PHY_CHK** flag instructs the driver to wait until the PHY link status has changed to **up** before continuing. This time period could be as long as six seconds, but in general is on the order of two seconds. If clear, the check will not be performed. This option may be selected if the delay is unacceptable, but it is possible that a fast target may attempt to send packets before the link is up. This will result in **no carrier** errors in packet transmission.

Transmit treshold value: The **DEC_USR_THR_XXX** flags enable the user to choose among different threshold values for the transmit FIFO. Transmission starts when the frame size within the transmit FIFO is larger than the treshold value. This should be selected taking into account the actual operating speed of the PHY. Again, see the device hardware reference manual for details.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires three external support functions and provides a hook function:

sysLanIntEnable( )

```
void sysLanIntEnable (int level)
```

This routine provides a target-specific interface for enabling Ethernet device interrupts at a specified interrupt level.

sysLanIntDisable( )

```
void sysLanIntDisable (void)
```

This routine provides a target-specific interface for disabling Ethernet device interrupts.

sysDec21x40EnetAddrGet( )

```
STATUS sysDec21x40EnetAddrGet (int unit, char *enetAdrs)
```

This routine provides a target-specific interface for accessing a device Ethernet address.

_func_dec21x40MediaSelect

```
FUNCPTR _func_dec21x40MediaSelect
```

If **_func_dec21x40MediaSelect** is NULL, this driver provides a default media-select routine that reads and sets up physical media using the configuration information from a Version 3 DEC Serial ROM. Any other media configuration can be supported by initializing **_func_dec21x40MediaSelect**, typically in *sysHwInit*( ), to a target-specific media select routine.

A media select routine is typically defined as:

```
STATUS decMediaSelect
    (
    DEC21X40_DRV_CTRL *      pDrvCtrl,   /* driver control */
    UINT *                   pCsr6Val    /* CSR6 return value */
    )
    {
        ...
    }
```

The *pDrvCtrl* parameter is a pointer to the driver control structure that this routine can use to access the Ethenet device. The driver control structure member **mediaCount**, is initialized to 0xff at startup, while the other media control members (**mediaDefault**,

mediaCurrent, and **gprModeVal**) are initialized to zero. This routine can use these fields in any manner. However, all other driver control structure members should be considered read-only and should not be modified.

This routine should reset, initialize, and select an appropriate media. It should also write necessary the CSR6 bits (port select, PCS, SCR, and full duplex) to the memory location pointed to by *pCsr6Val*. The driver uses this value to program register CSR6. This routine should return OK or ERROR.

SEE ALSO

ifLib, *DECchip 21040 Ethernet LAN Controller for PCI, Digital Semiconductor 21140A PCI Fast Ethernet LAN Controller, Using the Digital Semiconductor 21140A with Boot ROM, Serial ROM, and External Register: An Application Note*

---

dhcpcBootLib

NAME

dhcpcBootLib – DHCP boot-time client library

ROUTINES

***dhcpcBootInit*( )** – set up the DHCP client parameters and data structures
***dhcpcBootBind*( )** – initialize the network with DHCP at boot time
***dhcpcBootOptionSet*( )** – add an option to the option request list

DESCRIPTION

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

HIGH-LEVEL INTERFACE

The VxWorks boot program uses this library to obtain configuration parameters with DHCP according to the client-server interaction detailed in RFC 1541 using the boot device specified in the boot parameters. The boot device must be capable of sending broadcast messages. Currently, only Ethernet devices and the shared-memory network drivers are supported. To use DHCP, first build a boot ROM image with **INCLUDE_DHCPC** defined and set the appropriate flag in the boot parameters before initiating booting with the "@" command. The DHCP client will attempt to retrieve entries for the boot file name, host IP address, and target IP address, as well as a subnet mask and broadcast address for the boot device. Any entries retrieved will only be used if the corresponding fields in the boot parameters are blank.

NOTE

After DHCP retrieves the boot parameters, the specified boot file is loaded and the system restarts. As a result, the boot-time DHCP client cannot renew the lease associated with the assigned IP address. To avoid potential IP address conflicts while loading the boot file, the **DHCPC_MIN_LEASE** value should be set to exceed the file transfer time. In addition, the boot file must also contain the DHCP client library so that the lease obtained before the restart can be renewed. Otherwise, the network initialization using the boot parameters will fail.

INCLUDE FILES

dhcpcBootLib.h

SEE ALSO

dhcpcLib, RFC 1541, RFC 1533

dhcpcLib

NAME

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

ROUTINES

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

DESCRIPTION

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

CONFIGURATION INTERFACE

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

HIGH-LEVEL INTERFACE

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

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

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

LOW-LEVEL INTERFACE

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

OPTIONAL INTERFACE

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

INCLUDE FILES **dhcpcLib.h**

SEE ALSO RFC 1541, RFC 1533

dhcpcShow

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

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

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

INCLUDE FILES **dhcpcLib.h**

SEE ALSO **dhcpcLib**, *Network Programmer's Guide: Network Configuration Protocols*

dhcprLib

NAME **dhcprLib** – DHCP relay agent library

ROUTINES No Callable Routines

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

HIGH-LEVEL INTERFACE

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

INCLUDE FILES **dhcprLib.h**

SEE ALSO RFC 1541, RFC 1533

dhcpsLib

NAME **dhcpsLib** – Dynamic Host Configuration Protocol (DHCP) server library

ROUTINES

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

DESCRIPTION

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

PRIMARY INTERFACE

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

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

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

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

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

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

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

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

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

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

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

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

short
: Takes a 16-bit integer.

long
: Takes a 32-bit integer.

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

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

ippairs

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

mtpt

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

clid

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

The following table lists the option specifiers and descriptions for every possible entry in the parameter list. When available, the option code from RFC 1533 is included.

Name	Code	Type	Description
snam	-	str	Optional server name.
file	-	str	Name of file containing the boot image.
siad	-	ip	Address of server that offers the boot image.
albp	-	bool	If true, this entry is also available to BOOTP clients. For entries using static allocation, this value becomes true by default and *maxl* becomes infinity.
maxl	-	long	Maximum lease duration in seconds.
dfll	-	long	Default lease duration in seconds. If a client does not request a specific lease duration, the server uses this value.
clid	-	clid	This specifies a client identifier for manual leases. The VxWorks client uses a MAC address as the client identifier.
pmid	-	clid	This specifies a client identifier for client-specific parameters to be included in a lease. It should be present in separate entries without IP addresses.
clas	-	str	This specifies a class identifier for class-specific parameters to be included in a lease. It should be present in separate entries without IP addresses.
snmk	1	ip	Subnet mask of the IP address to be allocated. The default is a natural mask corresponding to the IP address. The server will not issue IP addresses to clients on different subnets.
tmof	2	long	Time offset from UTC in seconds.
rout	3	iplist	A list of routers on the same subnet as the client.
tmsv	4	iplist	A list of time servers (RFC 868).
nmsv	5	iplist	A list of name servers (IEN 116).
dnsv	6	iplist	A list of DNS servers (RFC 1035).
lgsv	7	iplist	A list of MIT-LCS UDP log servers.
cksv	8	iplist	A list of Cookie servers (RFC 865).
lpsv	9	iplist	A list of LPR servers (RFC 1179).

Name	Code	Type	Description
imsv	10	iplist	A list of Imagen Impress servers.
rlsv	11	iplist	A list of Resource Location servers (RFC 887).
hstn	12	str	Hostname of the client.
btsz	13	short	Size of boot image.
mdmp	14	str	Path name to which client dumps core.
dnsd	15	str	Domain name for DNS.
swsv	16	ip	IP address of swap server.
rpth	17	str	Path name of root disk of the client.
epth	18	str	Extensions Path (See RFC 1533).
ipfd	19	bool	If true, the client performs IP forwarding.
nlsr	20	bool	If true, the client can perform non-local source routing.
plcy	21	ippairs	Policy filter for non-local source routing. A list of pairs of (Destination IP, Subnet mask).
mdgs	22	short	Maximum size of IP datagram that the client should be able to reassemble.
ditl	23	octet	Default IP TTL.
mtat	24	long	Aging timeout (in seconds) to be used with Path MTU discovery (RFC 1191).
mtpt	25	mtpt	A table of MTU sizes to be used with Path MTU Discovery.
ifmt	26	short	MTU to be used on an interface.
asnl	27	bool	If true, the client assumes that all subnets to which the client is connected use the same MTU.
brda	28	ip	Broadcast address in use on the client's subnet. The default is calculated from the subnet mask and the IP address.
mskd	29	bool	If true, the client should perform subnet mask discovery using ICMP.
msks	30	bool	If true, the client should respond to subnet mask requests using ICMP.
rtrd	31	bool	If true, the client should solicit routers using Router Discovery defined in RFC 1256.
rtsl	32	ip	Destination IP address to which the client sends router solicitation requests.
strt	33	ippairs	A table of static routes for the client, which are pairs of (Destination, Router). It is illegal to specify default route as a destination.
trlr	34	bool	If true, the client should negotiate the use of trailers with ARP (RFC 893).
arpt	35	long	Timeout in seconds for ARP cache.
encp	36	bool	If false, the client uses RFC 894 encapsulation. If true, it uses RFC 1042 (IEEE 802.3) encapsulation.
dttl	37	octet	Default TTL of TCP.

Name	Code	Type	Description
kain	38	long	Interval of the client's TCP keepalive in seconds.
kagb	39	bool	If true, the client should send TCP keepalive messages with a octet of garbage for compatibility.
nisd	40	str	Domain name for NIS.
nisv	41	iplist	A list of NIS servers.
ntsv	42	iplist	A list of NTP servers.
nnsv	44	iplist	A list of NetBIOS name server. (RFC 1001, 1002)
ndsv	45	iplist	A list of NetBIOS datagram distribution servers (RFC 1001, 1002).
nbnt	46	octet	NetBIOS node type (RFC 1001, 1002).
nbsc	47	str	NetBIOS scope (RFC 1001, 1002).
xfsv	48	iplist	A list of font servers of X Window system.
xdmn	49	iplist	A list of display managers of X Window system.
dht1	58	short	This value specifies when the client should start RENEWING. The default of 500 means the client starts RENEWING after 50% of the lease duration passes.
dht1	59	short	This value specifies when the client should start REBINDING. The default of 875 means the client starts REBINDING after 87.5% of the lease duration passes.

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

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

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

SECONDARY INTERFACE

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

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

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

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

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

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

OPTIONAL INTERFACE

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

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

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

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

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

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

INCLUDE FILES **dhcpsLib.h**

SEE ALSO RFC 1541, RFC 1533

dirLib

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

ROUTINES

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

***statfs*( )** – get file status information using a pathname (POSIX)
***utime*( )** – update time on a file

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

SEARCHING DIRECTORIES

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

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

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

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

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

GETTING FILE INFORMATION

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

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

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

S_ISREG
test if the file is a regular file

S_ISDIR
test if the file is a directory

S_ISCHR
test if the file is a character special file

S_ISBLK
test if the file is a block special file

S_ISFIFO
test if the file is a FIFO special file

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

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

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

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

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

dosFsLib

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

ROUTINES
***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

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*( )**.

This initialization is enabled when the configuration macro **INCLUDE_DOSFS** is defined; ***dosFsInit*( )** is then 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 ("o 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 **FIOUNMOUNT** function code. (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 portion of its 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.

Do not attempt to use ***dosFsVolUnmount*( )** with volumes mounted using ***usrFdConfig*( )**. This routine does not return the **DOS_VOL_CONFIG** structure required by ***dosFsVolUnmount*( )**. Instead use ***ioctl*( )** with **FIOUNMOUNT**, which accesses the volume information via the file descriptor.

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. Note that this will happen the first time the disk is accessed after the volume has been 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
Performs the same function as FIOFLUSH, and 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 (located 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, the ***readdir*( )** routine is used to read a directory, rather than using the **FIOREADDIR** function directly. See **dirLib**.

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

FIOFSTATGET

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

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

Any other ***ioctl*( )** function codes are passed to the block device driver for handling.

MEMORY CONSUMPTION

In order to minimize memory fragmentation in the system memory pool, all memory consumed by **dosFsLib** will be contained within a dedicated memory partition. This partition is accessible via the *dosFsMemPartId* global variable.

To display the current amount of memory used by **dosFsLib**, call show(dosFsMemPartId). Please see the manual page for ***memPartShow*( )** for more details.

The following varibles may be set *before* ***dosFsLib*** is initialized to change the behavior of the memory management.

If the **dosFsLib** memory partition is not provided, one will be allocated from the system memory pool. It's size defaults to 8 K, which may be changed via the *dosFsMemPartInitSize* global. To provide a memory pool, set *dosFsMemPartId* to a valid **PART_ID** returned from ***memPartCreate*( )**.

The global variable *dosFsMemPartIdOptions* may be modified to change the behavior of error handling for errors in ***malloc*( )** and ***free*( )**. The options default to **MEM_BLOCK_ERROR_LOG_FLAG**, which will log information about errors detected by ***free*( )**. These options only affect operations on the dosFs memory partition.

The private partition will dynamically grow as much as needed, allocating additional memory from the system memory pool, in units no smaller than 1 Kilobyte. This minumum unit size may be adjusted via the *dosFsMemPartGrowSize* global variable.

The maximum size for the dosFs memory partition may be limited via the global variable *dosFsMemPartCap*. Once the cap limit has been reached or surpassed, dosFs will not attempt to allocate more memory from the system memory partition. The default value is -1, which allows uninterupted use of the system memory partition.

Additional debugging may be enabled via the global boolean *dosFsDebug*. Setting this to 1 will enable verbose debug messages from the dosFs memory manager.

INCLUDE FILES **dosFsLib.h**

SEE ALSO **dosFsLib**, **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*

ei82596End

NAME **ei82596End** – END style Intel 82596 Ethernet network interface driver

ROUTINES ***ei82596EndLoad*( )** – initialize the driver and device

DESCRIPTION This module implements an Intel 82596 Ethernet network interface driver. This driver is designed to be moderately generic. It operates unmodified across the range of architectures and targets supported by VxWorks. To achieve this, this driver requires some external support routines as well as several target-specific parameters. 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

The driver provides one standard external interface, ***ei82596EndLoad*( )**. As input, this routine takes a string of colon-separated parameters. The parameters should be specified in hexadecimal (optionally preceded by "0x" or a minus sign "-"). The parameter string is parsed using ***strtok_r*( )**, and each parameter is converted from string to binary by a call to:

```
strtoul(parameter, NULL, 16).
```

TARGET-SPECIFIC PARAMETERS

The format of the parameter string is:

unit:*ivec*:*sysbus*:*memBase*:*nTfds*:*nRfds*:*offset*

unit
: A convenient holdover from the former model. It is only used in the string name for the driver.

ivec
: This is the interrupt vector number of the hardware interrupt generated by this ethernet device. The driver uses ***intConnect*( )** to attach an interrupt handler to this interrupt.

sysbus
: This parameter tells the device about the system bus. To determine the correct value for a target, see *Intel 32-bit Local Area Network (LAN) Component User's Manual.*

memBase
: This parameter specifies the base address of a DMA-able cache-free pre-allocated memory region for use as a memory pool for transmit/receive descriptors, buffers, and other device control structures. If there is no pre-allocated memory available for the driver, this parameter should be -1 (NONE). In which case, the driver calls ***cacheDmaAlloc*( )** to allocate cache-safe memory.

nTfds
: This parameter specifies the number of transmit descriptor/buffers to be allocated. If this parameter is zero or -1 (NULL), a default of 32 is used.

nRfds
: This parameter specifies the number of receive descriptor/buffers to be allocated. If this parameter is zero or -1 (NULL), a default of 32 is used.

offset
: Specifies the memory alignment offset.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires seven external support functions:

***sys596IntEnable*( )**

```
void sys596IntEnable (int unit)
```

This routine provides a target-specific interface to enable Ethernet device interrupts for a given device unit.

sys596IntDisable( )

```
void sys596IntDisable (int unit)
```

This routine provides a target-specific interface to disable Ethernet device interrupts for a given device unit.

sysEnetAddrGet( )

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

This routine provides a target-specific interface to access a device Ethernet address. This routine should provide a six-byte Ethernet address in the *enetAdrs* parameter and return OK or ERROR.

sys596Init( )

```
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.

sys596Port( )

```
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 and, under some conditions, during error recovery procedures.

sys596ChanAtn( )

```
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.

sys596IntAck( )

```
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

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 on RFDs and TFDs, see the *Intel 82596 User's Manual.*

The 82596 requires ether that this shared memory region is non-cacheable or that the hardware implements bus snooping. The driver cannot maintain cache coherency for the device. This is because fields within the command structures are asynchronously modified by both the driver and the device, and these fields might share the same cache line.

TUNING HINTS

The only adjustable parameters are the number of TFDs and RFDs that are 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 might be a good idea. Increasing the number of TFDs provides no performance benefit after a certain point. Increasing the number of 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 User's Manual*

el3c90xEnd

NAME

el3c90xEnd – END network interface driver for 3COM 3C90xB XL

ROUTINES

***el3c90xEndLoad*( )** – initialize the driver and device
***el3c90xInitParse*( )** – parse the initialization string

DESCRIPTION

This module implements the device driver for the 3COM EtherLink Xl and Fast EtherLink XL PCI network interface cards.

The 3c90x 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 in the driver. The second part of the interface comprises of the I/O control registers and their programming. The third part of the interface comprises of the descriptors and the 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 target-specific values and the external support routines are described below.

This driver supports multiple 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.

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 which the chip DMA s to have to be swapped and written to, so that when the hardware swaps the accesses, the chip would see them correctly. The chip still has to be programmed to operated 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 input and output byte stream need not be swapped.

The 3c90x series chips use a bus-master DMA interface for transfering packets to and from the controller chip. Some of the old 3c59x cards also supported a bus master mode, however for those chips you could only DMA packets to and from a contiguous memory buffer. For transmission this would mean copying the contents of the queued **M_BLK** chain into a an **M_BLK** cluster and then DMAing the cluster. This extra copy would sort of defeat the purpose of the bus master support for any packet that doesn't fit into a single **M_BLK**. By contrast, the 3c90x cards support a fragment-based bus master mode where **M_BLK** chains can be encapsulated using TX descriptors. This is also called the gather technique, where the fragments in an mBlk chain are directly incorporated into the download transmit descriptor. This avoids any copying of data from the mBlk chain.

NETWORK CARDS SUPPORTED

– 3Com 3c900-TPO 10Mbps/RJ-45
– 3Com 3c900-COMBO 10Mbps/RJ-45,AUI,BNC
– 3Com 3c905-TX 10/100Mbps/RJ-45
– 3Com 3c905-T4 10/100Mbps/RJ-45
– 3Com 3c900B-TPO 10Mbps/RJ-45
– 3Com 3c900B-COMBO 10Mbps/RJ-45,AUI,BNC
– 3Com 3c905B-TX 10/100Mbps/RJ-45
– 3Com 3c905B-FL/FX 10/100Mbps/Fiber-optic
– 3Com 3c980-TX 10/100Mbps server adapter
– Dell Optiplex GX1 on-board 3c918 10/100Mbps/RJ-45

BOARD LAYOUT

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

EXTERNAL INTERFACE

The only external interface is the *el3c90xEndLoad*( ) routine, which expects the *initString* parameter as input. This parameter passes in a colon-delimited string of the format:

unit:*devMemAddr*:*devIoAddr*:*pciMemBase:<vecNum*:*intLvl*:*memAdrs*:
memSize:*memWidth*:*flags*:*buffMultiplier*

The ***el3c90xEndLoad*( )** function uses ***strtok*( )** to parse the string.

TARGET-SPECIFIC PARAMETERS

unit
: A convenient holdover from the former model. This parameter is used only in the string name for the driver.

devMemAddr
This parameter in the memory base address of the device registers in the memory map of the CPU. It indicates to the driver where to find the register set. < This parameter should be equal to NONE if the device does not support memory mapped registers.

devIoAddr
This parameter in the IO base address of the device registers in the IO map of some CPUs. It indicates to the driver where to find the RDP register. If both *devIoAddr* and *devMemAddr* are given then the device chooses *devMemAddr* which is a memory mapped register base address. This parameter should be equal to NONE if the device does not support IO mapped registers.

pciMemBase

This parameter is the base address of the CPU memory as seen from the PCI bus. This parameter is zero for most intel architectures.

vecNum
This parameter is the vector associated with the device interrupt. 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.

intLvl
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 ***sysEl3c90xIntEnable*( )** is called to perform any board-specific operations required to allow the servicing of a NIC interrupt. For a description of ***sysEl3c90xIntEnable*( )**, see "External Support Requirements" below.

memAdrs
This parameter gives the driver the memory address to carve out its buffers and data structures. If this parameter is specified to be NONE then the driver allocates cache coherent memory for buffers and descriptors from the system pool. The 3C90x NIC 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 NIC. It assumes that this shared memory is directly available to it without any arbitration or timing concerns.

memSize
This parameter can be used to explicitly limit the amount of shared memory (bytes) this driver will use. 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.

memWidth

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.

flags

This is parameter is used for future use, currently its value should be zero.

buffMultiplier

This parameter is used increase the number of buffers allocated in the driver pool. If this parameter is -1 then a default multiplier of 2 is choosen. With a multiplier of 2 the total number of clusters allocated is 64 which is twice the cumulative number of upload and download descriptors. The device has 16 upload and 16 download descriptors. For example on choosing the buffer multiplier of 3, the total number of clusters allocated will be 96 ((16 + 16)*3). There are as many clBlks as the number of clusters. The number of mBlks allocated are twice the number of clBlks. By default there are 64 clusters, 64 clBlks and 128 mBlks allocated in the pool for the device. Depending on the load of the system increase the number of clusters allocated by incrementing the buffer multiplier.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires several external support functions, defined as macros:

```
SYS_INT_CONNECT(pDrvCtrl, routine, arg)
SYS_INT_DISCONNECT (pDrvCtrl, routine, arg)
SYS_INT_ENABLE(pDrvCtrl)
SYS_INT_DISABLE(pDrvCtrl)
SYS_OUT_BYTE(pDrvCtrl, reg, data)
SYS_IN_BYTE(pDrvCtrl, reg, data)
SYS_OUT_WORD(pDrvCtrl, reg, data)
SYS_IN_WORD(pDrvCtrl, reg, data)
SYS_OUT_LONG(pDrvCtrl, reg, data)
SYS_IN_LONG(pDrvCtrl, reg, data)
SYS_DELAY (delay)
sysEl3c90xIntEnable(pDrvCtrl->intLevel)
sysEl3c90xIntDisable(pDrvCtrl->intLevel)
sysDelay (delay)
```

There are default values in the source code for these macros. They presume memory mapped accesses to the device registers and the normal ***intConnect( )***, and ***intEnable( )*** BSP functions. The first argument to each is the device controller structure. Thus, each has access back to all the device-specific information. Having the pointer in the macro facilitates the addition of new features to this driver.

The macros **SYS_INT_CONNECT**, **SYS_INT_DISCONNECT**, **SYS_INT_ENABLE**, and **SYS_INT_DISABLE** allow the driver to be customized for BSPs that use special versions of these routines.

The macro **SYS_INT_CONNECT** is used to connect the interrupt handler to the appropriate vector. By default it is the routine ***intConnect*( )**.

The macro **SYS_INT_DISCONNECT** is used to disconnect the interrupt handler prior to unloading the module. By default this is a dummy routine that returns OK.

The macro **SYS_INT_ENABLE** is used to enable the interrupt level for the end device. It is called once during initialization. It calls an external board level routine ***sysEl3c90xIntEnable*( )**.

The macro **SYS_INT_DISABLE** is used to disable the interrupt level for the end device. It is called during stop. It calls an external board level routine ***sysEl3c90xIntDisable*( )**.

The macro **SYS_DELAY** is used for a delay loop. It calls an external board level routine sysDelay(delay). The granularity of delay is one microsecond.

SYSTEM RESOURCE USAGE

When implemented, this driver requires the following system resources:

– one mutual exclusion semaphore
– one interrupt vector
– 24072 bytes in text for a I80486 target
– 112 bytes in the initialized data section (data)
– 0 bytes in the uninitialized data section (BSS)

The driver allocates clusters of size 1536 bytes for receive frames and and transmit frames. There are 16 descriptors in the upload ring and 16 descriptors in the download ring. The buffer multiplier by default is 2, which means that the total number of clusters allocated by default are 64 ((upload descriptors + download descriptors)*2). There are as many clBlks as the number of clusters. The number of mBlks allocated are twice the number of clBlks. By default there are 64 clusters, 64 clBlks and 128 mBlks allocated in the pool for the device. Depending on the load of the system increase the number of clusters allocated by incrementing the buffer multiplier.

INCLUDES

end.h endLib.h etherMultiLib.h el3c90xEnd.h

SEE ALSO

muxLib, **endLib**, **netBufLib**, *VxWorks Programmer's Guide: Writing and Enhanced Network Driver*

BIBLIOGRAPHY

3COM 3c90x and 3c90xB NICs Technical reference.

elt3c509End

NAME **elt3c509End** – END network interface driver for 3COM 3C509

ROUTINES ***elt3c509Load*( )** – initialize the driver and device
***elt3c509Parse*( )** – parse the init string

DESCRIPTION This module implements the 3COM 3C509 EtherLink III Ethernet network interface driver. This driver is designed to be moderately generic. Thus, it operates unmodified across the range of architectures and targets supported by VxWorks. To achieve this, the driver load routine requires an input string consisting of several target-specific values. The driver also requires some external support routines. These target-specific values and the external support routines are described below.

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

EXTERNAL INTERFACE

The only external interface is the ***elt3c509Load*( )** routine, which expects the *initString* parameter as input. This parameter passes in a colon-delimited string of the format:

unit:*port*:*intVector*:*intLevel*:*attachementType*:*nRxFrames*

The ***elt3c509Load*( )** function uses ***strtok*( )** to parse the string.

TARGET-SPECIFIC PARAMETERS

unit

A convenient holdover from the former model. This parameter is used only in the string name for the driver.

intVector

Configures the ELT 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 ELT interrupt.

intLevel

This parameter is passed to an external support routine, ***sysEltIntEnable*( )**, which is described below in "External Support Requirements." This routine is called during as part of driver's initialization. It handles any board-specific operations required to allow the servicing of a ELT interrupt on targets that use additional interrupt controller devices to help organize and service the various interrupt sources. This parameter makes it possible for this driver to avoid all board-specific knowledge of such devices.

attachmentType
This parameter is used to select the transceiver hardware attachment. This is then used by the ***elt3c509BoardInit*( )** routine to activate the selected attachment. ***elt3c509BoardInit*( )** is called as a part of the driver's initialization.

nRxFrames
This parameter is used as number of receive frames by the driver.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires several external support functions, defined as macros:

```
SYS_INT_CONNECT(pDrvCtrl, routine, arg)
SYS_INT_DISCONNECT (pDrvCtrl, routine, arg)
SYS_INT_ENABLE(pDrvCtrl)
SYS_INT_DISABLE(pDrvCtrl)
SYS_OUT_BYTE(pDrvCtrl, reg, data)
SYS_IN_BYTE(pDrvCtrl, reg, data)
SYS_OUT_WORD(pDrvCtrl, reg, data)
SYS_IN_WORD(pDrvCtrl, reg, data)
SYS_OUT_WORD_STRING(pDrvCtrl, reg, pData, len)
SYS_IN_WORD_STRING(pDrvCtrl, reg, pData, len)

sysEltIntEnable(pDrvCtrl->intLevel)
sysEltIntDisable(pDrvCtrl->intLevel)
```

There are default values in the source code for these macros. They presume IO-mapped accesses to the device registers and the normal ***intConnect*( )**, and ***intEnable*( )** BSP functions. The first argument to each is the device controller structure. Thus, each has access back to all the device-specific information. Having the pointer in the macro facilitates the addition of new features to this driver.

The macros **SYS_INT_CONNECT**, **SYS_INT_DISCONNECT**, and **SYS_INT_ENABLE** allow the driver to be customized for BSPs that use special versions of these routines.

The macro **SYS_INT_CONNECT** is used to connect the interrupt handler to the appropriate vector. By default it is the routine ***intConnect*( )**.

The macro **SYS_INT_DISCONNECT** is used to disconnect the interrupt handler prior to unloading the module. By default this is a dummy routine that returns OK.

The macro **SYS_INT_ENABLE** is used to enable the interrupt level for the end device. It is called once during initialization. It calls an external board level routine ***sysEltIntEnable*( )**.

The macro **SYS_INT_DISABLE** is used to disable the interrupt level for the end device. It is called during stop. It calls an external board level routine ***sysEltIntDisable*( )**.

SYSTEM RESOURCE USAGE

When implemented, this driver requires the following system resources:

– one interrupt vector
– 9720 bytes of text
– 88 bytes in the initialized data section (data)
– 0 bytes of bss

The driver requires 1520 bytes of preallocation for Transmit Buffer and 1520*nRxFrames of receive buffers. The default value of nRxFrames is 64 therefore total pre-allocation is (64 + 1)*1520.

TUNING HINTS nRxFrames parameter can be used for tuning no of receive frames to be used for handling packet receive. More no. of these could help receiving more loaning in case of massive reception.

INCLUDES **end.h endLib.h etherMultiLib.h elt3c509End.h**

SEE ALSO **muxLib**, **endLib***Writing and Enhanced Network Driver*

endLib

NAME **endLib** – support library for END-based drivers

ROUTINES
***mib2Init*( )** – initialize a MIB-II structure
***mib2ErrorAdd*( )** – change a MIB-II error count
***endObjInit*( )** – initialize an **END_OBJ** structure
***endObjFlagSet*( )** – set the **flags** member of an **END_OBJ** structure
***endEtherAddressForm*( )** – form an Ethernet address into a packet
***endEtherPacketDataGet*( )** – return the beginning of the packet data
***endEtherPacketAddrGet*( )** – locate the addresses in a packet

DESCRIPTION This library contains support routines for Enhanced Network Drivers. These routines are common to ALL ENDs. Specialized routines should only appear in the drivers themselves.

envLib

NAME **envLib** – environment variable library

ROUTINES
***envLibInit*( )** – initialize environment variable facility
***envPrivateCreate*( )** – create a private environment

***envPrivateDestroy*( )** – destroy a private environment
***putenv*( )** – set an environment variable
***getenv*( )** – get an environment variable (ANSI)
***envShow*( )** – display the environment for a task

DESCRIPTION

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

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

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

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

INCLUDE FILES

envLib.h

SEE ALSO

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

---

errnoLib

NAME

errnoLib – error status library

ROUTINES

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

DESCRIPTION

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

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

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

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

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

ERROR STATUS VALUES

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

PRINTING ERROR STATUS VALUES

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

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

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

```
#define S_myFile_ERROR_TOO_MANY_COOKS        0x230003
```

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

On UNIX:

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

On Windows:

```
makeStatTbl target/h
```

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

If the user now types from the VxWorks shell:

```
-> printErrno 0x230003
```

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

```
S_myFile_ERROR_TOO_MANY_COOKS
```

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

```
#define S_xxx  n
```

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

```
#define S_thisFile_MEANINGFUL_ERROR_MESSAGE   0xnnnn
```

where *thisFile* is the name of the module.

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

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

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

etherLib

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

ROUTINES

***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
***etherTypeGet*( )** – get the type from an ethernet packet

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_cpm – Motorola MC68EN360 QUICC network interface driver
if_eex – Intel EtherExpress 16
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_fn – Fujitsu MB86960 NICE Ethernet driver
if_ln – Advanced Micro Devices Am7990 LANCE Ethernet 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

if_gn – generic MUX interface layer

The following drivers support only the input-hook routines:

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

etherMultiLib

NAME

etherMultiLib – a library to handle Ethernet multicast addresses

ROUTINES

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

DESCRIPTION

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

INCLUDE FILES

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

SEE ALSO

etherMultiLib

evbNs16550Sio

NAME

evbNs16550Sio – NS16550 serial driver for the IBM PPC403GA evaluation

ROUTINES

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

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

SEE ALSO

evbNs16550Sio

excArchLib

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

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

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

INCLUDE FILES **excLib.h**

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

excLib

NAME **excLib** – generic exception handling facilities

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

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

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

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

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

NORMAL EXCEPTION HANDLING

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

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

ADDITIONAL EXCEPTION HANDLING ROUTINE

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

TASK-LEVEL SUPPORT

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

DBGLIB

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

SIGLIB

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

INCLUDE FILES

excLib.h

SEE ALSO

dbgLib, **sigLib**, **intLib**

fei82557End

NAME

fei82557End – END style Intel 82557 Ethernet network interface driver

ROUTINES

***fei82557EndLoad*( )** – initialize the driver and device

DESCRIPTION

This module implements an Intel 82557 Ethernet network interface driver. This is a fast Ethernet PCI bus controller, IEEE 802.3 10Base-T and 100Base-T compatible. It also features a glueless 32-bit PCI bus master interface, fully compliant with PCI Spec version 2.1. An interface to MII compliant physical layer devices is built-in in the card. The 82557 Ethernet PCI bus controller also includes Flash support up to 1 MByte and EEPROM support, altough these features are not dealt with in the driver.

The 82557 establishes a shared memory communication system with the CPU, which is divided into three parts: the Control/Status Registers (CSR), the Command Block List (CBL) and the Receive Frame Area (RFA). The CSR is on chip and is either accessible with I/O or memory cycles, whereas the other structures reside on the host.

The CSR is the main meance of communication between the device and the host, meaning that the latter issues commands through these registers while the chip posts status changes in it, occurred as a result of those commands. Pointers to both the CBL and RFA are also stored in the CSR.

The CBL consists of a linked list of frame descriptors through which individual action commands can be performed. These may be transmit commands as well as non-transmit commands, e.g. Configure or Multicast setup commands. While the CBL list may function in two different modes, only the simplified memory mode is implemented in the driver.

The RFA is a linked list of receive frame descriptors. Only support for the simplified memory mode is granted. In this model, the data buffer immediately follows the related frame descriptor.

The 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.

BOARD LAYOUT

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

EXTERNAL INTERFACE

The driver provides the standard external interface, ***fei82557EndLoad*( )**, which takes a string of colon separated parameters. The parameters should be specified in hexadecimal, optionally preceeded by "0x" or a minus sign "-".

The parameter string is parsed using ***strtok_r*( )** and each parameter is converted from a string representation to binary by a call to strtoul(parameter, NULL, 16).

The format of the parameter string is:

"*memBase*:*memSize*:*nTfds*:*nRfds*:*flags*"

In addition, the two global variables **feiEndIntConnect** and **feiEndIntDisconnect** specify respectively the interrupt connect routine and the interrupt disconnect routine to be used depending on the BSP. The former defaults to ***intConnect*( )** and the user can override this to use any other interrupt connect routine (say ***pciIntConnect*( )**) in ***sysHwInit*( )** or any device specific initialization routine called in ***sysHwInit*( )**. Likewise, the latter is set by default to NULL, but it may be overridden in the BSP in the same way.

TARGET-SPECIFIC PARAMETERS

memBase

This parameter is passed to the driver via ***fei82557EndLoad*( )**.

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 will allocate cache safe memory for its use using ***cacheDmaAlloc*( )**.

memSize

The memory size parameter specifies the size of the pre-allocated memory region. If memory base is specified as NONE (-1), the driver ignores this parameter. Otherwise, the driver checks the size of the provoded memory region is adequate with respect to the given number of Command Frame Descriptor and Receive Frame Descriptor.

nTfds

This parameter specifies the number of transmit descriptor/buffers to be allocated. If this parameter is less than two, a default of 32 is used.

nRfds

This parameter specifies the number of receive descriptor/buffers to be allocated. If this parameter is less than two, a default of 32 is used.

flags

User flags may control the run-time characteristics of the Ethernet chip. Not implemented.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires one external support function:

```
STATUS sys557Init (int unit, FEI_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 calls ***cacheDmaMalloc*( )** to allocate memory to share with the 82557. The size of this area is affected by the configuration parameters specified by ***fei82557EndLoad*( )**.

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.

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 ***fei82557EndLoad*( )** 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 number of TFDs 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, Intel 32-bit Local Area Network (LAN) Component User's Manual*

---

fioLib

NAME

fioLib – formatted I/O library

ROUTINES

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

DESCRIPTION

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

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

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

INCLUDE FILES

fioLib.h, **stdio.h**

SEE ALSO

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

---

floatLib

NAME

floatLib – floating-point formatting and scanning library

ROUTINES

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

DESCRIPTION

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

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

INCLUDE FILES

math.h

SEE ALSO

fioLib

fppArchLib

NAME

fppArchLib – architecture-dependent floating-point coprocessor support

ROUTINES

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

DESCRIPTION

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

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

INITIALIZATION

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

NOTE I386/I486 On this architecture, VxWorks disables the six FPU exceptions that can send an IRQ to the CPU.

NOTE ARM

This architecture does not currently support floating-point coprocessors.

INCLUDE FILES

fppLib.h

SEE ALSO

fppLib, ***intConnect*( )**, *Motorola MC68881/882 Floating-Point Coprocessor User's Manual*, *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

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

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

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

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

VX_FP_TASK OPTION

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

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

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

INCLUDE FILES **fppLib.h**

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

fppShow

NAME

fppShow – floating-point show routines

ROUTINES

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

DESCRIPTION

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

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

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

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

INCLUDE FILES

fppLib.h

SEE ALSO

fppLib

ftpdLib

NAME

ftpdLib – File Transfer Protocol (FTP) server

ROUTINES

***ftpdInit*( )** – initialize the FTP server task
***ftpdDelete*( )** – terminate the FTP server task

DESCRIPTION

This library implements the server side of the File Transfer Protocol (FTP), which provides remote access to the file systems available on a target. The protocol is defined in RFC 959. This implementation supports all commands required by that specification, as well as several additional commands.

USER INTERFACE

During system startup, the ***ftpdInit*( )** routine creates a control connection at the predefined FTP server port which is monitored by the primary FTP task. Each FTP session established is handled by a secondary server task created as necessary. The server accepts the following commands:

HELP	– List supported commands.
USER	– Verify user name.

PASS	– Verify password for the user.
QUIT	– Quit the session.
LIST	– List out contents of a directory.
NLST	– List directory contents using a concise format.
RETR	– Retrieve a file.
STOR	– Store a file.
CWD	– Change working directory.
TYPE	– Change the data representation type.
PORT	– Change the port number.
PWD	– Get the name of current working directory.
STRU	– Change file structure settings.
MODE	– Change file transfer mode.
ALLO	– 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

The ***ftpdDelete*( )** routine will disable the FTP server until restarted. It reclaims all system resources used by the server tasks and cleanly terminates all active sessions.

INCLUDE FILES **ftpdLib.h**

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

ftpLib

NAME **ftpLib** – File Transfer Protocol (FTP) library

ROUTINES

***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
***ftpLs*( )** – list directory contents via FTP

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

ROUTINES
***hostTblInit*( )** – initialize the network host table
***hostAdd*( )** – add a host to the host table
***hostDelete*( )** – delete a host from the host table
***hostGetByName*( )** – look up a host in the host table by its name
***hostGetByAddr*( )** – look up a host in the host table by its Internet address
***sethostname*( )** – set the symbolic name of this machine
***gethostname*( )** – get the symbolic name of this machine

DESCRIPTION
This library provides routines to store and access the network host database. The host table contains information regarding the known hosts on the local network. The host table (displayed with ***hostShow*( )**) contains the Internet address, the official host name, and aliases.

By convention, network addresses are specified in dotted (".") decimal notation. The library **inetLib** contains Internet address manipulation routines. Host names and aliases may contain any printable character.

Before any of the routines in this module can be used, the library must be initialized by ***hostTblInit*( )**. This is done automatically if the configuration macro **INCLUDE_NET_INIT** is defined.

INCLUDE FILES
hostLib.h

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

i8250Sio

NAME
i8250Sio – I8250 serial driver

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

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.

This driver handles setting of hardware options such as parity(odd, even) and number of data bits(5, 6, 7, 8). Hardware flow control is provided with the handshakes RTS/CTS. The function HUPCL(hang up on last close) is available.

INCLUDE FILES

drv/sio/i8250Sio.h

icmpShow

NAME

icmpShow – ICMP Information display routines

ROUTINES

***icmpShowInit*( )** – initialize ICMP show routines
***icmpstatShow*( )** – display statistics for ICMP

DESCRIPTION

This library provides routines to show ICMP related statistics.

Interpreting these statistics requires detailed knowledge of Internet network protocols. Information on these protocols can be found in the following books:

– *TCP/IP Illustrated Volume II, The Implementation,* by Richard Stevens

– *The Design and Implementation of the 4.4 BSD UNIX Operating System*, by Leffler, McKusick, Karels and Quarterman

The ***icmpShowInit*( )** routine links the ICMP show facility into the VxWorks system. This is performed automatically if **INCLUDE_NET_SHOW** is defined in **configAll.h**.

SEE ALSO

netLib, **netShow**, *Network Programmer's Guide*

ideDrv

NAME

ideDrv – IDE disk device driver

ROUTINES

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

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

ROUTINES

***ifAddrAdd*( )** – Add an interface address for a network interface
***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

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_cpm

NAME **if_cpm** – Motorola CPM core network interface driver

ROUTINES ***cpmattach*( )** – publish the **cpm** network interface and initialize the driver
***cpmStartOutput*( )** – output packet to network interface device

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 ***cpmInit( )*** 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_cs

NAME **if_cs** – Crystal Semiconductor CS8900 network interface driver

ROUTINES
***csAttach*( )** – publish the **cs** network interface and initialize the driver.
***csShow*( )** – shows statistics for the **cs** network interface

DESCRIPTION This module implements a driver for a Crystal Semiconductor CS8900 Ethernet controller chip.

The CS8900 is a single chip Ethernet controller with a direct ISA bus interface which can operate in either memory space or I/O space. It also supports a direct interface to a host DMA controller to transfer receive frames to host memory. The device has a 4K RAM which is used for transmit, and receive buffers; a serial EEPROM interface; and both 10BASE-T/AUI port support.

This driver is capable of supporting both memory mode and I/O mode operations of the chip. When configured for memory mode, the intenal RAM of the chip is mapped to a contiguous 4K address block, providing the CPU direct access to the internal registers and frame buffers. When configured for I/O mode, the internal registers are accessible through eight contiguous, 16-bit I/O ports. The driver also supports an interface to an EEPROM containing device configuration.

While the DMA slave mode is supported by the device for receive frame transfers, this driver does not enable DMA.

This network interface driver does not support output hook routines, because to do so requires that an image of the transmit packet be built in memory before the image is copied to the CS8900 chip. It is much more efficient to copy the image directly from the mbuf chain to the CS8900 chip. However, this network interface driver does support input hook routines.

CONFIGURATION

The defined I/O address and IRQ in **config.h** must match the one stored in EEPROM by the vendor's DOS utility program.

The I/O Address parameter is the only required ***csAttach*( )** parameter. If the CS8900 chip has a EEPROM attached, then the I/O Address parameter, passed to the ***csAttach*( )** routine, must match the I/O address programmed into the EEPROM. If the CS8900 chip does not have a EEPROM attached, then the I/O Address parameter must be 0x300.

The Interrupt Level parameter must have one of the following values:
0 – Get interrupt level from EEPROM
5 – IRQ 5
10 – IRQ 10
11 – IRQ 11
12 – IRQ 12

If the Interrupt Vector parameter is zero, then the network interface driver derives the interrupt vector from the interrupt level if possible. It is possible to derive the interrupt vector in an IBM PC compatible system. This parameter is present for systems which are not IBM PC compatible.

The Memory Address parameter specifies the base address of the CS8900 chip's memory buffer (PacketPage). If the Memory Address parameter is not zero, then the CS8900 chip operates in memory mode at the specified address. If the Memory Address parameter is zero, then the CS8900 chip operates in the mode specified by the EEPROM or the Configuration Flags parameter.

The Media Type parameter must have one of the following values:

```
0 – Get media type from EEPROM
1 – AUI  (Thick Cable)
2 – BNC  10Base2 (Thin Cable)
3 – RJ45 10BaseT (Twisted Pair)
```

The Configuration Flags parameter is usually passed to the ***csAttach*( )** routine as zero and the Configuration Flags information is retrieved from the EEPROM. The bits in the Configuration Flags parameter are usually specified by a hardware engineer and not by the end user. However, if the CS8900 chip does not have a EEPROM attached, then this information must be passed as a parameter to the ***csAttach*( )** routine. The Configuration Flags are:

```
0x8000 – CS_CFGFLG_NOT_EEPROM    Don't get Config. Flags from the EEPROM
```

```
0x0001 – CS_CFGFLG_MEM_MODE      Use memory mode to access the chip
0x0002 – CS_CFGFLG_USE_SA        Use system addr to qualify MEMCS16 signal
0x0004 – CS_CFGFLG_IOCHRDY       Use IO Channel Ready signal to slow access
0x0008 – CS_CFGFLG_DCDC_POL      The DC/DC conv. enable pin is active high
0x0010 – CS_CFGFLG_FDX           10BaseT is full duplex
```

If configuration flag information is passed to the ***csAttach*( )** routine, then the **CS_CFGFLG_NOT_EEPROM** flag should be set. This ensures that the Configuration Flags parameter is not zero, even if all specified flags are zero.

If the Memory Address parameter is not zero and the Configuration Flags parameter is zero, then the CS8900 network interface driver implicitly sets the **CS_CFGFLG_MEM_MODE** flag and the CS8900 chip operates in memory mode. However, if the Configuration Flags parameter is not zero, then the CS8900 chip operates in memory mode only if the **CS_CFGFLG_MEM_MODE** flag is explicitly set. If the Configuration Flags parameter in not zero and the **CS_CFGFLG_MEM_MODE** flag is not set, then the CS8900 chip operates in I/O mode.

The Ethernet Address parameter is usually passed to the ***csAttach*( )** routine as zero and the Ethernet address is retrieved from the EEPROM. The Ethernet address (also called hardware address and individual address) is usually supplied by the adapter manufacturer and is stored in the EEPROM. However, if the CS8900 chip does not have a EEPROM attached, then the Ethernet address must be passed as a parameter to the ***csAttach*( )** routine. The Ethernet Address parameter, passed to the ***csAttach*( )** routine, contains the address of a NULL terminated string. The string consists of 6 hexadecimal numbers separated by colon characters. Each hexadecimal number is in the range 00 – FF. An example of this string is:

```
"00:24:20:10:FF:2A"
```

BOARD LAYOUT

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

EXTERNAL INTERFACE

The only user-callable routines are ***csAttach*( )**:

***csAttach*( )**
: publishes the **cs** interface and initializes the driver and device.

The network interface driver includes a show routine, called ***csShow*( )**, which displays driver configuration and statistics information. To invoke the show routine, type at the shell prompt:

```
-> csShow
```

To reset the statistics to zero, type at the shell prompt:

```
-> csShow 0, 1
```

Another routine that you may find useful is:

```
-> ifShow "cs0"
```

EXTERNAL ROUTINES

For debugging purposes, this driver calls ***logMsg*( )** to print error and debugging information. This will cause the **logLib** library to be linked with any image containing this driver.

This driver needs the following macros defined for proper execution. Each has a default definition that assumes a PC386/PC486 system and BSP.

The macro **CS_IN_BYTE** (reg,pAddr) reads one byte from the I/O address **reg**, placing the result at address **pAddr**. There is no status result from this operation, we assume the operation completes normally, or a bus exception will occur. By default, this macro assumes there is a BSP routine ***sysInByte*( )** to perform the I/O operation.

The macro **CS_IN_WORD** (reg,pAddr) read a short word (2 bytes) from the I/O address **reg**, storing the result at address **pAddr**. We assume this completes normally, or causes a bus exception. The default declaration assumes a BSP routine ***sysInWord*( )** to perform the operation.

The macro **CS_OUT_WORD** (reg,data) writes a short word value **data** at the I/O address **reg**. The default declaration assumes a BSP routine ***sysOutWord*( )**.

The macro **CS_INT_ENABLE** (level, pResult) is used to enable the interrupt level passed as an argument to csAttach. The default definition call the BSP routine sysIntEnablePIC(level). The STATUS return value from the actual routine is stored at **pResult** for the driver to examine.

The macro **CS_INT_CONNECT** (ivec,rtn,arg,pResult) macro is used to connect the driver interrupt routine to the vector provided as an argument to csAttach (after translaction by **INUM_TO_IVEC**). The default definition calls the cpu architecture routine ***intConnect*( )**.

The macro **CS_IRQ0_VECTOR** (pAddr) is used to fetch the base vector for the interrupt level mechanism. If the int vector argument to csAttach is zero, then the driver will compute a vector number by adding the interrupt level to the value returned by this macro. If the user supplies a non-zero interrupt vector number, then this macro is not used. The default definition of this macro fetches the base vector number from a global value called **sysVectorIRQ0**.

The macro **CS_MSEC_DELAY** (msec) is used to delay execution for a specified number of milliseconds. The default definition uses taskDelay to suspend task for some number of clock ticks. The resolution of the system clock is usually around 16 milliseconds (msecs), which is fairly coarse.

1

if_dc

NAME

if_dc – DEC 21x4x Ethernet LAN network interface driver

ROUTINES

***dcattach*( )** – publish the **dc** network interface.
***dcReadAllRom*( )** – read entire serial rom
***dcViewRom*( )** – display lines of serial ROM for dec21140
***dcCsrShow*( )** – display dec 21040/21140 status registers 0 thru 15

DESCRIPTION

This module implements an ethernet interface driver for the DEC 21x4x family, and currently supports the following variants -- 21040, 21140, and 21140A.

The DEC 21x4x PCI Ethernet controllers are inherently little-endian since they are designed for a little-endian PCI bus. While the 21040 only supports a 10Mps interface, other members of this family are dual-speed devices which support both 10 and 100 Mbps.

This driver is designed to be moderately generic, operating unmodified across the range of architectures and targets supported by VxWorks; and on multiple versions of the dec21x4x family. To achieve this, the driver takes several parameters, and external support routines which are detailed below. Also stated below are assumptions made by the driver of the hardware, and if any of these assumptions are not true for your hardware, the driver will probably not function correctly.

This driver supports up to 4 ethernet units per CPU, and can be configured for either big-endian or little-endian architectures. It contains error-recovery code to handle known device errata related to DMA activity.

On a dec21040, 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.

The dec21140, and dec21140A devices support both 10 and 100Mbps and also a variety of MII and non-MII PHY interfaces. This driver reads a DEC version 2.0 SROM device for PHY initialization information, and automatically configures an apropriate active PHY media.

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 device 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 or DEC21140 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 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 device 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 device 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 DEC 21x4x 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 DEC 21x4x. 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 DEC 21x4x device. This should be done on hardware that restricts the DEC 21x4x device 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 explicitly limit the amount of shared memory (bytes) this driver will use. 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 DEC 21x4x 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 for both the DEC21040 and DEC21140 interfaces are listed below.

DC_PROMISCUOUS_FLAG 0x01
DC_MULTICAST_FLAG 0x02

The mode flags specific to the DEC21140 interface are listed below.

DC_100_MB_FLAG 0x04
DC_21140_FLAG 0x08
DC_SCRAMBLER_FLAG 0x10
DC_PCS_FLAG 0x20
DC_PS_FLAG 0x40
DC_FULLDUPLEX_FLAG 0x10

Loopback mode flags:

DC_ILOOPB_FLAG 0x100
DC_ELOOPB_FLAG 0x200
DC_HBE_FLAG 0x400

Ethernet address
This is obtained by the driver by reading an ethernet ROM register or the DEC serial ROM.

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 DEC 21x4x device. Typically, this involves interrupt controller hardware, either internal or external to the CPU.

This routine is called once via the macro **SYS_INT_ENABLE**.

SEE ALSO **ifLib**, *DECchip 21040 or 21140 Ethernet LAN Controller for PCI.*

if_eex

NAME **if_eex** – Intel EtherExpress 16 network interface driver

ROUTINES
***eexattach*( )** – publish the **eex** network interface and initialize the driver and device
***eexTxStartup*( )** – start output on the chip

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

ROUTINES

***eiattach*( )** – publish the **ei** network interface and initialize the driver and device
***eiTxStartup*( )** – start output on the chip

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 number of TFDs 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.

CAVEAT

This driver does not support promiscuous mode.

SEE ALSO

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

if_eihk

NAME

if_eihk – Intel 82596 Ethernet network interface driver for hkv3500

ROUTINES

***eihkattach*( )** – publish the **ei** network interface and initialize the driver and device
***eiTxStartup*( )** – start output on the chip
***eiInt*( )** – entry point for handling interrupts from the 82596

DESCRIPTION

This module implements a hkv3500 specfic Intel 82596 Ethernet network interface driver.

This driver is derived from the generic if_ei ethernet driver to support hkv3500 target board. The receive buffer scheme has been modified from a simplified memory structure to a flexible memory structure so that receive buffers can be word-aligned, and thus support buffer loaning on a MIPS CPU architecture.

The driver requires several target-specific parameters, and some external support routines which 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 ***eihkattach*( )**, 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 ***eihkattach*( )**.

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 ***eihkattach*( )**.

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 ***eihkattach*( )**.

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 ***eihkattach*( )**.

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 ***eihkattach*( )**.

`STATUS sys596Init (int unit, SCB *pScb)`
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 ***eihkattach*( )**.

`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 ***eihkattach*( )** 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 ***eihkattach*( )** 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 number of TFDs 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 82596 User's Manual*, *Intel 32-bit Local Area Network (LAN) Component User's Manual*

if_elc

NAME

if_elc – SMC 8013WC Ethernet network interface driver

ROUTINES

***elcattach*( )** – publish the **elc** network interface and initialize the driver and device
***elcPut*( )** – copy a packet to the interface.
***elcShow*( )** – display statistics for the SMC 8013WC **elc** network interface

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.

SEE ALSO

if_elc

if_elt

NAME

if_elt – 3Com 3C509 Ethernet network interface driver

ROUTINES

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

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

ROUTINES
***eneattach*( )** – publish the **ene** network interface and initialize the driver and device
***enePut*( )** – copy a packet to the interface.
***eneShow*( )** – display statistics for the NE2000 **ene** network interface

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.

```
 ________________________________________________________
|                                                        |
|                                                        |
|                                           WWWWWWWW     |
|                       WWWW  WWW           87654321     ||
|                       1111   11         1 ........     ||
|                       5432  901         2 ........     ||
|                       ....  ...         3 ........     ||
|                       ....  ...                        ||
|               W                                        |
|               1                                        |
|               6                                        |
|               .                                        |___
|               .                                        |___|
|                                                        |
|________              ___                          ____|
        |              | |                          |
        |______________| |__________________________|
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_esmc

NAME **if_esmc** – Ampro Ethernet2 SMC-91c9x Ethernet network interface driver

ROUTINES

***esmcattach*( )** – publish the **esmc** network interface and initialize the driver.
***esmcPut*( )** – copy a packet to the interface.
***esmcShow*( )** – display statistics for the **esmc** network interface

DESCRIPTION

This module implements the Ampro Ethernet2 SMC-91c9x Ethernet network interface driver.

CONFIGURATION

The W3 and W4 jumper should be set for IO address and IRQ. The defined I/O address and IRQ in **config.h** must match the one stored in EEROM and the jumper setting.

BOARD LAYOUT

The diagram below shows the relevant jumpers for VxWorks configuration.

```
  __________________________________
 |             * * * *              |
 |  ______                          |
 |  |    |                          |
 |  | U1 | W1  W3                   |
 |  |PROM| X   "                    |
 |  |    | .  -                   |
 |  |    |    -                   |
 |  |    |    -                   |
 |  |____|                          |
 |                                  |
 |                W4                |
 |                "                 |
 |                "                 |
 |               -              |
 |               -              |
 |__________________________________|
W1:  Boot PROM Size
W3:  IO-address, IRQ, Media
W4:  IRQ Group Selection
```

EXTERNAL INTERFACE

The only user-callable routines are ***esmcattach*( )**:

***esmcattach*( )**
: publishes the **esmc** interface and initializes the driver and device.

The last parameter of ***esmcattach*( )**, *mode*, is a receive mode. If it is 0, a packet is received in the interrupt level. If it is 1, a packet is received in the task level. Receiving packets in the interrupt level requires about 10K bytes of memory, but minimize a risk of dropping packets. Receiving packets in the task level doesn't require extra memory, but might have a risk of dropping packets.

if_fei

NAME

if_fei – Intel 82557 Ethernet network interface driver

ROUTINES

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

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 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.

There is a global variable **feiIntConnect** which specifies the interrupt connect routine to be used depending on the BSP. This is by default set to ***intConnect*( )** and the user can override this to use any other interrupt connect routine (say ***pciIntConnect*( )**) in ***sysHwInit*( )** or any device specific initialization routine called in ***sysHwInit*( )**.

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 the following external support function:

```
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

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

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 network interface driver

ROUTINES

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

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 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 ***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( )***.

This parameter can be used to explicitly limit the amount of shared memory (bytes) this driver will use. 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 ***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.

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 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 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_lnPci

NAME

if_lnPci – AMD Am79C970 PCnet-PCI Ethernet network interface driver

ROUTINES

***lnPciattach*( )** – publish the **lnPci** network interface and initialize the driver and device

DESCRIPTION

This module implements the Advanced Micro Devices Am79C970 PCnet-PCI Ethernet 32 bit network interface driver.

The PCnet-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 in the driver. The second part of the interface comprises of the I/O control registers and their programming. The third part of the interface comprises of the descriptors and the 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 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.

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 which the chip DMA s to have to be swapped and written to, so that when the hardware swaps the accesses, the chip would see them correctly. The chip still has to be programmed to operated 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 input and output byte stream 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 ***lnPciattach*( )**, which publishes the **lnPci**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_long lnPciCSR_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 Am79C970 (PCnet-PCI).*

base address of device registers

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

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 ***lnPciattach*( )**.

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 turn on LANCE interrupt generation. A similar routine, ***sysLanIntDisable*( )**, is called by the driver before a LANCE reset to perform board-specific operations required to turn off LANCE interrupt generation. For a description of ***sysLanIntEnable*( )**, and ***sysLanIntDisable*( )**, see "External Support Requirements" below.

This parameter is passed to the external routine.

shared memory address

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

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 ***lnPciattach*( )**.

This parameter can be used to explicitly limit the amount of shared memory (bytes) this driver will use. 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 ***lnPciattach*( )**.

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 ***lnPciattach*( )**.

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 programming an interrupt controller hardware, either internal or external to the CPU.

This routine is called during chip initialization, at startup and each LANCE device reset.

void sysLanIntDisable (int level)

This routine provides a target-specific disable of the interrupt for the LANCE device. Typically, this involves programming an interrupt controller hardware, either internal or external to the CPU.

This routine is called before a LANCE device reset.

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 ***lnPciattach*( )** 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 PCnet-PCI Ethernet Controller for PCI.*

if_loop

NAME **if_loop** – software loopback network interface driver

ROUTINES ***loattach*( )** – publish the **lo** network interface and initialize the driver and pseudo-device

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

ROUTINES
***mbcattach*( )** – publish the **mbc** network interface and initialize the driver
***mbcStartOutput*( )** – output packet to network interface device
***mbcIntr*( )** – network interface interrupt handler

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_nicEvb

NAME

if_nicEvb – National Semiconductor ST-NIC Chip network interface driver

ROUTINES

***nicEvbattach*( )** – publish and initialize the **nicEvb** network interface driver
***nicTxStartup*( )** – the driver's actual output routine

DESCRIPTION

This module implements the National Semiconductor 83902A ST-NIC Ethernet network interface driver.

This driver is non-generic and is for use on the IBM EVB403 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 ***nicEvbattach*( )**, which publishes the **nicEvb**interface and initializes the driver and device.

TARGET-SPECIFIC PARAMETERS

device I/O address
: This parameter is passed to the driver by ***nicEvbattach*( )**. It specifies the base address of the device's I/O register set.

interrupt vector
: This parameter is passed to the driver by ***nicEvbattach*( )**. It specifies the interrupt vector to be used by the driver to service an interrupt from the ST-NIC device. The driver will connect the interrupt handler to this vector by calling ***intConnect*( )**.

device restart/reset delay
: The global variable nicRestartDelay (UINT32), defined in this file, should be initialized in the BSP ***sysHwInit*( )** routine. nicRestartDelay is used only with PowerPC platform and is equal to the number of time base increments which makes for 1.6 msec. This corresponds to the delay necessary to respect when restarting or resetting the device.

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 ***nicEnetAddrGet( )*** routine.

SYSTEM RESOURCE USAGE

When implemented, this driver requires the following system resources:

– one mutual exclusion semaphore
– one interrupt vector

SEE ALSO

ifLib

if_sl

NAME

if_sl – Serial Line IP (SLIP) network interface driver

ROUTINES

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

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.

TARGET-SPECIFIC PARAMETERS

The global flag slipLoopBack is set to 1 by default. This flag enables the packets to be sent to the loopback interface if they are destined to to a local slip interface address. By setting this flag, any packets sent to a local slip interface address will not be seen on the actual serial link. Set this flag to 0 to turn off this facility. If this flag is not set any packets sent to the local slip interface address will actually be sent out on the link and it is the peer's responsibility to loop the packet back.

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

ROUTINES ***smIfAttach*( )** – publish the **sm** interface and initialize the driver and device

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 driver

ROUTINES ***snattach*( )** – publish the **sn** network interface and initialize the driver and device

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.

NOTE 1

The previous transmit descriptor does not exist until the transmitter has been asked to send at least one packet. Unfortunately the test for this condition must be done every time a new descriptor is to be added, even though the condition is only true the first time. However, it is a valuable test, since we should not use the fragment count field as an index if it is 0.

NOTE 2

There are some things unsupported in this version:

a) buffer loaning on receive
b) output hooks
c) trailer protocol
d) promiscuous mode

Also, the receive setup needs work so that the number of RRA descriptors is not fixed at four. It would be a nice addition to allow all the sizes of the shared memory structures to be specified by the runtime functions that call our init routines.

SEE ALSO

ifLib

if_ulip

NAME

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

ROUTINES

***ulipInit*( )** – initialize the ULIP interface (VxSim)
***ulattach*( )** – attach a ULIP interface to a list of network interfaces (VxSim)
***ulipDelete*( )** – delete a ULIP interface (VxSim)
***ulStartOutput*( )** – push packets onto "interface"
***ulipDebugSet*( )** – Set debug flag in UNIX's ULIP driver

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) 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

VxWorks Programmer's Guide: VxSim

if_ultra

NAME

if_ultra – SMC Elite Ultra Ethernet network interface driver

ROUTINES

***ultraattach*( )** – publish **ultra** interface and initialize device
***ultraPut*( )** – copy a packet to the interface.
***ultraShow*( )** – display statistics for the **ultra** network interface

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.

igmpShow

NAME

igmpShow – IGMP information display routines

ROUTINES

***igmpShowInit*( )** – initialize IGMP show routines
***igmpstatShow*( )** – display statistics for IGMP

DESCRIPTION

This library provides routines to show IGMP related statistics.

Interpreting these statistics requires detailed knowledge of Internet network protocols. Information on these protocols can be found in the following books:

– *TCP/IP Illustrated Volume II, The Implementation,* by Richard Stevens

– *The Design and Implementation of the 4.4 BSD UNIX Operating System,* by Leffler, McKusick, Karels and Quarterman

The ***igmpShowInit*( )** routine links the IGMP show facility into the VxWorks system. This is performed automatically if **INCLUDE_NET_SHOW** is defined in **configAll.h**.

SEE ALSO **netLib**, **netShow**, *Network Programmer's Guide*

inetLib

NAME **inetLib** – Internet address manipulation routines

ROUTINES
***inet_addr*( )** – convert a dot notation Internet address to a long integer
***inet_lnaof*( )** – get the local address (host number) from the Internet address
***inet_makeaddr_b*( )** – form an Internet address from network and host numbers
***inet_makeaddr*( )** – form an Internet address from network and host numbers
***inet_netof*( )** – return the network number from an Internet address
***inet_netof_string*( )** – extract the network address in dot notation
***inet_network*( )** – convert an Internet network number from string to address
***inet_ntoa_b*( )** – convert an network address to dot notation, store it in a buffer
***inet_ntoa*( )** – convert a network address to dotted decimal notation
***inet_aton*( )** – convert a network address from dot notation, store in a structure

DESCRIPTION
This library provides routines for manipulating Internet addresses, including the UNIX BSD 4.3 **inet_** routines. It includes routines for converting between character addresses in Internet standard dotted decimal notation and integer addresses, routines for extracting the network and host portions out of an Internet address, and routines for constructing Internet addresses given the network and host address parts.

All Internet addresses are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine format integer values.

INTERNET ADDRESSES
Internet addresses are typically specified in dotted decimal notation or as a 4-byte number. Values specified using the dotted decimal notation take one of the following forms:

```
a.b.c.d
a.b.c
a.b
a
```

If four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address. Note that when an Internet address is viewed as a 32-bit integer quantity on any MC68000 family machine, the bytes referred to above appear as "**a.b.c.d**" and are ordered from left to right.

If a three-part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right-most two bytes of the network address. This makes the three-part address format convenient for specifying Class B network addresses as "128.net.host".

If a two-part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right-most three bytes of the network address. This makes the two-part address format convenient for specifying Class A network addresses as "net.host".

If only one part is given, the value is stored directly in the network address without any byte rearrangement.

Although dotted decimal notation is the default, it is possible to use the dot notation with hexadecimal or octal numbers. The base is indicated using the same prefixes as are used in C. That is, a leading 0x or 0X indicates a hexadecimal number. A leading 0 indicates an octal number. If there is no prefix, the number is interpreted as decimal.

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

ROUTINES *inflate*( ) – inflate compressed code

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

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

intArchLib

NAME

intArchLib – architecture-dependent interrupt library

ROUTINES

***intLevelSet*( )** – set the interrupt level (MC680x0, SPARC, i960, x86, ARM)
***intLock*( )** – lock out interrupts
***intUnlock*( )** – cancel interrupt locks
***intEnable*( )** – enable corresponding interrupt bits (MIPS, PowerPC, ARM)
***intDisable*( )** – disable corresponding interrupt bits (MIPS, PowerPC, ARM)
***intCRGet*( )** – read the contents of the cause register (MIPS)
***intCRSet*( )** – write the contents of the cause register (MIPS)
***intSRGet*( )** – read the contents of the status register (MIPS)
***intSRSet*( )** – update the contents of the status register (MIPS)
***intConnect*( )** – connect a C routine to a hardware interrupt
***intHandlerCreate*( )** – construct ISR for a C routine (MC680x0, SPARC, i960, x86, MIPS)
***intLockLevelSet*( )** – set current interrupt lock-out level (MC680x0, SPARC, i960, x86, ARM)
***intLockLevelGet*( )** – get current interrupt lock-out level (MC680x0, SPARC, i960, x86, ARM)
***intVecBaseSet*( )** – set vector (trap) base address (MC680x0, SPARC, i960, x86, MIPS, ARM)
***intVecBaseGet*( )** – get vector (trap) base address (MC680x0, SPARC, i960, x86, MIPS, ARM)
***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, ARM)
***intUninitVecSet*( )** – set the uninitialized vector handler (ARM)

DESCRIPTION

This library provides architecture-dependent routines to manipulate and connect to hardware interrupts. Any C language routine can be connected to any interrupt by calling ***intConnect*( )**. Vectors can be accessed directly by ***intVecSet*( )** and ***intVecGet*( )**. The vector (trap) base register (if present) can be accessed by the routines ***intVecBaseSet*( )** and ***intVecBaseGet*( )**.

Tasks can lock and unlock interrupts by calling ***intLock*( )** and ***intUnlock*( )**. The lock-out level can be set and reported by ***intLockLevelSet*( )** and ***intLockLevelGet*( )** (MC680x0, SPARC, i960, i386/i486 and ARM only). The routine ***intLevelSet*( )** changes the current interrupt level of the processor (MC680x0, SPARC, i960 and ARM).

WARNING

Do not call VxWorks system routines with interrupts locked. Violating this rule may re-enable interrupts unpredictably.

INTERRUPT VECTORS AND NUMBERS

Most of the routines in this library take an interrupt vector as a parameter, which is generally the byte offset into the vector table. Macros are provided to convert between interrupt vectors and interrupt numbers:

IVEC_TO_INUM (intVector)
converts a vector to a number.

INUM_TO_IVEC (intNumber)
converts a number to a vector.

TRAPNUM_TO_IVEC (trapNumber)
converts a trap number to a vector.

EXAMPLE

To switch between one of several routines for a particular interrupt, the following code fragment is one alternative:

```
vector  = INUM_TO_IVEC(some_int_vec_num);
oldfunc = intVecGet (vector);
newfunc = intHandlerCreate (routine, parameter);
intVecSet (vector, newfunc);
...
intVecSet (vector, oldfunc);    /* use original routine */
...
intVecSet (vector, newfunc);    /* reconnect new routine */
```

INCLUDE FILES

iv.h, **intLib.h**

SEE ALSO

intLib

---

intLib

NAME

intLib – architecture-independent interrupt subroutine library

ROUTINES

***intContext*( )** – determine if the current state is in interrupt or task context
***intCount*( )** – get the current interrupt nesting depth

DESCRIPTION

This library provides generic routines for interrupts. Any C language routine can be connected to any interrupt (trap) by calling ***intConnect*( )**, which resides in **intArchLib**. The ***intCount*( )** and ***intContext*( )** routines are used to determine whether the CPU is running in an interrupt context or in a normal task context. For information about architecture-dependent interrupt handling, see the manual entry for **intArchLib**.

INCLUDE FILES

intLib.h

SEE ALSO

intArchLib, *VxWorks Programmer's Guide: Basic OS*

ioLib

NAME

ioLib – I/O interface library

ROUTINES

***creat*( )** – create a file
***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

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*

---

iOlicomEnd

NAME **iOlicomEnd** – END style Intel Olicom PCMCIA network interface driver

ROUTINES ***iOlicomEndLoad*( )** – initialize the driver and device
***iOlicomIntHandle*( )** – interrupt service for card interrupts

DESCRIPTION This module implements the Olicom (Intel 82595TX) network interface driver. The physical device is a PCMCIA card. This driver also houses code to manage a Vadem PCMCIA Interface controller on the ARM PID board, which is strictly a subsystem in it's own right.

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 maintains cache coherency by allocating buffer space using the ***cacheDmaMalloc*( )** routine.

BOARD LAYOUT The device resides on a PCMCIA card and is soft configured. No jumpering diagram is necessary.

EXTERNAL INTERFACE

This driver provides the END external interface with the following exceptions. The only external interface is the ***iOlicomEndLoad*( )** routine. All of the paramters are passed as strings in a colon (:) separated list to the load function as an initString. The ***iOlicomEndLoad*( )** function uses ***strtok*( )** to parse the string.

The string contains the target specific parameters like this:

"*io_baseA*:*attr_baseA*:*mem_baseA*:*io_baseB*:*attr_baseB*:*mem_baseB*: \
ctrl_base:*intVectA*:*intLevelA*:*intVectB*:*intLevelB*: \
txBdNum:*rxBdNum*:*pShMem*:*shMemSize*"

TARGET-SPECIFIC PARAMETERS

I/O base address A
: This is the first parameter passed to the driver init string. This parameter indicates the base address of the PCMCIA I/O space for socket A.

Attribute base address A
: This is the second parameter passed to the driver init string. This parameter indicates the base address of the PCMCIA attribute space for socket A. On the PID board, this should be the offset of the beginning of the attribute space from the beginning of the memory space.

Memory base address A
: This is the third parameter passed to the driver init string. This parameter indicates the base address of the PCMCIA memory space for socket A.

I/O base address B
: This is the fourth parameter passed to the driver init string. This parameter indicates the base address of the PCMCIA I/O space for socket B.

Attribute base address B
: This is the fifth parameter passed to the driver init string. This parameter indicates the base address of the PCMCIA attribute space for socket B. On the PID board, this should be the offset of the beginning of the attribute space from the beginning of the memory space.

Memory base address B
: This is the sixth parameter passed to the driver init string. This parameter indicates the base address of the PCMCIA memory space for socket B.

PCMCIA controller base address
: This is the seventh parameter passed to the driver init string. This parameter indicates the base address of the Vadem PCMCIA controller.

interrupt vectors and levels
: These are the eighth, ninth, tenth and eleventh parameters passed to the driver init string.

: The mapping of IRQs generated at the Card/PCMCIA level to interrupt levels and vectors is system dependent. Furthermore the slot holding the PCMCIA card is not initially known. The interrupt levels and vectors for both socket A and socket B must be passed to ***iOlicomEndLoad*( )**, allowing the driver to select the required parameters later.

number of transmit and receive buffer descriptors
: These are the twelfth and thirteenth parameters passed to the driver init string.

The number of transmit and receive buffer descriptors (BDs) used is configurable by the user upon attaching the driver. There must be a minimum of two transmit and two receive BDs, and there is a maximum of twenty transmit and twenty receive BDs. If this parameter is "NULL" a default value of 16 BDs will be used.

offset
: This is the fourteenth parameter passed to the driver in the init string.

  This parameter defines the offset which is used to solve alignment problem.

  base address of buffer pool

  This is the fifteenth parameter passed to the driver in the init string.

  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. If this parameter is "NONE", space for buffers will be obtained by calling ***cacheDmaMalloc*( )** in ***iOlicomEndLoad*( )**.

mem size of buffer pool
: This is the sixteenth parameter passed to the driver in the init string.

  The memory size parameter specifies the size of the pre-allocated memory region. If memory base is specified as NONE (-1), the driver ignores this parameter.

Ethernet address
: This parameter is obtained from the Card Information Structure on the Olicom PCMCIA card.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires three external support function:

`void sysLanIntEnable (int level)`
: This routine provides a target-specific interface for enabling Ethernet device interrupts at a specified interrupt level. This routine is called each time that the ***iOlicomStart*( )** routine is called.

`void sysLanIntDisable (int level)`
: This routine provides a target-specific interface for disabling Ethernet device interrupts. The driver calls this routine from the ***iOlicomStop*( )** routine each time a unit is disabled.

`void sysBusIntAck(void)`
: This routine acknowledge the interrupt if it's necessary.

SEE ALSO

muxLib, **endLib**, *Intel 82595TX ISA/PCMCIA High Integration Ethernet Controller User Manual, Vadem VG-468 PC Card Socket Controller Data Manual.*

ioMmuMicroSparcLib

NAME

ioMmuMicroSparcLib – microSparc I/II I/O DMA library

ROUTINES

***ioMmuMicroSparcInit*( )** – initialize the microSparc I/II I/O MMU data structures
***ioMmuMicroSparcMap*( )** – map the I/O MMU for microSparc I/II
(TMS390S10/MB86904)

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**

iosLib

NAME

iosLib – I/O system library

ROUTINES

***iosInit*( )** – initialize the I/O system
***iosDrvInstall*( )** – install an I/O driver
***iosDrvRemove*( )** – remove an I/O driver
***iosDevAdd*( )** – add a device to the I/O system
***iosDevDelete*( )** – delete a device from the I/O system
***iosDevFind*( )** – find an I/O device in the device list
***iosFdValue*( )** – validate an open file descriptor and return the driver-specific value

DESCRIPTION

This library is the driver-level interface to the I/O system. Its primary purpose is to route user I/O requests to the proper drivers, using the proper parameters. To do this, **iosLib** keeps tables describing the available drivers (e.g., names, open files).

The I/O system should be initialized by calling ***iosInit*( )**, before calling any other routines in **iosLib**. Each driver then installs itself by calling ***iosDrvInstall*( )**. The devices serviced by each driver are added to the I/O system with ***iosDevAdd*( )**.

The I/O system is described more fully in the *I/O System* chapter of the *Programmer's Guide.*

INCLUDE FILES

iosLib.h

SEE ALSO

intLib, **ioLib**, *VxWorks Programmer's Guide: I/O System*

iosShow

NAME **iosShow** – I/O system show routines

ROUTINES ***iosShowInit*( )** – initialize the I/O system show facility
***iosDrvShow*( )** – display a list of system drivers
***iosDevShow*( )** – display the list of devices in the system
***iosFdShow*( )** – display a list of file descriptor names in the system

DESCRIPTION This library contains I/O system information display routines.

The routine ***iosShowInit*( )** links the I/O system information show facility into the VxWorks system. It is called automatically when **INCLUDE_SHOW_ROUTINES** is defined in **configAll.h**.

SEE ALSO **intLib**, **ioLib**, *VxWorks Programmer's Guide: I/O System*, windsh, *Tornado User's Guide: Shell*

ipFilterLib

NAME **ipFilterLib** – ip filter hooks library

ROUTINES ***ipFilterLibInit*( )** – initialize ip filter facility
***ipFilterHookAdd*( )** – add a routine to receive all internet protocol packets
***ipFilterHookDelete*( )** – delete a ip filter hook routine

DESCRIPTION This library provides utilities that give direct access to IP packets. Incoming raw IP packets can be examined or processed using the hooks ***ipFilterHookAdd*( )**. The input hook can be used to receive raw IP packets that are a part of IP (Internet Protocol) protocols. The filter hook can also be used to build IP traffic monitoring and testing tools.

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

The ***ipFilterLibInit*( )** routine links the ip filtering facility into the VxWorks system. This is performed automatically if **INCLUDE_IP_FILTER** is defined in **configAll.h**.

SEE ALSO *VxWorks Programmer's Guide: Network*

ipProto

NAME **ipProto** – an interface between the BSD IP protocol and the MUX

ROUTINES ***ipAttach*( )** – a generic attach routine for the TCP/IP network stack
***ipDetach*( )** – a generic detach routine for the TCP/IP network stack

DESCRIPTION This library provides an interface between the Berkeley protocol stack and the MUX interface. The ***ipAttach*( )** routine binds the IP protocol to a specific device. It is called automatically during network initialization if **INCLUDE_END** is defined. The ***ipDetach*( )** routine removes an existing binding.

INCLUDE FILES **end.h muxLib.h etherMultiLib.h sys/ioctl.h etherLib.h**

kernelLib

NAME **kernelLib** – VxWorks kernel library

ROUTINES ***kernelInit*( )** – initialize the kernel
***kernelVersion*( )** – return the kernel revision string
***kernelTimeSlice*( )** – enable round-robin selection

DESCRIPTION The VxWorks kernel provides tasking control services to an application. The libraries **kernelLib**, **taskLib**, **semLib**, **tickLib**, and **wdLib** comprise the kernel functionality. This library is the interface to the VxWorks kernel initialization, revision information, and scheduling control.

KERNEL INITIALIZATION

The kernel must be initialized before any other kernel operation is performed. Normally kernel initialization is taken care of by the system configuration code in ***usrInit*( )** in **usrConfig.c**.

Kernel initialization consists of the following:

(1) Defining the starting address and size of the system memory partition. The ***malloc*( )** routine uses this partition to satisfy memory allocation requests of other facilities in VxWorks.

(2) Allocating the specified memory size for an interrupt stack. Interrupt service routines will use this stack unless the underlying architecture does not support a separate interrupt stack, in which case the service routine will use the stack of the interrupted task.

(3) Specifying the interrupt lock-out level. VxWorks will not exceed the specified level during any operation. The lock-out level is normally defined to mask the highest priority possible. However, in situations where extremely low interrupt latency is required, the lock-out level may be set to ensure timely response to the interrupt in question. Interrupt service routines handling interrupts of priority greater than the interrupt lock-out level may not call any VxWorks routine.

Once the kernel initialization is complete, a root task is spawned with the specified entry point and stack size. The root entry point is normally ***usrRoot*( )** of the **usrConfig.c** module. The remaining VxWorks initialization takes place in ***usrRoot*( )**.

ROUND-ROBIN SCHEDULING

Round-robin scheduling allows the processor to be shared fairly by all tasks of the same priority. Without round-robin scheduling, when multiple tasks of equal priority must share the processor, a single non-blocking task can usurp the processor until preempted by a task of higher priority, thus never giving the other equal-priority tasks a chance to run.

Round-robin scheduling is disabled by default. It can be enabled or disabled with the routine ***kernelTimeSlice*( )**, which takes a parameter for the "time slice" (or interval) that each task will be allowed to run before relinquishing the processor to another equal-priority task. If the parameter is zero, round-robin scheduling is turned off. If round-robin scheduling is enabled and preemption is enabled for the executing task, the 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

ROUTINES

***ledOpen*( )** – create a new line-editor ID
***ledClose*( )** – discard the line-editor ID
***ledRead*( )** – read a line with line-editing
***ledControl*( )** – change the line-editor ID parameters

DESCRIPTION

This library provides a line-editing layer on top of a **tty** device. The shell uses this interface for its history-editing features.

The shell history mechanism is similar to the UNIX Korn shell history facility, with a built-in line-editor similar to UNIX **vi** that allows previously typed commands to be edited. The command ***h*( )** displays the 20 most recent commands typed into the shell; old commands fall off the top as new ones are entered.

To edit a command, type ESC to enter edit mode, and use the commands listed below. The ESC key switches the shell to edit mode. The RETURN key always gives the line to the shell from either editing or input mode.

The following list is a summary of the commands available in edit mode.

Movement and search commands:

*n***G**	– Go to command number *n*.
/*s*	– Search for string *s* backward in history.
?*s*	– Search for string *s* forward in history.
n	– Repeat last search.
N	– Repeat last search in opposite direction.
*n***k**	– Get *n*th previous shell command in history.
*n***-**	– Same as **k**.
*n***j**	– Get *n*th next shell command in history.
*n***+**	– Same as **j**.
*n***h**	– Move left *n* characters.
CTRL+H	– Same as **h**.
*n***l**	– (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

ln97xEnd

NAME

ln97xEnd – END style AMD Am79C97X PCnet-PCI Ethernet driver

ROUTINES

***ln97xEndLoad*( )** – initialize the driver and device
***ln97xInitParse*( )** – parse the initialization string

DESCRIPTION

This module implements the Advanced Micro Devices Am79C971 Am79C972 and Am79C973 PCnet-PCI Ethernet 32 bit network interface driver.

The PCnet-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 in the driver. The second part of the interface comprises of the I/O control registers and their programming. The third part of the interface comprises of the descriptors and the 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 target-specific values and the external support routines are described below.

This driver supports multiple 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.

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 which the chip DMA s to have to be swapped and written to, so that when the hardware swaps the accesses, the chip would see them correctly. The chip still has to be programmed to operated 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 input and output byte stream need not be swapped.

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

EXTERNAL INTERFACE

The only external interface is the ***ln97xEndLoad*( )** routine, which expects the *initString* parameter as input. This parameter passes in a colon-delimited string of the format:

unit:*devMemAddr*:*devIoAddr*:*pciMemBase*:<*vecNum*:*intLvl*:*memAdrs*:
memSize:*memWidth*:*csr3b*:*offset*:*flags*

The ***ln97xEndLoad*( )** function uses ***strtok*( )** to parse the string.

TARGET-SPECIFIC PARAMETERS

unit
: A convenient holdover from the former model. This parameter is used only in the string name for the driver.

devMemAddr
: This parameter in the memory base address of the device registers in the memory map of the CPU. 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 derived from the "base address." This parameter should be equal to NONE if memory map is not used.

devIoAddr
: This parameter in the IO base address of the device registers in the IO map of some CPUs. It indicates to the driver where to find the RDP register. If both *devIoAddr* and *devMemAddr* are given then the device chooses *devMemAddr* which is a memory mapped register base address. This parameter should be equal to NONE if IO map is not used.

pciMemBase
: This parameter is the base address of the CPU memory as seen from the PCI bus. This parameter is zero for most intel architectures.

vecNum
: This parameter is the vector associated with the device interrupt. 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.

intLvl

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 ***sysLan97xIntEnable*( )** is called to perform any board-specific operations required to allow the servicing of a LANCE interrupt. For a description of ***sysLan97xIntEnable*( )**, see "External Support Requirements" below.

memAdrs

This parameter gives the driver the memory address to carve out its buffers and data structures. If this parameter is specified to be NONE then the driver allocates cache coherent memory for buffers and descriptors from the system pool. 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.

memSize

This parameter can be used to explicitly limit the amount of shared memory (bytes) this driver will use. 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.

memWidth

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.

csr3b

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 "UINT32 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. Normally for devices on the PCI bus this should always be little endian. This value is zero normally

offset

This parameter specifies the offset from which the packet has to be loaded from the begining of the device buffer. Normally this parameter is zero except for architectures which access long words only on aligned addresses. For these architectures the value of this offset should be 2.

flags
This is parameter is used for future use, currently its value should be zero.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires several external support functions, defined as macros:

```
SYS_INT_CONNECT(pDrvCtrl, routine, arg)
SYS_INT_DISCONNECT (pDrvCtrl, routine, arg)
SYS_INT_ENABLE(pDrvCtrl)
SYS_INT_DISABLE(pDrvCtrl)
SYS_OUT_BYTE(pDrvCtrl, reg, data)
SYS_IN_BYTE(pDrvCtrl, reg, data)
SYS_OUT_WORD(pDrvCtrl, reg, data)
SYS_IN_WORD(pDrvCtrl, reg, data)
SYS_OUT_LONG(pDrvCtrl, reg, data)
SYS_IN_LONG(pDrvCtrl, reg, data)
SYS_ENET_ADDR_GET(pDrvCtrl, pAddress)
sysLan97xIntEnable(pDrvCtrl->intLevel)
sysLan97xIntDisable(pDrvCtrl->intLevel)
sysLan97xEnetAddrGet(pDrvCtrl, enetAdrs)
```

There are default values in the source code for these macros. They presume memory mapped accesses to the device registers and the normal ***intConnect*( )**, and ***intEnable*( )** BSP functions. The first argument to each is the device controller structure. Thus, each has access back to all the device-specific information. Having the pointer in the macro facilitates the addition of new features to this driver.

The macros **SYS_INT_CONNECT**, **SYS_INT_DISCONNECT**, **SYS_INT_ENABLE**, and **SYS_INT_DISABLE** allow the driver to be customized for BSPs that use special versions of these routines.

The macro **SYS_INT_CONNECT** is used to connect the interrupt handler to the appropriate vector. By default it is the routine ***intConnect*( )**.

The macro **SYS_INT_DISCONNECT** is used to disconnect the interrupt handler prior to unloading the module. By default this is a dummy routine that returns OK.

The macro **SYS_INT_ENABLE** is used to enable the interrupt level for the end device. It is called once during initialization. It calls an external board level routine ***sysLan97xIntEnable*( )**.

The macro **SYS_INT_DISABLE** is used to disable the interrupt level for the end device. It is called during stop. It calls an external board level routine ***sysLan97xIntDisable*( )**.

The macro **SYS_ENET_ADDR_GET** is used get the ethernet hardware of the chip. This macro calls an external board level routine namely ***sysLan97xEnetAddrGet*( )** to get the ethernet address.

SYSTEM RESOURCE USAGE

When implemented, this driver requires the following system resources:

- one mutual exclusion semaphore
- one interrupt vector
- 13288 bytes in text for a I80486 target
- 64 bytes in the initialized data section (data)
- 0 bytes in the uninitialized data section (BSS)

The driver allocates clusters of size 1520 bytes for receive frames and and transmit frames.

INCLUDES **end.h endLib.h etherMultiLib.h ln97xEnd.h**

SEE ALSO **muxLib**, **endLib**, **netBufLib***Writing and Enhanced Network Driver Advanced Micro Devices PCnet-PCI Ethernet Controller for PCI.*

ln7990End

NAME **ln7990End** – END style AMD 7990 LANCE Ethernet network interface driver

ROUTINES

***ln7990EndLoad*( )** – initialize the driver and device
***ln7990InitParse*( )** – parse the initialization string
***ln7990InitMem*( )** – initialize memory for Lance chip

DESCRIPTION

This module implements the Advanced Micro Devices Am7990 LANCE Ethernet network interface driver. The driver can be configured to support big-endian or little-endian architectures, and it contains error recovery code to handle known device errata related to DMA activity.

This driver is designed to be moderately generic. Thus, it operates unmodified across the range of architectures and targets supported by VxWorks. To achieve this, the driver load routine requires an input string consisting of several target-specific values. The driver also requires some external support routines. These target-specific values and the external support routines are described below. If any of the assumptions stated below are not true for your particular hardware, this driver might not function correctly with that hardware.

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

EXTERNAL INTERFACE

The only external interface is the ***ln7990EndLoad*( )** routine, which expects the *initString* parameter as input. This parameter passes in a colon-delimited string of the format:

unit:*CSR_reg_addr*:*RAP_reg_addr*:*int_vector*:*int_level*:*shmem_addr*:*shmem_size*:*shmem_width*:*offset*:*csr3B*

The ***ln7990EndLoad( )*** function uses ***strtok( )*** to parse the string.

TARGET-SPECIFIC PARAMETERS

unit
A convenient holdover from the former model. This parameter is used only in the string name for the driver.

CSR_register_addr
Tells the driver where to find the CSR register.

RAP_register_addr
Tells the driver where to find the RAP register.

int_vector
Configures the LANCE device to generate hardware interrupts for various events within the device. Thus, it contains an interrupt handler routine. The driver calls ***sysIntConnect( )*** to connect its interrupt handler to the interrupt vector generated as a result of the LANCE interrupt.

int_level
This parameter is passed to an external support routine, ***sysLanIntEnable( )***, which is described below in "External Support Requirements." This routine is called during as part of driver's initialization. It handles any board-specific operations required to allow the servicing of a LANCE interrupt on targets that use additional interrupt controller devices to help organize and service the various interrupt sources. This parameter makes it possible for this driver to avoid all board-specific knowledge of such devices.

shmem_addr
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.

shmem_size
Use this parameter to explicitly limit the amount of shared memory (bytes) that this driver uses. Use "NONE" to indicate that there is no specific size limitation. This parameter is used only if a specific memory region is provided to the driver.

shmem_width
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 such targets, performing an access of an invalid width causes a bus error. Use this parameter to specify the

number of bytes on which data must be aligned if it is to be used by the driver during access to the shared memory. Use "NONE" to indicate that there are no restrictions. The support for this mechanism is not robust. Thus, its current implementation might not work on all targets requiring these restrictions.

offset
Specifies the memory alignment offset.

csr3B
Specifies the value that is placed into LANCE control register #3. This value determines the bus mode of the device and thus allows the support of big-endian and little-endian architectures. The default value supports Motorola-type buses. Normally this value is 0x4. For SPARC CPUs, it is normally set to 0x7 to add the ACON and BCON control bits. For more information on this register and the bus mode of the LANCE controller, see *Advanced Micro Devices Local Area Network Controller Am7990 (LANCE).*

EXTERNAL SUPPORT REQUIREMENTS

This driver requires several external support functions, defined as macros:

```
SYS_INT_CONNECT(pDrvCtrl, routine, arg)
SYS_INT_DISCONNECT (pDrvCtrl, routine, arg)
SYS_INT_ENABLE(pDrvCtrl)
SYS_OUT_SHORT(pDrvCtrl, reg, data)
SYS_IN_SHORT(pDrvCtrl, reg, pData)
```

There are default values in the source code for these macros. They presume memory-mapped accesses to the device registers and the normal ***intConnect*( )**, and ***intEnable*( )** BSP functions. The first argument to each is the device controller structure. Thus, each has access back to all the device-specific information. Having the pointer in the macro facilitates the addition of new features to this driver.

SYSTEM RESOURCE USAGE

When implemented, this driver requires the following system resources:

– one interrupt vector
– 68 bytes in the initialized data section (data)
– 0 bytes of bss

The above data and BSS requirements are for the MC68020 architecture and can 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 using the ***ln7990EndLoad*( )** 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 data that is written by the driver. That is because members within the shared structures are asynchronously modified by both the driver and the device, and these members might share the same cache line.

SEE ALSO **muxLib**, *Advanced Micro Devices Local Area Network Controller Am7990 (LANCE)*

---

loadLib

NAME **loadLib** – object module loader

ROUTINES ***loadModule*( )** – load an object module into memory
***loadModuleAt*( )** – load an object module into memory

DESCRIPTION This library provides a generic object module loading facility. Any supported format files may be loaded into memory, relocated properly, their external references resolved, and their external definitions added to the system symbol table for use by other modules and from the shell. Modules may be loaded from any I/O stream which allows repositioning of the pointer. This includes **netDrv**, nfs, or local file devices. It does not include sockets.

EXAMPLE

```
fdX = open ("/devX/objFile", O_RDONLY);
loadModule (fdX, LOAD_ALL_SYMBOLS);
close (fdX);
```

This code fragment would load the object file "objFile" located on device "/devX/" into memory which would be allocated from the system memory pool. All external and static definitions from the file would be added to the system symbol table.

This could also have been accomplished from the shell, by typing:

```
-> ld (1) </devX/objFile
```

INCLUDE FILE **loadLib.h**

SEE ALSO **usrLib**, **symLib**, **memLib**, *VxWorks Programmer's Guide: Basic OS*

loginLib

NAME

loginLib – user login/password subroutine library

ROUTINES

***loginInit*( )** – initialize the login table
***loginUserAdd*( )** – add a user to the login table
***loginUserDelete*( )** – delete a user entry from the login table
***loginUserVerify*( )** – verify a user name and password in the login table
***loginUserShow*( )** – display the user login table
***loginPrompt*( )** – display a login prompt and validate a user entry
***loginStringSet*( )** – change the login string
***loginEncryptInstall*( )** – install an encryption routine
***loginDefaultEncrypt*( )** – default password encryption routine

DESCRIPTION

This library provides a login/password facility for network access to the VxWorks shell. When installed, it requires a user name and password match to gain access to the VxWorks shell from rlogin or telnet. Therefore VxWorks can be used in secure environments where access must be restricted.

Routines are provided to prompt for the user name and password, and verify the response by looking up the name/password pair in a login user table. This table contains a list of user names and encrypted passwords that will be allowed to log in to the VxWorks shell remotely. Routines are provided to add, delete, and access the login user table. The list of user names can be displayed with ***loginUserShow*( )**.

INSTALLATION

The login security feature is initialized by the root task, ***usrRoot*( )**, in **usrConfig.c**, if the configuration macro **INCLUDE_SECURITY** is defined. Defining this macro also adds a single default user to the login table. The default user and password are defined as **LOGIN_USER_NAME** and **LOGIN_PASSWORD**. These can be set to any desired name and password. More users can be added by making additional calls to ***loginUserAdd*( )**. If **INCLUDE_SECURITY** is not defined, access to VxWorks will not be restricted and secure.

The name/password pairs are added to the table by calling ***loginUserAdd*( )**, which takes the name and an encrypted password as arguments. The VxWorks host tool vxencrypt is used to generate the encrypted form of a password. For example, to add a user name of "fred" and password of "flintstone", first run vxencrypt on the host to find the encryption of "flintstone" as follows:

```
% vxencrypt
please enter password: flintstone
encrypted password is ScebRezb9c
```

Then invoke the routine ***loginUserAdd*( )** in VxWorks:

```
loginUserAdd ("fred", "ScebRezb9c");
```

This can be done from the shell, a start-up script, or application code.

LOGGING IN

When the login security facility is installed, every attempt to rlogin or telnet to the VxWorks shell will first prompt for a user name and password.

```
% rlogin target
VxWorks login: fred
Password: flintstone
->
```

The delay in prompting between unsuccessful logins is increased linearly with the number of attempts, in order to slow down password-guessing programs.

ENCRYPTION ALGORITHM

This library provides a simple default encryption routine, ***loginDefaultEncrypt*( )**. This algorithm requires that passwords be at least 8 characters and no more than 40 characters.

The routine ***loginEncryptInstall*( )** allows a user-specified encryption function to be used instead of the default.

INCLUDE FILES

loginLib.h

SEE ALSO

shellLib, **vxencrypt**, *VxWorks Programmer's Guide: Shell*

---

logLib

NAME

logLib – message logging library

ROUTINES

***logInit*( )** – initialize message logging library
***logMsg*( )** – log a formatted error message
***logFdSet*( )** – set the primary logging file descriptor
***logFdAdd*( )** – add a logging file descriptor
***logFdDelete*( )** – delete a logging file descriptor
***logTask*( )** – message-logging support task

DESCRIPTION

This library handles message logging. It is usually used to display error messages on the system console, but such messages can also be sent to a disk file or printer.

The routines ***logMsg*( )** and ***logTask*( )** are the basic components of the logging system. The ***logMsg*( )** routine has the same calling sequence as ***printf*( )**, but instead of formatting and outputting the message directly, it sends the format string and arguments to a message queue. The task ***logTask*( )** waits for messages on this message queue. It formats each message according to the format string and arguments in the message, prepends the ID of the sender, and writes it on one or more file descriptors that have been specified as logging output streams (by ***logInit*( )** or subsequently set by ***logFdSet*( )** or ***logFdAdd*( )**).

USE IN INTERRUPT SERVICE ROUTINES

Because ***logMsg*( )** does not directly cause output to I/O devices, but instead simply writes to a message queue, it can be called from an 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, the routine ***logInit*( )** must be called before calling any other routine in this module. This is done by the root task, ***usrRoot*( )**, in **usrConfig.c**.

INCLUDE FILES

logLib.h

SEE ALSO

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

lptDrv

NAME **lptDrv** – parallel chip device driver for the IBM-PC LPT

ROUTINES ***lptDrv*( )** – initialize the LPT driver
***lptDevCreate*( )** – create a device for an LPT port
***lptShow*( )** – show LPT statistics

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

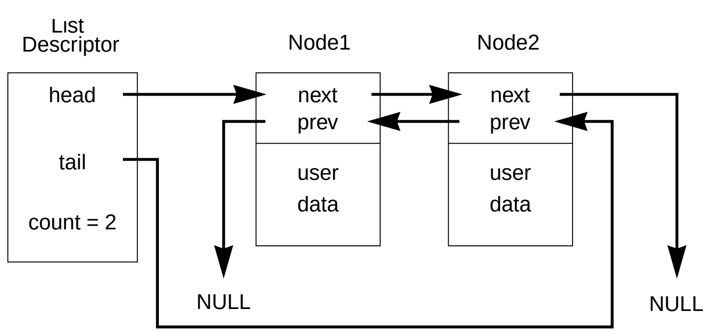
ROUTINES

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

DESCRIPTION

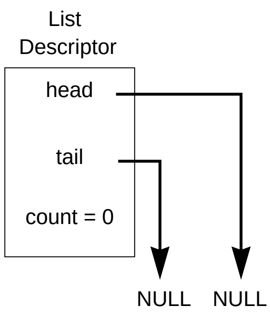
This subroutine library supports the creation and maintenance of a doubly linked list. The user supplies a list descriptor (type LIST) that will contain pointers to the first and last nodes in the list, and a count of the number of nodes in the list. The nodes in the list can be any user-defined structure,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

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



NON-EMPTY LIST

EMPTY LIST



INCLUDE FILES **lstLib.h**

m2IcmpLib

NAME **m2IcmpLib** – MIB-II ICMP-group API for SNMP Agents

ROUTINES
***m2IcmpInit*( )** – initialize MIB-II ICMP-group access
***m2IcmpGroupInfoGet*( )** – get the MIB-II ICMP-group global variables
***m2IcmpDelete*( )** – delete all resources used to access the ICMP group

DESCRIPTION This library provides MIB-II services for the ICMP group. It provides routines to initialize the group, and to access the group scalar variables. For a broader description of MIB-II services, see the manual entry for **m2Lib**.

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 **m2IcmpLib**, **m2Lib**, **m2IfLib**, **m2IpLib**, **m2TcpLib**, **m2SysLib**

m2IfLib

NAME

m2IfLib – MIB-II interface-group API for SNMP agents

ROUTINES

***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

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

ROUTINES

***m2IpInit*( )** – initialize MIB-II IP-group access
***m2IpGroupInfoGet*( )** – get the MIB-II IP-group scalar variables
***m2IpGroupInfoSet*( )** – set MIB-II IP-group variables to new values
***m2IpAddrTblEntryGet*( )** – get an IP MIB-II address entry
***m2IpAtransTblEntryGet*( )** – get a MIB-II ARP table entry
***m2IpAtransTblEntrySet*( )** – add, modify, or delete a MIB-II ARP entry
***m2IpRouteTblEntryGet*( )** – get a MIB-2 routing table entry
***m2IpRouteTblEntrySet*( )** – set a MIB-II routing table entry
***m2IpDelete*( )** – delete all resources used to access the IP group

DESCRIPTION

This library provides MIB-II services for the IP group. It provides routines to initialize the group, access the group scalar variables, read the table IP address, route and ARP table. The route and ARP table can also be modified. For a broader description of MIB-II services, see the manual entry for **m2Lib**.

USING THIS LIBRARY

To use this library, the MIB-II interface group must also be initialized; see the manual entry for **m2IfLib**. This library (**m2IpLib**) can be initialized and deleted by calling ***m2IpInit*( )** and ***m2IpDelete*( )** respectively, if only the IP group's services are needed. If full MIB-II support is used, this group and all other groups can be initialized and deleted by calling ***m2Init*( )** and ***m2Delete*( )**.

The following example demonstrates how to access and change IP scalar variables:

```
M2_IP   ipVars;
int     varToSet;
```

```
    if (m2IpGroupInfoGet (&ipVars) == OK)
        /* values in ipVars are valid */
    /* if IP is forwarding packets (MIB-II value is 1) turn it off */
    if (ipVars.ipForwarding == M2_ipForwarding_forwarding)
        {
        /* Not forwarding (MIB-II value is 2) */
        ipVars.ipForwarding = M2_ipForwarding_not_forwarding;
        varToSet |= M2_IPFORWARDING;
        }
    /* change the IP default time to live parameter */
    ipVars.ipDefaultTTL = 55;
    if (m2IpGroupInfoSet (varToSet, &ipVars) == OK)
        /* values in ipVars are valid */
```

The IP address table is a read-only table. Entries to this table can be retrieved as follows:

```
    M2_IPADDRTBL ipAddrEntry;
    /* Specify the index as zero to get the first entry in the table */
    ipAddrEntry.ipAdEntAddr = 0; /* Local IP address in host byte order */
    /* get the first entry in the table */
    if ((m2IpAddrTblEntryGet (M2_NEXT_VALUE, &ipAddrEntry) == OK)
        /* values in ipAddrEntry in the first entry are valid  */
    /* Process first entry in the table */
    /*
     * For the next call, increment the index returned in the previous call.
     * The increment is to the next possible lexicographic entry; for
     * example, if the returned index was 147.11.46.8 the index passed in the
     * next invocation should be 147.11.46.9.  If an entry in the table
     * matches the specified index, then that entry is returned.
     * Otherwise the closest entry following it, in lexicographic order,
     * is returned.
     */
    /* get the second entry in the table */
    if ((m2IpAddrTblEntryGet (M2_NEXT_VALUE, &ipAddrEntryEntry) == OK)
        /* values in ipAddrEntry in the second entry are valid  */
```

The IP Address Translation Table (ARP table) includes the functionality of the AT group plus additional functionality. The AT group is supported through this MIB-II table. Entries in this table can be added and deleted. An entry is deleted (with a set operation) by setting the **ipNetToMediaType** field to the MIB-II "invalid" value (2). The following example shows how to delete an entry:

```
M2_IPATRANSTBL        atEntry;
    /* Specify the index for the connection to be deleted in the table */
    atEntry.ipNetToMediaIfIndex     = 1       /* interface index */
    /* destination IP address in host byte order */
    atEntry.ipNetToMediaNetAddress  = 0x930b2e08;
                                            /* mark entry as invalid */
```

```
atEntry.ipNetToMediaType        = M2_ipNetToMediaType_invalid;
/* set the entry in the table */
if ((m2IpAtransTblEntrySet (&atEntry) == OK)
    /* Entry deleted successfully */
```

The IP route table allows for entries to be read, deleted, and modified. This example demonstrates how an existing route is deleted:

```
M2_IPROUTETBL        routeEntry;
/* Specify the index for the connection to be deleted in the table */
/* destination IP address in host byte order */
routeEntry.ipRouteDest       = 0x930b2e08;
                                            /* mark entry as invalid */
routeEntry.ipRouteType       = M2_ipRouteType_invalid;
/* set the entry in the table */
if ((m2IpRouteTblEntrySet (M2_IP_ROUTE_TYPE, &routeEntry) == OK)
    /* Entry deleted successfully */
```

INCLUDE FILES **m2Lib.h**

SEE ALSO **m2Lib**, **m2SysLib**, **m2IfLib**, **m2IcmpLib**, **m2UdpLib**, **m2TcpLib**

---

m2Lib

NAME **m2Lib** – MIB-II API library for SNMP agents

ROUTINES *m2Init*( ) – initialize the SNMP MIB-2 library
m2Delete( ) – delete all the MIB-II library groups

DESCRIPTION This library provides Management Information Base (MIB-II, defined in RFC 1213) services for applications wishing to have access to MIB parameters.

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

ROUTINES
*m2SysInit***( )** – initialize MIB-II system-group routines
*m2SysGroupInfoGet***( )** – get system-group MIB-II variables
*m2SysGroupInfoSet***( )** – set system-group MIB-II variables to new values
*m2SysDelete***( )** – delete resources used to access the MIB-II system group

DESCRIPTION This library provides MIB-II services for the system group. It provides routines to initialize the group and to access the group scalar variables. For a broader description of MIB-II services, see the manual entry for **m2Lib**.

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

ROUTINES

***m2TcpInit*( )** – initialize MIB-II TCP-group access
***m2TcpGroupInfoGet*( )** – get MIB-II TCP-group scalar variables
***m2TcpConnEntryGet*( )** – get a MIB-II TCP connection table entry
***m2TcpConnEntrySet*( )** – set a TCP connection to the closed state
***m2TcpDelete*( )** – delete all resources used to access the TCP group

DESCRIPTION

This library provides MIB-II services for the TCP group. It provides routines to initialize the group, access the group global variables, read the table of TCP connections, and change the state of a TCP connection. For a broader description of MIB-II services, see the manual entry for **m2Lib**.

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 addr 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

ROUTINES
***m2UdpInit*( )** – initialize MIB-II UDP-group access
***m2UdpGroupInfoGet*( )** – get MIB-II UDP-group scalar variables
***m2UdpTblEntryGet*( )** – get a UDP MIB-II entry from the UDP list of listeners
***m2UdpDelete*( )** – delete all resources used to access the UDP group

DESCRIPTION This library provides MIB-II services for the UDP group. It provides routines to initialize the group, access the group scalar variables, and read the table of UDP listeners. For a broader description of MIB-II services, see the manual entry for **m2Lib**.

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 addr 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 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

ROUTINES
***m68302SioInit*( )** – initialize a **M68302_CP**
***m68302SioInit2*( )** – initialize a **M68302_CP** (part 2)

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.

INCLUDE FILES **drv/sio/m68302Sio.h sioLib.h**

SEE ALSO **ttyDrv**, **tyLib**

m68332Sio

NAME **m68332Sio** – Motorola MC68332 tty driver

ROUTINES
***m68332DevInit*( )** – initialize the SCC
***m68332Int*( )** – handle an SCC interrupt

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

ROUTINES
***m68360DevInit*( )** – initialize the SCC
***m68360Int*( )** – handle an SCC interrupt

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

ROUTINES

***m68562HrdInit*( )** – initialize the DUSCC
***m68562RxTxErrInt*( )** – handle a receiver/transmitter error interrupt
***m68562RxInt*( )** – handle a receiver interrupt
***m68562TxInt*( )** – handle a transmitter interrupt

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

ROUTINES

***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

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

ROUTINES ***m68901DevInit*( )** – initialize a **M68901_CHAN** structure

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**

---

masterIoLib

NAME **masterIoLib** – default IO routines for the SNMP master agent

ROUTINES

***masterIoInit*( )** – create the IPC mechanism at the SNMP master agent
***snmpMonitorSpawn*( )** – spawn **tMonQue** to run ***snmpQueMonitor*( )**
***masterIpcComp*( )** – transmit a completion of transmission message
***masterIoWrite*( )** – send the encoded buffer to the subagent
***masterIpcSend*( )** – send a message to a subagent
***masterIpcRcv*( )** – wait for a reply from the subagent

***masterIpcAyt*( )** – check the status of the IPC link
***masterIpcFree*( )** – free the IPC resources allocated by the SNMP master agent
***masterQueCleanup*( )** – free resources allocated for SNMP master agent

DESCRIPTION

This module implements the I/O routines used by the SNMP master agent. As shipped, the WindNet SNMP code uses message queues to communicate between the master agent and its subagents. The SNMP master agent also uses a message queue to handle communication between its two component tasks, **tSnmpd**and **tMonQue**. The **tSnmpd** task handles communication with the SNMP manager. The **tMonQue** task is a secondary task spawned from **tSnmpd** to receive messages from subagents.

When **tSnmpd** spawns **tMonQue**, it assigns ***snmpQueMonitor*( )** to manage the process. This function waits on the message queue that subagents use to send messages to the master agent. The ***snmpQueMonitor*( )** function interprets messages on its queue using an **SA_MESSAGE_T** structure, which is defined in **ipcLib.h** as:

```
typedef struct SA_MESSAGE_S
    {
    int             msgType;
    MSG_Q_ID        saId;
    EBUFFER_T       mesg;
    } SA_MESSAGE_T;
```

A switch internal to ***snmpQueMonitor*( )** handles the message according to the value of the **msgType** member.

If the message type is **CALL_QUERY_HANDLER**, the message is a response to a query from the master agent. The buffer referenced in the **mesg** is then transferred to the local message queue monitored by **tSnmpd**, which is waiting for a query response from a subagent.

If the message type is **CALL_REG_HANDLER**, the message is either a registration request, a deregistration request, or some other control message (such as a trap). To respond to such requests, ***snmpQueMonitor*( )** passes the buffer in **mesg** to ***snmpMasterHandlerWR*( )**.

If the message submitted to ***snmpMasterHandlerWR*( )** is a registration request, it includes information on a set of leaves representing the objects that the subagent wants to add to the master agent's MIB tree. If the message passes all checks, the objects are added to the master agent's MIB tree and ***snmpMasterHandlerWR*( )** returns success. All objects registered in one message become part of a group. They share the same IPC information, and, if the IPC link to their subagent is broken, they are deactivated as a group.

If ***snmpMasterHandlerWR*( )** returns a function value indicating success, it also returns a message for the subagent containing the group ID for the variables just added. The ***snmpQueMonitor*( )** takes responsibility for forwarding this message to the subagent. The subagent uses the group ID contained in this message when it comes time to deregister, as well as when it must register instances of an already registered object.

The returned function value of ***snmpMasterHandlerWR*( )** could indicate failure or an opcode. You might want to rewrite this code to do something different. For example, if the subagent had sent a trap up to the master agent, the returned value of ***snmpMasterHandlerWR*( )** would be **SA_TRAP_REQUEST**, and the *vblist* parameter would contain a varbind list from the subagent. In this case, you would want to modify ***snmpQueMonitor*( )** to pass the trap on to the SNMP manager.

MIB VARIABLES ADDED BY A SUBAGENT

These MIB variables that the subagent adds to the master agent's MIB tree look transparent to the SNMP manager that is in communication with the master agent. However, the method routines associated with these MIB variables in the master agent are not standard MIB routines. Instead, they are special routines that know how to queue test, get, and set work on the subagent that registered the MIB variables with the master agent. From the point of view of the PDU processing code, these special method routines look like any other method routines. However, when **tSnmpd** executes one of these routines, the special method routine actually passes the work on to a subagent while **tSnmpd** waits on a local message queue.

Because the subagent does not know about this local message queue, its response to the master agent is somewhat indirect. The only master agent message queue known to the subagent is the message queue managed by **tMonQue**, so the subagent puts its response on that queue. When the ***snmpQueMonitor*( )** function that **tMonQue** runs to monitor the message queue sees that the message is a query response, it then transfers the message to the local queue upon which **tSnmpd** is awaiting a response. When **tSnmpd** sees the response, it parses it and merges the message into the PDU processing system.

SERIAL VERSUS ASYNCHRONOUS SUBAGENT PROCESSING

In the shipped implementation, communication between the master agent and its subagents is handled serially. For example, if the SNMP manager made a request concerning three MIB variables managed by three different subagents, the master agent would query each subagent in turn. After gathering all three responses, the master agent would then pack them up and ship the information back to the SNMP manager.

With some modifications to the code, you could rewrite the SNMP master agent to query all three subagents simultaneously (see the description of the ***snmpMasterHandlerAsync*( )** function defined in **subagentLib.c**). That is, the master agent would query all three subagents one after the other without waiting for a response after making each request. If the subagents reside on different targets (each with its own processor), this asynchronous query method of multiple subagents lets you take advantage of the capacity for parallel processing.

However, if the subagents reside on different targets, you will also need to replace the code that implements the IPC mechanism used between the master agent and its subagents. In the shipped code, message queues serve as the IPC mechanism. To support agents that reside on different machines, you must replace this IPC mechanism with something such as sockets. To make this possible, the functions that implement the IPC mechanism are isolated to **masterIoLib.c** and **saIoLib.c**. These files ship as source code that you should feel free to edit as needed.

mathALib

NAME

mathALib – C interface library to high-level math functions

ROUTINES

***acos*( )** – compute an arc cosine (ANSI)
***asin*( )** – compute an arc sine (ANSI)
***atan*( )** – compute an arc tangent (ANSI)
***atan2*( )** – compute the arc tangent of y/x (ANSI)
***cbrt*( )** – compute a cube root
***ceil*( )** – compute the smallest integer greater than or equal to a specified value (ANSI)
***cos*( )** – compute a cosine (ANSI)
***cosh*( )** – compute a hyperbolic cosine (ANSI)
***exp*( )** – compute an exponential value (ANSI)
***fabs*( )** – compute an absolute value (ANSI)
***floor*( )** – compute the largest integer less than or equal to a specified value (ANSI)
***fmod*( )** – compute the remainder of x/y (ANSI)
***infinity*( )** – return a very large double
***irint*( )** – convert a double-precision value to an integer
***iround*( )** – round a number to the nearest integer
***log*( )** – compute a natural logarithm (ANSI)
***log10*( )** – compute a base-10 logarithm (ANSI)
***log2*( )** – compute a base-2 logarithm
***pow*( )** – compute the value of a number raised to a specified power (ANSI)
***round*( )** – round a number to the nearest integer
***sin*( )** – compute a sine (ANSI)
***sincos*( )** – compute both a sine and cosine
***sinh*( )** – compute a hyperbolic sine (ANSI)
***sqrt*( )** – compute a non-negative square root (ANSI)
***tan*( )** – compute a tangent (ANSI)
***tanh*( )** – compute a hyperbolic tangent (ANSI)
***trunc*( )** – truncate to integer
***acosf*( )** – compute an arc cosine (ANSI)
***asinf*( )** – compute an arc sine (ANSI)
***atanf*( )** – compute an arc tangent (ANSI)
***atan2f*( )** – compute the arc tangent of y/x (ANSI)
***cbrtf*( )** – compute a cube root
***ceilf*( )** – compute the smallest integer greater than or equal to a specified value (ANSI)
***cosf*( )** – compute a cosine (ANSI)
***coshf*( )** – compute a hyperbolic cosine (ANSI)
***expf*( )** – compute an exponential value (ANSI)
***fabsf*( )** – compute an absolute value (ANSI)
***floorf*( )** – compute the largest integer less than or equal to a specified value (ANSI)
***fmodf*( )** – compute the remainder of x/y (ANSI)
***infinityf*( )** – return a very large float

***irintf*( )** – convert a single-precision value to an integer
***iroundf*( )** – round a number to the nearest integer
***logf*( )** – compute a natural logarithm (ANSI)
***log10f*( )** – compute a base-10 logarithm (ANSI)
***log2f*( )** – compute a base-2 logarithm
***powf*( )** – compute the value of a number raised to a specified power (ANSI)
***roundf*( )** – round a number to the nearest integer
***sinf*( )** – compute a sine (ANSI)
***sincosf*( )** – compute both a sine and cosine
***sinhf*( )** – compute a hyperbolic sine (ANSI)
***sqrtf*( )** – compute a non-negative square root (ANSI)
***tanf*( )** – compute a tangent (ANSI)
***tanhf*( )** – compute a hyperbolic tangent (ANSI)
***truncf*( )** – truncate to integer

DESCRIPTION

This library provides a C interface to high-level floating-point math functions, which can use either a hardware floating-point unit or a software floating-point emulation library. The appropriate routine is called based on whether ***mathHardInit*( )** or ***mathSoftInit*( )** or both have been called to initialize the interface.

All angle-related parameters are expressed in radians. All functions in this library with names corresponding to ANSI C specifications are ANSI compatible.

WARNING

Not all functions in this library are available on all architectures. 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

ROUTINES

***mathHardInit*( )** – initialize hardware floating-point math support

DESCRIPTION

This library provides support routines for using hardware floating-point units with high-level math functions. The high-level functions include 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

ROUTINES

***mathSoftInit*( )** – initialize software floating-point math support

DESCRIPTION

This library provides software emulation of various high-level floating-point operations. This emulation is generally for use in systems that lack a floating-point coprocessor.

WARNING

Software floating point is not supported for all architectures. See the architecture-specific appendices of the *VxWorks Programmer's Guide.*

INCLUDE FILES

math.h

SEE ALSO

mathHardLib, **mathALib**, *VxWorks Programmer's Guide* architecture-specific appendices

mb86940Sio

NAME **mb86940Sio** – MB 86940 UART tty driver

ROUTINES ***mb86940DevInit*( )** – install the driver function table

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**

mb86960End

NAME **mb86960End** – END-style Fujitsu MB86960 Ethernet network interface driver

ROUTINES ***mb86960EndLoad*( )** – initialize the driver and device
***mb86960InitParse*( )** – parse the initialization string
***mb86960MemInit*( )** – initialize memory for the chip

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.

The MB86960 Network Interface Controller with Encoder/Decoder (NICE) chip is a highly integrated monolithic device which incorporates both network controller, complete with buffer management and Manchester encoder/decoder.

TARGET-SPECIFIC PARAMETERS

The format of the parameter string is *unit*:*devBaseAddr*:*ivec*, where:

unit
: A convenient holdover from the former model. It is only used in the string name for the driver.

devBaseAddr
: The base Address of the chip registers.

ivec
: This is the interrupt vector number of the hardware interrupt generated by this ethernet device. The driver uses ***intConnect*( )** to attach an interrupt handler to this interrupt.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires seven external support functions:

***sys86960IntEnable*( )**

```
void sysEnetIntEnable (int unit)
```

This routine provides a target-specific interface to enable Ethernet device interrupts for a given device unit. For this driver, value of unit must be 0.

***sys86960IntDisable*( )**

```
void sysEnetIntDisable (int unit)
```

This routine provides a target-specific interface to disable Ethernet device interrupts for a given device unit. For this driver, value of unit must be 0.

***sysEnetAddrGet*( )**

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

This routine provides a target-specific interface to access a device Ethernet address. This routine should provide a six-byte Ethernet address in the *enetAdrs* parameter and return OK or ERROR.

In this driver the macros **SYS_OUT_SHORT** and **SYS_IN_SHORT** which call BSP-specific functions to access the chip register.

INCLUDES **end.h endLib.h etherMultiLib.h**

SEE ALSO **muxLib**, **endLib**, *Writing and Enhanced Network Driver*

mb87030Lib

NAME
mb87030Lib – Fujitsu MB87030 SCSI Protocol Controller (SPC) library

ROUTINES
***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

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*

mbcEnd

NAME
mbcEnd – Motorola 68302fads END network interface driver

ROUTINES
***mbcEndLoad*( )** – initialize the driver and device
***mbcParse*( )** – parse the init string
***mbcMemInit*( )** – initialize memory for the chip
***mbcAddrFilterSet*( )** – set the address filter for multicast addresses

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 requires that the memory used for transmit and receive buffers be allocated in cache-safe RAM area.

Up to 61 multicast addresses are supported. Multicast addresses are supported by adding the multicast ethernet addresses to the address table in the ethernet part. If more than 61 multicast addresses are desired, address hashing must be used (the address table holds 62 entries at most). However, address hashing does not appear to work in this ethernet part.

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

The only external interface is the ***mbcEndLoad( )*** routine, which expects the *initString* parameter as input. This parameter passes in a colon-delimited string of the format:

unit:memAddr:ivec:txBdNum:rxBdNum:dmaParms:bufBase:offset

TARGET-SPECIFIC PARAMETERS

unit
: A convenient holdover from the former model. This parameter is used only in the string name for the driver.

memAddr
: This parameter is the base address of the Ethernet module. The driver addresses all other Ethernet device registers as offsets from this address.

ivec
: The interrupt vector to be used in connecting the interrupt handler.

txBdNum
: The number of transmit buffer descriptors to use.

rxBdNum
: The number of receive buffer descriptors to use.

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 0, 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.

dmaParms
: Ethernet DMA parameters.

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.

bufBase
Base address of the 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 will be performed. Each buffer is 1520 bytes. If this parameter is "NULL", space for buffers will be obtained by calling ***cacheDmaMalloc*( )** in ***mbcMemInit*( )**.

offset
Specifies the memory alignment offset.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires several external support functions, defined as macros:

```
SYS_INT_CONNECT(pDrvCtrl, routine, arg)
SYS_INT_DISCONNECT (pDrvCtrl, routine, arg)
SYS_INT_ENABLE(pDrvCtrl)
SYS_OUT_SHORT(pDrvCtrl, reg, data)
SYS_IN_SHORT(pDrvCtrl, reg, pData)
```

There are default values in the source code for these macros. They presume memory-mapped accesses to the device registers and the normal ***intConnect*( )**, and ***intEnable*( )** BSP functions. The first argument to each is the device controller structure. Thus, each has access back to all the device-specific information. Having the pointer in the macro facilitates the addition of new features to this driver.

SYSTEM RESOURCE USAGE

The driver requires the following system resources:

– one watchdog timer
– one interrupt vector
– 52 bytes in the initialized data section (data)
– 0 bytes in the uninitialized data section (bss)

The above data and bss requirements are for MC680x0 architectures and varies for other architectures. Code size (text) varies greatly between architectures and is 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, the total size requested is roughly 100,000 bytes. If a memory region is provided to the driver, the size of this region is adjustable to suit user needs.

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.

INCLUDES **end.h endLib.h etherMultiLib.h**

SEE ALSO **muxLib**, **endLib**, *Writing and Enhanced Network Driver*

memDrv

NAME **memDrv** – pseudo memory device driver

ROUTINES

***memDrv*( )** – install a memory driver
***memDevCreate*( )** – create a memory device
***memDevCreateDir*( )** – create a memory device for multiple files
***memDevDelete*( )** – delete a memory device

DESCRIPTION

This driver allows the I/O system to access memory directly as a pseudo-I/O device. Memory location and size are specified when the device is created. This feature is useful when data must be preserved between boots of VxWorks or when sharing data between CPUs.

Additionally, it can be used to build some files into a VxWorks binary image (having first converted them to data arrays in C source files, using a utility such as memdrvbuild), and then mount them in the filesystem; this is a simple way of delivering some non-changing files with VxWorks. For example, a system with an integrated web server may use this technique to build some HTML and associated content files into VxWorks.

memDrv can be used to simply provide a high-level method of reading and writing bytes in absolute memory locations through I/O calls. It can also be used to implement a simple, essentially read-only filesystem (exsisting files can be rewritten within their existing sizes); directory searches and a limited set of IOCTL calls (including ***stat*( )**) are supported.

USER-CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. Four routines, however, can be called directly: ***memDrv*( )** to initialize the driver, ***memDevCreate*( )** and ***memDevCreateDir*( )** to create devices, and ***memDevDelete*( )** to delete devices.

Before using the driver, it must be initialized by calling ***memDrv*( )**. This routine should be called only once, before any reads, writes, or ***memDevCreate*( )** calls. It may be called from ***usrRoot*( )** in **usrConfig.c**or at some later point.

IOCTL FUNCTIONS

The dosFs file system supports the following ***ioctl*( )** functions. The functions listed are defined in the header **ioLib.h**. Unless stated otherwise, the file descriptor used for these functions may be any file descriptor which is opened to a file or directory on the volume or to the volume itself.

FIOGETFL
Copies to *flags* the open mode flags of the file (**O_RDONLY**, **O_WRONLY**, **O_RDWR**):

```
int flags;
status = ioctl (fd, FIOGETFL, &flags);
```

FIOSEEK
Sets the current byte offset in the file to the position specified by *newOffset*:

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

The **FIOSEEK** offset is always relative to the beginning of the file. The offset, if any, given at open time by using pseudo-file name is overridden.

FIOWHERE
Returns the current byte position in the file. This is the byte offset of the next byte to be read or written. It takes no additional argument:

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

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

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

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

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

FIOFSTATGET
Gets file status information (directory entry data). The argument *statStruct* is a pointer to a stat structure that is filled with data describing the specified file. File inode numbers, user and group IDs, and times are not supported (returned as 0).

Normally, the ***stat*( )** or ***fstat*( )** routine is used to obtain file information, rather than using the **FIOFSTATGET** function directly. See **dirLib**.

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

Any other ***ioctl*( )** function codes will return error status.

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

memLib

NAME

memLib – full-featured memory partition manager

ROUTINES

***memPartOptionsSet*( )** – set the debug options for a memory partition
***memalign*( )** – allocate aligned memory
***valloc*( )** – allocate memory on a page boundary
***memPartRealloc*( )** – reallocate a block of memory in a specified partition
***memPartFindMax*( )** – find the size of the largest available free block
***memOptionsSet*( )** – set the debug options for the system memory partition
***calloc*( )** – allocate space for an array (ANSI)
***realloc*( )** – reallocate a block of memory (ANSI)
***cfree*( )** – free a block of memory
***memFindMax*( )** – find the largest free block in the system memory partition

DESCRIPTION

This library provides full-featured facilities for managing the allocation of blocks of memory from ranges of memory called memory partitions. The library is an extension of **memPartLib** and provides enhanced memory management features, including error handling, aligned allocation, and ANSI allocation routines. For more information about the core memory partition management facility, see the manual entry for **memPartLib**.

The system memory partition is created when the kernel is initialized by ***kernelInit*( )**, which is called by the root task, ***usrRoot*( )**, in **usrConfig.c**. The ID of the system memory partition is stored in the global variable **memSysPartId**; its declaration is included in **memLib.h**.

The ***memalign*( )** routine is provided for allocating memory aligned to a specified boundary.

This library includes three ANSI-compatible routines: ***calloc*( )** allocates a block of memory for an array; ***realloc*( )** changes the size of a specified block of memory; and ***cfree*( )** returns to the free memory pool a block of memory that was previously allocated with ***calloc*( )**.

ERROR OPTIONS

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

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

MEM_ALLOC_ERROR_SUSPEND_FLAG
Suspend the task when there is an error in allocating memory (unless the task was spawned with the **VX_UNBREAKABLE** option, in which case it cannot be suspended).

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

MEM_BLOCK_ERROR_SUSPEND_FLAG
Suspend the task when there is an error in freeing memory (unless the task was spawned with the **VX_UNBREAKABLE** option, in which case it cannot be suspended).

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

MEM_BLOCK_CHECK
Check each block freed.

Setting either of the **MEM_BLOCK_ERROR** options automatically sets **MEM_BLOCK_CHECK**.

The default options when a partition is created are:

MEM_ALLOC_ERROR_LOG_FLAG
MEM_BLOCK_CHECK
MEM_BLOCK_ERROR_LOG_FLAG
MEM_BLOCK_ERROR_SUSPEND_FLAG

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

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

INCLUDE FILES **memLib.h**

SEE ALSO **memPartLib**, **smMemLib**

memPartLib

NAME

memPartLib – core memory partition manager

ROUTINES

***memPartCreate*( )** – create a memory partition
***memPartAddToPool*( )** – add memory to a memory partition
***memPartAlignedAlloc*( )** – allocate aligned memory from a partition
***memPartAlloc*( )** – allocate a block of memory from a partition
***memPartFree*( )** – free a block of memory in a partition
***memAddToPool*( )** – add memory to the system memory partition
***malloc*( )** – allocate a block of memory from the system memory partition (ANSI)
***free*( )** – free a block of memory (ANSI)

DESCRIPTION

This library provides core facilities for managing the allocation of blocks of memory from ranges of memory called memory partitions. The library was designed to provide a compact implementation; full-featured functionality is available with **memLib**, which provides enhanced memory management features built as an extension of **memPartLib**. (For more information about enhanced memory partition management options, see the manual entry for memLib.) This library consists of two sets of routines. The first set, ***memPart...*( )**, comprises a general facility for the creation and management of memory partitions, and for the allocation and deallocation of blocks from those partitions. The second set provides a traditional ANSI-compatible ***malloc*( )**/***free*( )** interface to the system memory partition.

The system memory partition is created when the kernel is initialized by ***kernelInit*( )**, which is called by the root task, ***usrRoot*( )**, in **usrConfig.c**. The ID of the system memory partition is stored in the global variable **memSysPartId**; its declaration is included in **memLib.h**.

The allocation of memory, using ***malloc*( )** in the typical case and ***memPartAlloc*( )** for a specific memory partition, is done with a first-fit algorithm. Adjacent blocks of memory are coalesced when they are freed with ***memPartFree*( )** and ***free*( )**. There is also a routine provided for allocating memory aligned to a specified boundary from a specific memory partition, ***memPartAlignedAlloc*( )**.

CAVEATS

Architectures have various alignment constraints. To provide optimal performance, ***malloc*( )** returns a pointer to a buffer having the appropriate alignment for the architecture in use. The portion of the allocated buffer reserved for system bookkeeping, known as the overhead, may vary depending on the architecture.

Architecture	Boundary	Overhead
68K	4	8
SPARC	8	12
MIPS	16	12
i960	16	16

INCLUDE FILES **memLib.h**, **stdlib.h**

SEE ALSO **memLib**, **smMemLib**

memShow

NAME **memShow** – memory show routines

ROUTINES
memShowInit( ) – initialize the memory partition show facility
memShow( ) – show system memory partition blocks and statistics
memPartShow( ) – show partition blocks and statistics
memPartInfoGet( ) – get partition information

DESCRIPTION This library contains memory partition information display routines. To use this facility, it must first be installed using ***memShowInit( )***, which is called automatically when the memory partition show facility is configured into VxWorks using either of the following methods:

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

– If you use the Tornado project facility, select **INCLUDE_MEM_SHOW**.

SEE ALSO **memLib**, **memPartLib**, *VxWorks Programmer's Guide: Target Shell*, windsh, *Tornado User's Guide: Shell*

mmanPxLib

NAME **mmanPxLib** – memory management library (POSIX)

ROUTINES
mlockall( ) – lock all pages used by a process into memory (POSIX)
munlockall( ) – unlock all pages used by a process (POSIX)
mlock( ) – lock specified pages into memory (POSIX)
munlock( ) – unlock specified pages (POSIX)

DESCRIPTION This library contains POSIX interfaces designed to lock and unlock memory pages, i.e., to control whether those pages may be swapped to secondary storage. Since VxWorks does not use swapping (all pages are always kept in memory), these routines have no real effect and simply return 0 (OK).

INCLUDE FILES **sys/mman.h**

SEE ALSO POSIX 1003.1b document

mmuL64862Lib

NAME **mmuL64862Lib** – LSI Logic L64862 MBus-to-SBus Interface: I/O DMA library (SPARC)

ROUTINES ***mmuL64862DmaInit*( )** – initialize the L64862 I/O MMU DMA data structures (SPARC)

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**

mmuPro32Lib

NAME **mmuPro32Lib** – mmu library for PentiumPro/II

ROUTINES ***mmuPro32LibInit*( )** – initialize module

DESCRIPTION **mmuPro32Lib.c** provides the architecture dependent routines that directly control the memory management unit. It provides 10 routines that are called by the higher level architecture independent routines in **vmLib.c**:

***mmuLibInit*( )** – initialize module
***mmuTransTblCreate*( )** – create a new translation table
***mmuTransTblDelete*( )** – delete a translation table.
***mmuEnable*( )** – turn MMU on or off
***mmuStateSet*( )** – set state of virtual memory page
***mmuStateGet*( )** – get state of virtual memory page
***mmuPageMap*( )** – map physical memory page to virtual memory page
***mmuGlobalPageMap*( )** – map physical memory page to global virtual memory page
***mmuTranslate*( )** – translate a virtual address to a physical address
***mmuCurrentSet*( )** – change active translation table

Applications using the MMU will never call these routines directly; the visable interface is supported in **vmLib.c**.

mmuLib supports the creation and maintenance of multiple translation tables, one of which is the active translation table when the mmu is enabled. Note that VxWorks does not include a translation table as part of the task context; individual tasks do not reside in private virtual memory. However, we include the facilities to create multiple translation tables so that the user may create "private" virtual memory contexts and switch them in an application specific manner. New translation tables are created with a call to ***mmuTransTblCreate*( )**, and installed as the active translation table with ***mmuCurrentSet*( )**. Translation tables are modified and potentially augmented with calls to ***mmuPageMap*( )** and ***mmuStateSet*( )**. The state of portions of the translation table can be read with calls to ***mmuStateGet*( )** and ***mmuTranslate*( )**.

The traditional VxWorks architecture and design philosophy requires that all objects and operating systems resources be visable and accessable to all agents (tasks, ISRs, watchdog timers, etc) in the system. This has traditionally been insured by the fact that all objects and data structures reside in physical memory; thus, a data structure created by one agent may be accessed by any other agent using the same pointer (object identifiers in VxWorks are often pointers to data structures.) This creates a potential problem if you have multiple virtual memory contexts. For example, if a semaphore is created in one virtual memory context, you must gurantee that that semaphore will be visible in all virtual memory contexts if the semaphore is to be accessed at interrupt level, when a virtual memory context other than the one in which it was created may be active. Another example is that code loaded using the incremental loader from the shell must be accessible in all virtual memory contexts, since code is shared by all agents in the system.

This problem is resolved by maintaining a global "transparent" mapping of virtual to physical memory for all the contiguous segments of physical memory (on board memory, I/O space, sections of VME space, etc.) that is shared by all translation tables; all available physical memory appears at the same address in virtual memory in all virtual memory contexts. This technique provides an environment that allows resources that rely on a globally accessible physical address to run without modification in a system with multiple virtual memory contexts.

An additional requirement is that modifications made to the state of global virtual memory in one translation table appear in all translation tables. For example, memory containing the text segment is made read only (to avoid accidental corruption) by setting the appropriate writeable bits in the translation table entries corresponding to the virtual memory containing the text segment. This state information must be shared by all virtual memory contexts, so that no matter what translation table is active, the text segment is protected from corruption. The mechanism that implements this feature is architecture dependent, but usually entails building a section of a translation table that corresponds to the global memory, that is shared by all other translation tables. Thus, when changes to the state of the global memory are made in one translation table, the changes are reflected in all other translation tables.

mmuLib provides a seperate call for constructing global virtual memory – ***mmuGlobalPageMap*( )** – which creates translation table entries that are shared by all translation tables. Initialization code in **usrConfig** makes calls to ***vmGlobalMap*( )** (which in turn calls ***mmuGlobalPageMap*( )**) to set up global transparent virtual memory for all available physical memory. All calls made to ***mmuGlobaPageMap*( )** must occur before any virtual memory contexts are created; changes made to global virtual memory after virtual memory contexts are created are not guaranteed to be reflected in all virtual memory contexts.

Most MMU architectures will dedicate some fixed amount of virtual memory to a minimal section of the translation table (a "segment", or "block"). This creates a problem in that the user may map a small section of virtual memory into the global translation tables, and then attempt to use the virtual memory after this section as private virtual memory. The problem is that the translation table entries for this virtual memory are contained in the global translation tables, and are thus shared by all translation tables. This condition is detected by ***vmMap*( )**, and an error is returned, thus, the lower level routines in **mmuPro32Lib.c** (***mmuPageMap*( )**, ***mmuGlobalPageMap*( )**) need not perform any error checking.

A global variable called **mmuPageBlockSize** should be defined which is equal to the minimum virtual segment size. **mmuLib** must provide a routine ***mmuGlobalInfoGet*( )**, which returns a pointer to the **globalPageBlock[ ]** array. This provides the user with enough information to be able to allocate virtual memory space that does not conflict with the global memory space.

This module supports the PentiumPro/II MMU:

```
                            PDBR
                             |
                             |
            -------------------------------------
 top level  |pde  |pde  |pde  |pde  |pde  |pde  | ...
            -------------------------------------
               |     |     |     |     |     |
               |     |     |     |     |     |
      ----------     |     v     v     v     v
      |         ------    NULL  NULL  NULL  NULL
      |         |
      v         v
     ----     ----
l   |pte |   |pte |
o    ----     ----
w   |pte |   |pte |
e    ----     ----
r   |pte |   |pte |
l    ----     ----
e   |pte |   |pte |
v    ----     ----
e     .         .
l     .         .
      .         .
```

where the top level consists of an array of pointers (Page Directory Entry) held within a single 4K page. These point to arrays of Page Table Entry arrays in the lower level. Each of these lower level arrays is also held within a single 4K page, and describes a virtual space of 4 MB (each Page Table Entry is 4 bytes, so we get 1000 of these in each array, and each Page Table Entry maps a 4KB page – thus 1000 * 4096 = 4MB).

To implement global virtual memory, a seperate translation table called **mmuGlobalTransTbl[ ]** is created when the module is initialized. Calls to ***mmuGlobalPageMap*( )** will augment and modify this translation table. When new translation tables are created, memory for the top level array of sftd's is allocated and initialized by duplicating the pointers in ***mmuGlobalTransTbl*( )**'s top-level **sftd** array. Thus, the new translation table will use the global translation table's state information for portions of virtual memory that are defined as global. Here's a picture to illustrate:

```
                 GLOBAL TRANS TBL                     NEW TRANS TBL
                       PDBR                                PDBR
                        |                                   |
                        |                                   |
            -------------------------           -------------------------
 top level  |pde  |pde  | NULL| NULL|           |pde  |pde  | NULL| NULL|
            -------------------------           -------------------------
               |     |     |     |                 |     |     |     |
               |     |     |     |                 |     |     |     |
      ----------     |     v     v        ----------     |     v     v
      |         ------    NULL  NULL      |              |    NULL  NULL
      |         |                         |              |
      o------------------------------------              |
      |         |                                        |
      |         o-----------------------------------------
      |         |
      v         v
     ----     ----
l   |pte |   |pte |
o    ----     ----
w   |pte |   |pte |
e    ----     ----
r   |pte |   |pte |
l    ----     ----
e   |pte |   |pte |
v    ----     ----
e     .         .
l     .         .
      .         .
```

Note that with this scheme, the global memory granularity is 4MB. Each time you map a section of global virtual memory, you dedicate at least 4MB of the virtual space to global virtual memory that will be shared by all virtual memory contexts.

The physcial memory that holds these data structures is obtained from the system memory manager via memalign to insure that the memory is page aligned. We want to protect this memory from being corrupted, so we invalidate the descriptors that we set up in the global translation that correspond to the memory containing the translation table data structures. This creates a "chicken and the egg" paradox, in that the only way we can modify these data structures is through virtual memory that is now invalidated, and we can't validate it because the page descriptors for that memory are in invalidated memory (confused yet?) So, you will notice that anywhere that page table descriptors (pte's) are modified, we do so by locking out interrupts, momentarily disabling the mmu, accessing the memory with its physical address, enabling the mmu, and then re-enabling interrupts (see ***mmuStateSet*( )**, for example.)

Support for two new page attribute bits are added for PentiumPro's enhanced MMU. They are Global bit (G) and Page-level write-through/back bit (PWT). Global bit indicates a global page when set. When a page is marked global and the page global enable (PGE) bit in register CR4 is set, the page-table or page-directory entry for the page is not invalidated in the TLB when register CR3 is loaded or a task switch occures. This bit is provided to prevent frequently used pages (such as pages that contain kernel or other operating system or executive code) from being flushed from the TLB. Page-level write-through/back bit (PWT) controls the write-through or write- back caching policy of individual pages or page tables. When the PWT bit is set, write-through caching is enabled for the associated page or page table. When the bit is clear, write-back caching is enabled for the associated page and page table. Following macros are used to describe these attribute bits in the physical memory descriptor table **sysPhysMemDesc[ ]** in **sysLib.c**.

```
VM_STATE_WBACK      = use write-back cache policy for the page
VM_STATE_WBACK_NOT  = use write-through cache policy for the page
VM_STATE_GLOBAL     = set page global bit
VM_STATE_GLOBAL_NOT = not set page global bit
```

Support for two page size (4KB and 4MB) are added also. The linear address for 4KB pages is devided into three sections:

```
Page directory entry – bits 22 through 31.
Page table entry     – Bits 12 through 21.
Page offset          – Bits  0 through 11.
```

The linear address for 4MB pages is devided into two sections:

```
Page directory entry – Bits 22 through 31.
Page offset          – Bits  0 through 21.
```

These two page size is configurable by **VM_PAGE_SIZE** macro in **config.h**.

mmuSparcILib

NAME **mmuSparcILib** – ROM MMU initialization (SPARC)

ROUTINES ***mmuSparcRomInit*( )** – initialize the MMU for the ROM (SPARC)

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

ROUTINES
***moduleCreate*( )** – create and initialize a module
***moduleDelete*( )** – delete module ID information (use ***unld*( )** to reclaim space)
***moduleShow*( )** – show the current status for all the loaded modules
***moduleSegGet*( )** – get (delete and return) the first segment from a module
***moduleSegFirst*( )** – find the first segment in a module
***moduleSegNext*( )** – find the next segment in a module
***moduleCreateHookAdd*( )** – add a routine to be called when a module is added
***moduleCreateHookDelete*( )** – delete a previously added module create hook routine
***moduleFindByName*( )** – find a module by name
***moduleFindByNameAndPath*( )** – find a module by file name and path
***moduleFindByGroup*( )** – find a module by group number
***moduleIdListGet*( )** – get a list of loaded modules
***moduleInfoGet*( )** – get information about an object module
***moduleCheck*( )** – verify checksums on all modules
***moduleNameGet*( )** – get the name associated with a module ID
***moduleFlagsGet*( )** – get the flags associated with a module ID

DESCRIPTION This library is a class manager, using the standard VxWorks class/object facilities. The library is used to keep track of which object modules have been loaded into VxWorks, to maintain information about object module segments associated with each module, and to track which symbols belong to which module. Tracking modules makes it possible to list which modules are currently loaded, and to unload them when they are no longer needed.

The module object contains the following information:

– name
– linked list of segments, including base addresses and sizes
– symbol group number
– format of the object module (a.out, COFF, ECOFF, etc.)
– the *symFlag* passed to ***ld*( )** when the module was loaded. (For more information about *symFlag* and the loader, see the manual entry for loadLib.)

Multiple modules with the same name are allowed (the same module may be loaded without first being unloaded) but "find" functions find the most recently created module.

The symbol group number is a unique number for each module, used to identify the module's symbols in the symbol table. This number is assigned by **moduleLib** when a module is created.

In general, users will not access these routines directly, with the exception of ***moduleShow*( )**, which displays information about currently loaded modules. Most calls to this library will be from routines in **loadLib** and **unldLib**.

INCLUDE FILES **moduleLib.h**

SEE ALSO **loadLib**

motCpmEnd

NAME **motCpmEnd** – END style Motorola MC68EN360/MPC800 network interface driver

ROUTINES ***motCpmEndLoad*( )** – initialize the driver and device

DESCRIPTION This module implements the Motorola MC68EN360 QUICC as well as the MPC821 and MPC860 Power-QUICC Ethernet Enhanced network interface driver.

All the above mentioned microprocessors feature a number of Serial Communication Controllers (SCC) that support different serial protocols including IEEE 802.3 and Ethernet CSMA-CD. As a result, when the Ethernet mode of a SCC is selected, by properly programming its general Mode Register (GSMR), they can implement the full set of media access control and channel interface functions those protocol require. However, while the

MC68EN360 QUICC and the MPC860 Power-QUICC support up to four SCCs per unit, the MPC821 only includes two on-chip SCCs.

This driver is designed to support the Ethernet mode of a SCC residing on the CPM processor core, no matter which among the MC68EN360 QUICC or any of the PPC800 Series. In fact, the major differences among these processors, as far as the driver is concerned, are to be found in the mapping of the internal Dual-Port RAM. The driver is generic in the sense that it does not care which SCC is being used. In addition, it poses no constraint on the number of individual units that may be used per board. However, this number should be specified in the bsp through the macro **MAX_SCC_CHANNELS**. The default value for this macro in the driver is 4.

To achieve these goals, the driver requires several target-specific values provided as an input string to the load routine. It also requires some external support routines. These target-specific values and the external support routines are described 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 maintains cache coherency by allocating buffer space using the ***cacheDmaMalloc*( )** routine. This is provided for boards whose host processor use data cache space, e.g. the MPC800 Series. Altough the MC68EN360 does not have cache memory, it may be used in a particular configuration: **MC68EN360 in 040 companion mode** where that is attached to processors that may cache memory. However, 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 END external interface. The only external interface is the ***motCpmEndLoad*( )** routine. The parameters are passed into the ***motCpmEndLoad*( )** function as a single colon-delimited string. The ***motCpmEndLoad*( )** function uses ***strtok*( )** to parse the string, which it expects to be of the following format:

unit:*motCpmAddr*:*ivec*:*sccNum*:*txBdNum*:*rxBdNum*: *txBdBase*: *rxBdBase*:*bufBase*

TARGET-SPECIFIC PARAMETERS

unit
: A convenient holdover from the former model. This parameter is used only in the string name for the driver.

motCpmAddr
: Indicates the address at which the host processor 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, and, ultimately, to program the SCC for proper

operations. This parameter should point to the internal memory of the processor where the SCC physically resides. This location might not necessarily be the Dual-Port RAM of the microprocessor configured as master on the target board.

ivec

This driver configures the host processor 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 the VxWorks system function ***intConnect*( )**.

sccNum

This driver is written to support multiple individual device units. Thus, the multiple units supported by this driver can reside on different chips or on different SCCs within a single host processor. This parameter is used to explicitly state which SCC is being used (SCC1 is most commonly used, thus this parameter most often equals "1").

txBdNum and *rxBdNum*

Specify the number of transmit and receive buffer descriptors (BDs). Each buffer descriptor resides in 8 bytes of the processor'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, although more than a certain amount does not speed up the driver and wastes valuable dual-ported RAM space. If any of these parameters is "NULL", a default value of "32" BDs is used.

txBdBase and *rxBdBase*

Indicate the base location of the transmit and receive buffer descriptors (BDs). They are offsets, in bytes, from the base address of the host processor's 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 fit within dual-ported RAM. This includes any other BDs the target board might be using, including other SCCs, SMCs, and the SPI device. There is no default for these parameters. They must be provided by the user.

bufBase

Tells 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). Each buffer is 1520 bytes. If this parameter is "NONE", space for buffers is obtained by calling ***cacheDmaMalloc*( )** in ***motCpmEndLoad*( )**.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires three external support functions:

***sysXxxEnetEnable*( )**

This is either ***sys360EnetEnable*( )** or ***sysCpmEnetEnable*( )**, based on the actual host processor being used. See below for the actual prototypes. This routine is expected to handle any target-specific functions needed 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. This routine is expected to return OK on success, or ERROR. The driver calls this routine, once per unit, from the ***motCpmEndLoad*( )** routine.

***sysXxxEnetDisable*( )**
This is either ***sys360EnetDisable*( )** or ***sysCpmEnetDisable*( )**, based on the actual host processor being used. See below for the actual prototypes. This routine is expected to handle any target-specific functions required to disable the Ethernet controller. This usually involves disabling the Transmit Enable (TENA) signal. This routine is expected to return OK on success, or ERROR. The driver calls this routine from the ***motCpmEndStop*( )** routine each time a unit is disabled.

***sysXxxEnetAddrGet*( )**
This is either ***sys360EnetAddrGet*( )** or ***sysCpmEnetAddrGet*( )**, based on the actual host processor being used. See below for the actual prototypes. The driver expects this routine to provide the six-byte Ethernet hardware address that is 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 ***motCpmEndLoad*( )** routine.

In the case of the CPU32, the prototypes of the above mentioned support routines are as follows:

```
STATUS sys360EnetEnable (int unit, UINT32 regBase)
void sys360EnetDisable (int unit, UINT32 regBase)
STATUS sys360EnetAddrGet (int unit, u_char * addr)
```

In the case of the PPC860, the prototypes of the above mentioned support routines are as follows:

```
STATUS sysCpmEnetEnable (int unit)
void sysCpmEnetDisable (int unit)
STATUS sysCpmEnetAddrGet (int unit, UINT8 * addr)
```

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 could vary for other architectures. The code size (text) 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 (this is not configurable), 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 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 might share the same cache line. Additionally, the chip's dual-ported RAM must be declared as non-cacheable memory where applicable (for example, when attached to a 68040 processor). For more information, see the *Motorola MC68EN360 User's Manual* , *Motorola MPC860 User's Manual* , *Motorola MPC821 User's Manual*

motFecEnd

NAME

motFecEnd – END style Motorola FEC Ethernet network interface driver

ROUTINES

***motFecEndLoad*( )** – initialize the driver and device

DESCRIPTION

This module implements a Motorola Fast Ethernet Controller (FEC) network interface driver. The FEC is fully compliant with the IEEE 802.3 10Base-T and 100Base-T specifications. Hardware support of the Media Independent Interface (MII) is built-in in the chip.

The FEC establishes a shared memory communication system with the CPU, which is divided into two parts: the Control/Status Registers (CSR), and the buffer descriptors (BD).

The CSRs reside in the MPC860T Communication Controller's internal RAM. They are used for mode control and to extract status information of a global nature. For instance, the types of events that should generate an interrupt, or features like the promiscous mode or the max receive frame lenght may be set programming some of the CSRs properly. Pointers to both the Transmit Buffer Descriptors ring (TBD) and the Receive Buffer Descriptors ring (RBD) are also stored in the CSRs. The CSRs are located in on-chip RAM and must be accessed using the big-endian mode.

The BDs are used to pass data buffers and related buffer information between the hardware and the software. They reside in the host main memory and basically include local status information and a pointer to the actual buffer, again in external memory.

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.

BOARD LAYOUT

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

EXTERNAL INTERFACE

The driver provides the standard external interface, ***motFecEndLoad*( )**, which takes a string of colon-separated parameters. The parameters should be specified in hexadecimal, optionally preceeded by "0x" or a minus sign "-".

The parameter string is parsed using ***strtok_r*( )** and each parameter is converted from a string representation to binary by a call to strtoul(parameter, NULL, 16).

The format of the parameter string is:

"*motCpmAddr*:*ivec*:*bufBase*:*bufSize*:*fifoTxBase*:*fifoRxBase*:*tbdNum*:*rbdNum*:*phyAddr*:*isoPhyAddr*:*phyDefMode*:*userFlags*"

TARGET-SPECIFIC PARAMETERS

motCpmAddr
: Indicates the address at which the host processor presents its internal memory (also known as the dual ported RAM base address). With this address, the driver is able to compute the location of the FEC parameter RAM, and, ultimately, to program the FEC for proper operations.

ivec
: This driver configures the host processor 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 the VxWorks system function ***intConnect*( )**. It is also used to compute the interrupt level (0-7) associated with the FEC interrupt (one of the MPC860T SIU internal interrupt sources). The latter is given as a parameter to ***intEnable*( )**, in order to enable this level interrupt to the PPC core.

bufBase
: The Motorola Fast Ethernet Controller 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 FEC.

This parameter tells the driver that space for the both the TBDs and the RBDs need not be allocated but should be taken from a cache-coherent private memory space provided by the user at the given address. Be aware that memory used for buffers descriptors must be 8-byte aligned and non-cacheable. All the buffer descriptors should fit in the given memory space. If this parameter is "NONE", space for buffer descriptors is obtained by calling ***cacheDmaMalloc*( )** in ***motFecEndLoad*( )**.

bufSize
: The memory size parameter specifies the size of the pre-allocated memory region. If *bufBase* is specified as NONE (-1), the driver ignores this parameter. Otherwise, the driver checks the size of the provided memory region is adequate with respect to the given number of Transmit Buffer Descriptors and Receive Buffer Descriptors.

fifoTxBase
: Indicate the base location of the transmit FIFO, in internal memory. The user does not need to initialize this parameter, as the related FEC register defaults to a proper value

after reset. The specific reset value is microcode dependent. However, if the user wishes to reserve some RAM for other purposes, he may set this parameter to a different value. This should not be less than the default.

If *fifoTxBase* is specified as NONE (-1), the driver ignores it.

fifoRxBase

Indicate the base location of the receive FIFO, in internal memory. The user does not need to initialize this parameter, as the related FEC register defaults to a proper value after reset. The specific reset value is microcode dependent. However, if the user wishes to reserve some RAM for other purposes, he may set this parameter to a different value. This should not be less than the default.

If *fifoRxBase* is specified as NONE (-1), the driver ignores it.

tbdNum

This parameter specifies the number of transmit buffer descriptors (TBDs). Each buffer descriptor resides in 8 bytes of the processor's external RAM space, and each one points to a 1536-byte buffer again in external RAM. If this parameter is less than a minimum number specified in the macro **MOT_FEC_TBD_MIN**, or if it is "NULL", a default value of 64 is used. This default number is kept deliberately hugh, since each packet the driver sends may consume more than a single TBD. This parameter should always equal a even number.

rbdNum

This parameter specifies the number of receive buffer descriptors (RBDs). Each buffer descriptor resides in 8 bytes of the processor's external RAM space, and each one points to a 1536-byte buffer again in external RAM. If this parameter is less than a minimum number specified in the macro **MOT_FEC_RBD_MIN**, or if it is "NULL", a default value of 48 is used. This parameter should always equal a even number.

phyAddr

This parameter specifies the logical address of a MII-compliant physical device (PHY) that is to be used as a physical media on the network. Valid addresses are in the range 0-31. There may be more than one device under the control of the same management interface. If this parameter is "NULL", the default physical layer initialization routine will find out the PHY actual address by scanning the whole range. The one with the lowest address will be chosen.

isoPhyAddr

This parameter specifies the logical address of a MII-compliant physical device (PHY) that is to be electrically isolated by the management interface. Valid addresses are in the range 0-31. If this parameter equals 0xff, the default physical layer initialization routine will assume there is no need to isolate any device. However, this parameter will be ignored unless the **MOT_FEC_USR_PHY_ISO** bit in the *userFlags*is set to one.

phyDefMode

This parameter specifies the operating mode that will be set up by the default physical layer initialization routine in case all the attempts made to establish a valid link failed. If that happens, the first PHY that matches the specified abilities will be chosen to work in that mode, and the physical link will not be tested.

userFlags
This field enables the user to give some degree of customization to the driver, especially as regards the physical layer interface.

MOT_FEC_USR_PHY_NO_AN: the default physical layer initialization routine will exploit the auto-negotiation mechanism as described in the IEEE Std 802.3, to bring a valid link up. According to it, all the link partners on the media will take part to the negotiation process, and the highest priority common denominator technology ability will be chosen. It the user wishes to prevent auto-negotiation from occurring, he may set this bit in the user flags.

MOT_FEC_USR_PHY_TBL: in the auto-negotiation process, PHYs advertise all their technology abilities at the same time, and the result is that the maximum common denominator is used. However, this behaviour may be changed, and the user may affect the order how each subset of PHY's abilities is negotiated. Hence, when the **MOT_FEC_USR_PHY_TBL** bit is set, the default physical layer initialization routine will look at the motFecPhyAnOrderTbl[] table and auto-negotiate a subset of abilities at a time, as suggested by the table itself. It is worth noticing here, however, that if the **MOT_FEC_USR_PHY_NO_AN** bit is on, the above table will be ignored.

MOT_FEC_USR_PHY_NO_FD: the PHY may be set to operate in full duplex mode, provided it has this ability, as a result of the negotiation with other link partners. However, in this operating mode, the FEC will ignore the collision detect and carrier sense signals. If the user wishes not to negotiate full duplex mode, he should set the **MOT_FEC_USR_PHY_NO_FD** bit in the user flags.

MOT_FEC_USR_PHY_NO_HD: the PHY may be set to operate in half duplex mode, provided it has this ability, as a result of the negotiation with other link partners. If the user wishes not to negotiate half duplex mode, he should set the **MOT_FEC_USR_PHY_NO_HD** bit in the user flags.

MOT_FEC_USR_PHY_NO_100: the PHY may be set to operate at 100Mbit/s speed, provided it has this ability, as a result of the negotiation with other link partners. If the user wishes not to negotiate 100Mbit/s speed, he should set the **MOT_FEC_USR_PHY_NO_100** bit in the user flags.

MOT_FEC_USR_PHY_NO_10: the PHY may be set to operate at 10Mbit/s speed, provided it has this ability, as a result of the negotiation with other link partners. To not negotiate 10Mbit/s speed, set the **MOT_FEC_USR_PHY_NO_10** bit in the user flags.

MOT_FEC_USR_PHY_ISO: some boards may have different PHYs controlled by the same management interface. In some cases, there may be the need of electrically isolating some of them from the interface itself, in order to guarantee a proper behaviour on the medium layer. If the user wishes to electrically isolate one PHY from the MII interface, he should set the **MOT_FEC_USR_PHY_ISO** bit and provide its logical address in the *isoPhyAddr*field of the load string. The default behaviour is to not isolate any PHY on the board.

MOT_FEC_USR_SER: the user may set the **MOT_FEC_USR_SER** bit to enable the 7-wire interface instead of the MII which is the default.

MOT_FEC_USR_LOOP: when the **MOT_FEC_USR_LOOP** bit is set, the driver will configure the FEC to work in loopback mode, with the TX signal directly connected to the RX. This mode should only be used for testing.

MOT_FEC_USR_HBC: if the **MOT_FEC_USR_HBC** bit is set, the driver configures the FEC to perform heartbeat check following end of transmisson and the HB bit in the status field of the TBD will be set if the collision input does not assert within the heartbeat window (also see **_func_motFecHbFail**, below). The user does not normally need to set this bit.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires three external support functions:

***sysFecEnetEnable*( )**

```
STATUS sysFecEnetEnable (UINT32 motCpmAddr);
```

This routine is expected to handle any target-specific functions needed to enable the FEC. These functions typically include setting the Port D on the 860T-based board so that the MII interface may be used, and also disabling the IRQ7 signal. This routine is expected to return OK on success, or ERROR. The driver calls this routine, once per device, from the ***motFecEndLoad*( )** routine.

***sysFecEnetDisable*( )**

```
STATUS sysFecEnetDisable (UINT32 motCpmAddr);
```

This routine is expected to perform any target specific functions required to disable the MII interface to the FEC. This involves restoring the default values for all the Port D signals. This routine is expected to return OK on success, or ERROR. The driver calls this routine from the ***motFecEndStop*( )** routine each time a device is disabled.

***sysFecEnetAddrGet*( )**

```
STATUS sysFecEnetAddrGet (UINT32 motCpmAddr, UCHAR * enetAddr);
```

The driver expects this routine to provide the six-byte Ethernet hardware address that is used by this device. This routine must copy the six-byte address to the space provided by *enetAddr*. This routine is expected to return OK on success, or ERROR. The driver calls this routine, once per device, from the ***motFecEndLoad*( )** routine.

_func_motFecPhyInit

```
FUNCPTR _func_motFecPhyInit
```

This driver sets the global variable **_func_motFecPhyInit** to the MII-compliant media initialization routine ***motFecPhyInit*( )**. If the user wishes to exploit a different way to configure the PHY, he may set this variable to his own media initialization routine, tipically in ***sysHwInit*( )**.

_func_motFecHbFail

```
FUNCPTR _func_motFecPhyInit
```

The FEC may be configured to perform heartbeat check following end of transmission, and to generate an interrupt, when this event occurs. If this is the case, and if the global variable **_func_motFecHbFail**is not NULL, the routine referenced to by **_func_motFecHbFail** is called, with a pointer to the driver control structure as parameter. Hence, the user may set this variable to his own heart beat check fail routine, where he can take any action he sees appropriate. The default value for the global variable **_func_motFecHbFail** is NULL.

SYSTEM RESOURCE USAGE

If the driver allocates the memory to share with the Ethernet device, it does so by calling the ***cacheDmaMalloc( )*** routine. For the default case of 64 transmit buffers and 48 receive buffers, the total size requested is 912 bytes, and this includes the 16-byte alignment requirement of the device. 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 this memory region is non-cacheable or if the hardware implements bus snooping. The driver cannot maintain cache coherency for the device because the BDs are asynchronously modified by both the driver and the device, and these fields might share the same cache line.

Data buffers are instead allocated in the external memory through the regular memory allocation routine (memalign), and the related cache lines are then flushed or invalidated as appropriate. The user should not allocate memory for them.

TUNING HINTS

The only adjustable parameters are the number of TBDs and RBDs that will be created at run-time. These parameters are given to the driver when ***motFecEndLoad( )*** is called. There is one RBD associated with each received frame whereas a single transmit packet normally uses more than one TBD. For memory-limited applications, decreasing the number of RBDs may be desirable. Decreasing the number of TBDs below a certain point will provide substantial performance degradation, and is not reccomended. An adequate number of loaning buffers are also pre-allocated to provide more buffering before packets are dropped, but this is not configurable.

The relative priority of the netTask and of the other tasks in the system may heavily affect performance of this driver. Usually the best performance is achieved when the netTask priority equals that of the other applications using the driver.

SPECIAL CONSIDERATIONS

Due to the FEC8 errata in the document: "MPC860 Family Device Errata Reference" available at the Motorola web site, the number of receive buffer descriptors (RBD) for the FEC (see **configNet.h**) is kept deliberately high. According to Motorola, this problem was fixed in Rev. B3 of the silicon. In memory-bound applications, when using the above mentioned revision of the MPC860T processor, the user may decrease the number of RBDs to fit his needs.

SEE ALSO

ifLib, *MPC860T Fast Ethernet Controller (Supplement to the MPC860 User's Manual) Motorola MPC860 User's Manual*

mountLib

NAME

mountLib – Mount protocol library

ROUTINES

***mountdInit*( )** – initialize the mount daemon
***nfsExport*( )** – specify a file system to be NFS exported
***nfsUnexport*( )** – remove a file system from the list of exported file systems

DESCRIPTION

This library implements a mount server to support mounting VxWorks file systems remotely. The mount server is an implementation of version 1 of the mount protocol as defined in RFC 1094. It is closely connected with version 2 of the Network File System Protocol Specification, which in turn is implemented by the library **nfsdLib**.

NOTE

The only routines in this library that are normally called by applications are ***nfsExport*( )** and ***nfsUnexport*( )**. The mount daemon is normally initialized indirectly by ***nfsdInit*( )**.

The mount server is initialized by calling ***mountdInit*( )**. Normally, this is done by ***nfsdInit*( )**, although it is possible to call ***mountdInit*( )** directly if the NFS server is not being initialized. Defining **INCLUDE_NFS_SERVER** enables the call to ***nfsdInit*( )** during the boot process, which in turn calls ***mountdInit*( )**, so there is normally no need to call either routine manually. ***mountdInit*( )** spawns one task, **tMountd**, which registers as an RPC service with the portmapper.

Currently, only **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 the configuration macro **INCLUDE_NFS_SERVER** and rebuild VxWorks.

To initialize a file system to be exported, set **DOS_OPT_EXPORT** 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

This 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)

ROUTINES

***mqPxLibInit*( )** – initialize the POSIX message queue library
***mq_open*( )** – open a message queue (POSIX)
***mq_receive*( )** – receive a message from a message queue (POSIX)
***mq_send*( )** – send a message to a message queue (POSIX)
***mq_close*( )** – close a message queue (POSIX)
***mq_unlink*( )** – remove a message queue (POSIX)
***mq_notify*( )** – notify a task that a message is available on a queue (POSIX)
***mq_setattr*( )** – set message queue attributes (POSIX)
***mq_getattr*( )** – get message queue attributes (POSIX)

DESCRIPTION

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

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

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

MESSAGE QUEUE DESCRIPTOR DELETION

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

The routines in this library conform to POSIX 1003.1b.

INCLUDE FILES

mqueue.h

SEE ALSO

POSIX 1003.1b document, **msgQLib**, *VxWorks Programmer's Guide: Basic OS*

---

mqPxShow

NAME

mqPxShow – POSIX message queue show

ROUTINES

***mqPxShowInit*( )** – initialize the POSIX message queue show facility

DESCRIPTION

This library provides a show routine for POSIX objects.

msgQLib

NAME **msgQLib** – message queue library

ROUTINES ***msgQCreate*( )** – create and initialize a message queue
***msgQDelete*( )** – delete a message queue
***msgQSend*( )** – send a message to a message queue
***msgQReceive*( )** – receive a message from a message queue
***msgQNumMsgs*( )** – get the number of messages queued to a message queue

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

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

ROUTINES

***msgQShowInit*( )** – initialize the message queue show facility
***msgQInfoGet*( )** – get information about a message queue
***msgQShow*( )** – show information about a message queue

DESCRIPTION

This library provides routines to show message queue statistics, such as the task queuing method, messages queued, receivers blocked, etc.

The routine ***msgQshowInit*( )** links the message queue show facility into the VxWorks system. It is called automatically when the message queue show facility is configured into VxWorks using either of the following methods:

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

– If you use the Tornado project facility, select **INCLUDE_MSG_Q_SHOW**.

INCLUDE FILES **msgQLib.h**

SEE ALSO **pipeDrv**, *VxWorks Programmer's Guide: Basic OS*

msgQSmLib

NAME **msgQSmLib** – shared memory message queue library (VxMP Opt.)

ROUTINES ***msgQSmCreate*( )** – create and initialize a shared memory message queue (VxMP Opt.)

DESCRIPTION

This library provides the interface to shared memory message queues. Shared memory message queues allow a variable number of messages (varying in length) to be queued in first-in-first-out order. Any task running on any CPU in the system can send messages to or receive messages from a shared message queue. Tasks can also send to and receive from the same shared message queue. Full-duplex communication between two tasks generally requires two shared message queues, one for each direction.

Shared memory message queues are created with ***msgQSmCreate*( )**. Once created, they can be manipulated using the generic routines for local message queues; for more information on the use of these routines, see the manual entry for **msgQLib**.

MEMORY REQUIREMENTS

The shared memory message queue structure is allocated from a dedicated shared memory partition. This shared memory partition is initialized by the shared memory objects master CPU. The size of this partition is defined by the maximum number of shared message queues, **SM_OBJ_MAX_MSG_Q**.

The message queue buffers are allocated from the shared memory system partition.

RESTRICTIONS

Shared memory message queues differ from local message queues in the following ways:

Interrupt Use:
Shared memory message queues may not be used (sent to or received from) at interrupt level.

Deletion:
There is no way to delete a shared memory message queue and free its associated shared memory. Attempts to delete a shared message queue return ERROR and set **errno** to **S_smObjLib_NO_OBJECT_DESTROY**.

Queuing Style:
The shared message queue task queueing order specified when a message queue is created must be FIFO.

CONFIGURATION

Before routines in this library can be called, the shared memory objects facility must be initialized by calling ***usrSmObjInit*( )**, which is found in **src/config/usrSmObj.c**. This is done automatically from the root task, ***usrRoot*( )**, in **usrConfig.c** if the configuration macro **INCLUDE_SM_OBJ** is defined.

AVAILABILITY

This module is distributed as a component of the unbundled shared objects memory support option, VxMP.

INCLUDE FILES

msgQSmLib.h, **msgQLib.h**, **smMemLib.h**, **smObjLib.h**

SEE ALSO

msgQLib, **smObjLib**, **msgQShow**, ***usrSmObjInit*( )**, *VxWorks Programmer's Guide: Shared Memory Objects*

muxLib

NAME

muxLib – MUX network interface library

ROUTINES

***muxLibInit*( )** – initialize global state for the MUX
***muxDevLoad*( )** – load a driver into the MUX
***muxDevStart*( )** – start a device by calling its start routine
***muxDevStop*( )** – stop a device by calling its stop routine
***muxShow*( )** – all configured Enhanced Network Drivers
***muxBind*( )** – bind a protocol to the MUX given a driver name
***muxSend*( )** – send a packet out on a network interface
***muxPollSend*( )** – send a packet on a network interface
***muxPollReceive*( )** – poll for a packet from a device driver
***muxIoctl*( )** – send control information to the MUX or to a device
***muxMCastAddrAdd*( )** – add a multicast address to multicast table for a device
***muxMCastAddrDel*( )** – delete a multicast address from a device's multicast table
***muxMCastAddrGet*( )** – get the multicast address table from the MUX/Driver
***muxUnbind*( )** – detach a protocol from the specified driver
***muxDevUnload*( )** – remove a driver from the MUX
***muxAddressForm*( )** – form an address into a packet
***muxPacketDataGet*( )** – return the data from a packet
***muxPacketAddrGet*( )** – get addressing information from a packet
***endFindByName*( )** – find a device using its string name
***muxDevExists*( )** – tests whether a device is already loaded into the MUX
***muxAddrResFuncAdd*( )** – add an address resolution function
***muxAddrResFuncGet*( )** – get the address resolution function for ifType/protocol
***muxAddrResFuncDel*( )** – delete an address resolution function

DESCRIPTION

This library provides the routines that define the MUX interface, a facility that handles communication between the data link layer and the network protocol layer. Using the MUX, the VxWorks network stack has decoupled the data link and network layers. Thus, drivers and protocols no longer need knowledge of each other's internals. As a result, the network driver and protocol are nearly independent of each another. This independence makes it much easier to add a new drivers or protocols. For example, if you add a new END, all existing MUX-based protocols can use the new driver. Likewise, if you add a new MUX-based protocol, any existing END can use the MUX to access the new protocol.

INCLUDE FILES

errno.h, **lstLib.h**, **logLib.h**, **string.h**, **m2Lib.h**, **bufLib.h**, **if.h**, **end.h**, **muxLib.h**

SEE ALSO

Network Protocol Toolkit User's Guide

ncr710CommLib

NAME **ncr710CommLib** – common library for **ncr710Lib.c** and **ncr710Lib2.c**

ROUTINES ***ncr710SingleStep*( )** – perform a single-step
***ncr710StepEnable*( )** – enable/disable script single-step

DESCRIPTION Contains **ncr710Lib** and **ncr710Lib2** common driver interfaces which can be called from user code.

SEE ALSO **ncr710Lib.c**, **ncr710Lib2.c**, *NCR 53C710 SCSI I/O Processor Programming Guide, VxWorks Programmer's Guide: I/O System*

ncr710Lib

NAME **ncr710Lib** – NCR 53C710 SCSI I/O Processor (SIOP) library (SCSI-1)

ROUTINES ***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

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)

ROUTINES

***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

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)

ROUTINES

***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

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.

PCI MEMORY ADDRESSING

The global variable ncr810PciMemOffset was created to provide the BSP with a means of changing the **VIRT_TO_PHYS** mapping without changing the functions in the cacheFuncs structures. In generating physical addresses for DMA on the PCI bus, local addresses are passed through the function **CACHE_DMA_VIRT_TO_PHYS** and then the value of ncr810PciMemOffset is added. For backward compatibility, the initial value of ncr810PciMemOffset comes from the macro **PCI_TO_MEM_OFFSET** defined in **ncr810.h**.

I/O MACROS All device access for input and output is done via macros which can be customized for each BSP. These routines are **NCR810_IN_BYTE**, **NCR810_OUT_BYTE**, **NCR810_IN_16**, **NCR810_OUT_16**, **NCR810_IN_32** and **NCR810_OUT_32**. By default, these are defined as generic memory references.

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*

ncr5390Lib

NAME **ncr5390Lib** – NCR5390 SCSI-Bus Interface Controller library (SBIC)

ROUTINES ***ncr5390CtrlInit*( )** – initialize the user-specified fields in an ASC structure
***ncr5390Show*( )** – display the values of all readable NCR5390 chip registers

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 Board Support Package (BSP).

In order to configure the SCSI-1 driver, which depends upon **scsi1Lib**, the ***ncr5390CtrlCreate*( )** routine, defined in **ncr5390Lib**1, must be invoked. Similarly ***ncr5390CtrlCreateScsi2*( )**, defined in **ncr5390Lib**2 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)

ROUTINES ***ncr5390CtrlCreate*( )** – create a control structure for an NCR 53C90 ASC

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 ***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)

ROUTINES ***ncr5390CtrlCreateScsi2*( )** – create a control structure for an NCR 53C90 ASC

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 **ncr5390Lib2**, **scsiLib**, *NCR 53C90A, 53C90B Advanced SCSI Controller*, *VxWorks Programmer's Guide: I/O System*

ne2000End

NAME **ne2000End** – NE2000 END network interface driver

ROUTINES ***ne2000EndLoad*( )** – initialize the driver and device
***ne2000Parse*( )** – parse the init string

DESCRIPTION This module implements the NE2000 Ethernet network interface driver.

EXTERNAL INTERFACE

The only external interface is the ***ne2000EndLoad*( )** routine, which expects the *initString* parameter as input. This parameter passes in a colon-delimited string of the format:

unit:*adrs*:*vecNum*:*intLvl*:*byteAccess*:*usePromEnetAddr*:*offset*

The ***ne2000EndLoad*( )** function uses ***strtok*( )** to parse the string.

TARGET-SPECIFIC PARAMETERS

unit
A convenient holdover from the former model. This parameter is used only in the string name for the driver.

adrs
Tells the driver where to find the ne2000.

vecNum
Configures the ne2000 device to generate hardware interrupts for various events within the device. Thus, it contains an interrupt handler routine. The driver calls ***sysIntConnect*( )** to connect its interrupt handler to the interrupt vector generated as a result of the ne2000 interrupt.

intLvl
This parameter is passed to an external support routine, ***sysLanIntEnable*( )**, which is described below in "External Support Requirements." This routine is called during as part of driver's initialization. It handles any board-specific operations required to allow the servicing of a ne2000 interrupt on targets that use additional interrupt controller devices to help organize and service the various interrupt sources. This parameter makes it possible for this driver to avoid all board-specific knowledge of such devices.

byteAccess
Tells the driver the NE2000 is jumpered to operate in 8-bit mode. Requires that ***SYS_IN_WORD_STRING*( )** and ***SYS_OUT_WORD_STRING*( )** be written to properly access the device in this mode.

usePromEnetAddr
Attempt to get the ethernet address for the device from the on-chip (board) PROM attached to the NE2000. Will fall back to using the BSP-supplied ethernet address if this parameter is 0 or if unable to read the ethernet address.

offset
Specifies the memory alignment offset.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires several external support functions, defined as macros:

```
SYS_INT_CONNECT(pDrvCtrl, routine, arg)
SYS_INT_DISCONNECT (pDrvCtrl, routine, arg)
SYS_INT_ENABLE(pDrvCtrl)
SYS_IN_CHAR(pDrvCtrl, reg, pData)
SYS_OUT_CHAR(pDrvCtrl, reg, pData)
SYS_IN_WORD_STRING(pDrvCtrl, reg, pData)
SYS_OUT_WORD_STRING(pDrvCtrl, reg, pData)
```

These macros allow the driver to be customized for BSPs that use special versions of these routines.

The macro **SYS_INT_CONNECT** is used to connect the interrupt handler to the appropriate vector. By default it is the routine ***intConnect*( )**.

The macro **SYS_INT_DISCONNECT** is used to disconnect the interrupt handler prior to unloading the module. By default this is a dummy routine that returns OK.

The macro **SYS_INT_ENABLE** is used to enable the interrupt level for the end device. It is called once during initialization. By default this is the routine ***sysLanIntEnable*( )**, defined in the module **sysLib.o**.

The macro **SYS_ENET_ADDR_GET** is used to get the ethernet address (MAC) for the device. The single argument to this routine is the **END_DEVICE** pointer. By default this routine copies the ethernet address stored in the global variable ne2000EndEnetAddr into the **END_DEVICE** structure.

The macros **SYS_IN_CHAR**, **SYS_OUT_CHAR**, **SYS_IN_WORD_STRING** and **SYS_OUT_WORD_STRING** are used for accessing the ne2000 device. The default macros map these operations onto ***sysInByte*( )**, ***sysOutByte*( )**, ***sysInWordString*( )** and ***sysOutWordString*( )**.

INCLUDES
end.h endLib.h etherMultiLib.h

SEE ALSO
muxLib, **endLib***Writing and Enhanced Network Driver*

---

nec765Fd

NAME
nec765Fd – NEC 765 floppy disk device driver

ROUTINES
***fdDrv*( )** – initialize the floppy disk driver
***fdDevCreate*( )** – create a device for a floppy disk
***fdRawio*( )** – provide raw I/O access

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

netBufLib

NAME

netBufLib – network buffer library

ROUTINES

netBufLibInit( ) – initialize **netBufLib**
netPoolInit( ) – initialize a **netBufLib**-managed memory pool
netPoolDelete( ) – delete a memory pool
netMblkFree( ) – free an **mBlk** back to its memory pool
netClBlkFree( ) – free a **clBlk**-cluster construct back to the memory pool
netClFree( ) – free a cluster back to the memory pool
netMblkClFree( ) – free an **mBlk**-**clBlk**-cluster construct
netMblkClChainFree( ) – free a chain of **mBlk**-**clBlk**-cluster constructs
netMblkGet( ) – get an **mBlk**
netClBlkGet( ) – get a **clBlk**
netClusterGet( ) – get a cluster from the specified cluster pool
netMblkClGet( ) – get a **clBlk**-cluster and join it to the specified **mBlk**
netTupleGet( ) – get an **mBlk**-**clBlk**-cluster
netClBlkJoin( ) – join a cluster to a **clBlk** structure
netMblkClJoin( ) – join an **mBlk** to a **clBlk**-cluster construct
netClPoolIdGet( ) – return a **CL_POOL_ID** for a specified buffer size
netMblkToBufCopy( ) – copy data from an **mBlk** to a buffer
netMblkDup( ) – duplicate an **mBlk**
netMblkChainDup( ) – duplicate an **mBlk** chain

DESCRIPTION

This library contains routines that you can use to organize and maintain a memory pool that consists of pools of **mBlk** structures, pools of **clBlk**structures, and pools of clusters. The **mBlk** and **clBlk** structures are used to manage the clusters. The clusters are containers for the data described by the **mBlk** and **clBlk** structures.

These structures and the various routines of this library constitute a buffering API that has been designed to meet the needs both of network protocols and network device drivers.

The **mBlk** structure is the primary vehicle for passing data between a network driver and a protocol. However, the **mBlk** structure must first be properly joined with a **clBlk** structure that was previously joined with a cluster. Thus, the actual vehicle for passing data is not merely an **mBlk** structure but an **mBlk**-**clBlk**-cluster construct.

To include **netBufLib** in VxWorks, define **INCLUDE_NETWORK** in **configAll.h**. This also automatically configures VxWorks to call *netBufLibInit*( ).

INCLUDE FILES

netBufLib.h

netDrv

NAME **netDrv** – network remote file I/O driver

ROUTINES ***netDrv*( )** – install the network remote file driver
***netDevCreate*( )** – create a remote file device

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 the configuration macro **INCLUDE_NETWORK** is defined.

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 the manual entry for ***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 fields of the stat structure: st_dev, st_mode, and st_size. st_mode is always filled with **S_IFREG**.

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

LIMITATIONS
The **netDrv** implementation strategy implies that directories cannot always be distinguished from plain files. Thus, ***opendir*( )** does not work for directories mounted on **netDrv** devices, and ***ll*( )** does not flag subdirectories with the label "DIR" in listings from **netDrv** devices.

When the access method is FTP, operations can only be done on files that the FTP server allows to download. In particular it is not possible to stat a directory, doing so will result in "*dirname*: not a plain file" error.

INCLUDE FILES
netDrv.h

SEE ALSO
remLib, **netLib**, **sockLib**, ***hostAdd*( )**, *VxWorks Programmer's Guide: Network*

1

netLib

NAME **netLib** – network interface library

ROUTINES ***netLibInit*( )** – initialize the network package
***netTask*( )** – network task entry point

DESCRIPTION This library contains the network task that runs low-level network interface routines in a task context. The network task executes and removes routines that were added to the job queue. This facility is used by network interfaces in order to have interrupt-level processing at task level.

The routine ***netLibInit*( )** initializes the network and spawns the network task ***netTask*( )**. This is done automatically when the configuration macro **INCLUDE_NETWORK** is defined.

The routine ***netHelp*( )** in **usrLib** displays a summary of the network facilities available from the VxWorks shell.

INCLUDE FILES **netLib.h**

SEE ALSO **routeLib**, **hostLib**, **netDrv**, ***netHelp*( )**, *VxWorks Programmer's Guide: Network*

netShow

NAME **netShow** – network information display routines

ROUTINES ***ifShow*( )** – display the attached network interfaces
***inetstatShow*( )** – display all active connections for Internet protocol sockets
***ipstatShow*( )** – display IP statistics
***netPoolShow*( )** – show pool statistics
***netStackDataPoolShow*( )** – show network stack data pool statistics
***netStackSysPoolShow*( )** – show network stack system pool statistics
***mbufShow*( )** – report mbuf statistics
***netShowInit*( )** – initialize network show routines
***arpShow*( )** – display entries in the system ARP table
***arptabShow*( )** – display the known ARP entries
***routestatShow*( )** – display routing statistics
***routeShow*( )** – display host and network routing tables
***hostShow*( )** – display the host table
***mRouteShow*( )** – print the entries of the routing table

DESCRIPTION

This library provides routines to show various network-related statistics, such as configuration parameters for network interfaces, protocol statistics, socket statistics, and so on.

Interpreting these statistics requires detailed knowledge of Internet network protocols. Information on these protocols can be found in the following books:

– *Internetworking with TCP/IP Volume III*, by Douglas Comer and David Stevens

– *UNIX Network Programming*, by Richard Stevens

– *The Design and Implementation of the 4.3 BSD UNIX Operating System*, by Leffler, McKusick, Karels and Quarterman

The ***netShowInit*( )** routine links the network show facility into the VxWorks system. This is performed automatically if **INCLUDE_NET_SHOW** is defined in **configAll.h**.

SEE ALSO

ifLib, **icmpShow**, **igmpShow**, **tcpShow**, **udpShow**, *VxWorks Programmer's Guide: Network*

nfsdLib

NAME

nfsdLib – Network File System (NFS) server library

ROUTINES

***nfsdInit*( )** – initialize the NFS server
***nfsdStatusGet*( )** – get the status of the NFS server
***nfsdStatusShow*( )** – show the status of the NFS server

DESCRIPTION

This library is an implementation of version 2 of the Network File System Protocol Specification as defined in RFC 1094. It is closely connected with version 1 of the mount protocol, also defined in RFC 1094 and implemented in turn by **mountLib**.

The NFS server is initialized by calling ***nfsdInit*( )**. This is done automatically at boot time if the configuration macro **INCLUDE_NFS_SERVER** is defined.

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 the configuration macro **INCLUDE_NFS_SERVER** and rebuild VxWorks.

To export VxWorks file systems via NFS, you need facilities from both this library and from **mountLib**. To include both, define **INCLUDE_NFS_SERVER** and rebuild VxWorks.

Use the **mountLib** routine ***nfsExport*( )** to export file systems. For an example, see the manual page for **mountLib**.

VxWorks does not normally provide authentication services for NFS requests, and the DOS file system does not provide file permissions. If you need to authenticate incoming requests, see the documentation for ***nfsdInit*( )** and ***mountdInit*( )** for information about authorization hooks.

The following requests are accepted from clients. For details of their use, see RFC 1094, "NFS: Network File System Protocol Specification."

Procedure Name	Procedure Number
NFSPROC_NULL	0
NFSPROC_GETATTR	1
NFSPROC_SETATTR	2
NFSPROC_ROOT	3
NFSPROC_LOOKUP	4
NFSPROC_READLINK	5
NFSPROC_READ	6
NFSPROC_WRITE	8
NFSPROC_CREATE	9
NFSPROC_REMOVE	10
NFSPROC_RENAME	11
NFSPROC_LINK	12
NFSPROC_SYMLINK	13
NFSPROC_MKDIR	14
NFSPROC_RMDIR	15
NFSPROC_READDIR	16
NFSPROC_STATFS	17

AUTHENTICATION AND PERMISSIONS

Currently, no authentication is done on NFS requests. ***nfsdInit*( )** describes the authentication hooks that can be added should authentication be necessary.

Note that the DOS file system does not provide information about ownership or permissions on individual files. Before initializing a dosFs file system, three global variables--**dosFsUserId**, **dosFsGroupId**, and **dosFsFileMode**--can be set to define the user ID, group ID, and permissions byte for all files in all dosFs volumes initialized after setting these variables. To arrange for different dosFs volumes to use different user and group ID numbers, reset these variables before each volume is initialized. See the manual entry for **dosFsLib** for more information.

TASKS

Several NFS tasks are created by ***nfsdInit*( )**. They are:

tMountd
The mount daemon, which handles all incoming mount requests. This daemon is created by ***mountdInit*( )**, which is automatically called from ***nfsdInit*( )**.

tNfsd
The NFS daemon, which queues all incoming NFS requests.

tNfsdX
The NFS request handlers, which dequeues and processes all incoming NFS requests.

Performance of the NFS file system can be improved by increasing the number of servers specified in the ***nfsdInit*( )** call, if there are several different dosFs volumes exported from the same target system. The ***spy*( )** utility can be called to determine whether this is useful for a particular configuration.

SEE ALSO **nfsdLib**

nfsDrv

NAME **nfsDrv** – Network File System (NFS) I/O driver

ROUTINES
***nfsDrv*( )** – install the NFS driver
***nfsDrvNumGet*( )** – return the IO system driver number for the nfs driver
***nfsMount*( )** – mount an NFS file system
***nfsMountAll*( )** – mount all file systems exported by a specified host
***nfsDevShow*( )** – display the mounted NFS devices
***nfsUnmount*( )** – unmount an NFS device
***nfsDevListGet*( )** – create list of all the NFS devices in the system
***nfsDevInfoGet*( )** – read configuration information from the requested NFS device

DESCRIPTION
This driver provides facilities for accessing files transparently over the network via NFS (Network File System). By creating a network device with ***nfsMount*( )**, files on a remote NFS system (such as a UNIX system) can be handled as if they were local.

USER-CALLABLE ROUTINES
The ***nfsDrv*( )** routine initializes the driver. The ***nfsMount*( )** and ***nfsUnmount*( )** routines mount and unmount file systems. The ***nfsMountAll*( )** routine mounts all file systems exported by a specified host.

INITIALIZATION
Before using the network driver, it must be initialized by calling ***nfsDrv*( )**. This routine must be called before any reads, writes, or other NFS calls. This is done automatically when the configuration macro **INCLUDE_NFS** is defined.

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 call:

```
nfsMount ("wrs", "/d0/", NULL);
```

or from the shell:

```
nfsMount "wrs", "/d0/"
```

creates the device **/d0/**. The file **/d0/dog** is accessed by the same name, **/d0/dog**.

Before mounting a file system, the host must already have been created with ***hostAdd*( )**. The routine ***nfsDevShow*( )** displays the mounted NFS devices.

IOCTL FUNCTIONS The NFS driver responds to the following ***ioctl*( )** functions:

FIOGETNAME
Gets the file name of *fd* and copies it to the buffer referenced by *nameBuf*:

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

FIONREAD
Copies to *nBytesUnread* the number of bytes remaining in the file specified by *fd*:

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

FIOSEEK
Sets the current byte offset in the file to the position specified by *newOffset*. If the seek goes beyond the end-of-file, the file grows. The end-of-file pointer gets moved to the new position, and the new space is filled with zeros:

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

FIOSYNC
Flush data to the remote NFS file. It takes no additional argument:

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

FIOWHERE
Returns the current byte position in the file. This is the byte offset of the next byte to be read or written. It takes no additional argument:

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

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

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

FIOFSTATGET
Gets file status information (directory entry data). The argument *statStruct* is a

pointer to a stat structure that is filled with data describing the specified file. Normally, the ***stat*( )** or ***fstat*( )** routine is used to obtain file information, rather than using the **FIOFSTATGET** function directly. See the manual entry for **dirLib**:

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

FIOFSTATFSGET
Gets the file system parameters for and open file descriptor. The argument *statfsStruct* is a pointer to a statfs structure that is filled with data describing the underlying filesystem. Normally, the ***stat*( )** or ***fstat*( )** routine is used to obtain file information, rather than using the **FIOFSTATGET** function directly. See the manual entry for **dirLib**:

```
statfs statfsStruct;
fd = open ("directory", O_RDONLY);
status = ioctl (fd, FIOFSTATFSGET, &statfsStruct);
```

DEFICIENCIES

There is only one client handle/cache per task. Performance is poor if a task is accessing two or more NFS files.

Changing *nfsCacheSize* after a file is open could cause adverse effects. However, changing it before opening any NFS file descriptors should not pose a problem.

INCLUDE FILES

nfsDrv.h, **ioLib.h**, **dirent.h**

SEE ALSO

dirLib, **nfsLib**, ***hostAdd*( )**, ***ioctl*( )**, *VxWorks Programmer's Guide: Network*

nfsLib

NAME

nfsLib – Network File System (NFS) library

ROUTINES

***nfsHelp*( )** – display the NFS help menu
***nfsExportShow*( )** – display the exported file systems of a remote host
***nfsAuthUnixPrompt*( )** – modify the NFS UNIX authentication parameters
***nfsAuthUnixShow*( )** – display the NFS UNIX authentication parameters
***nfsAuthUnixSet*( )** – set the NFS UNIX authentication parameters
***nfsAuthUnixGet*( )** – get the NFS UNIX authentication parameters
***nfsIdSet*( )** – set the ID number of the NFS UNIX authentication parameters

DESCRIPTION

This library provides the client side of services for NFS (Network File System) devices. Most routines in this library should not be called by users, but rather by device drivers.

The driver is responsible for keeping track of file pointers, mounted disks, and cached buffers. This library uses Remote Procedure Calls (RPC) to make the NFS calls.

VxWorks is delivered with NFS disabled. The configuration macro for NFS is **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*

---

nicEvbEnd

NAME **nicEvbEnd** – National Semiconductor ST-NIC Chip network interface driver

ROUTINES

***nicEndLoad*( )** – initialize the driver and device
***nicEvbInitParse*( )** – parse the initialization string

DESCRIPTION

This module implements the National Semiconductor 83902A ST-NIC Ethernet network interface driver.

This driver is non-generic and is for use on the IBM EVB403 board. 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

The only external interface is the ***nicEvbEndLoad*( )** routine, which expects the *initString* parameter as input. This parameter passes in a colon-delimited string of the format:

unit:*nic_addr*:*int_vector*:*int_level*

The ***nicEvbEndLoad*( )** function uses ***strtok*( )** to parse the string.

TARGET-SPECIFIC PARAMETERS

unit
A convenient holdover from the former model. This parameter is used only in the string name for the driver.

nic_addr
Base address for NIC chip

int_vector
Configures the NIC device to generate hardware interrupts for various events within the device. Thus, it contains an interrupt handler routine. The driver calls ***sysIntConnect*( )** to connect its interrupt handler to the interrupt vector.

int_level
This parameter is passed to an external support routine, ***sysLanIntEnable*( )**, which is described below in "External Support Requirements." This routine is called during as part of driver's initialization. It handles any board-specific operations required to allow the servicing of a NIC interrupt on targets that use additional interrupt controller devices to help organize and service the various interrupt sources. This parameter makes it possible for this driver to avoid all board-specific knowledge of such devices.

device restart/reset delay
The global variable nicRestartDelay (UINT32), defined in this file, should be initialized in the BSP ***sysHwInit*( )** routine. nicRestartDelay is used only with PowerPC platform and is equal to the number of time base increments which makes for 1.6 msec. This corresponds to the delay necessary to respect when restarting or resetting the device.

EXTERNAL SUPPORT REQUIREMENTS

This driver requires several external support functions, defined as macros:

```
SYS_INT_CONNECT(pDrvCtrl, routine, arg)
SYS_INT_DISCONNECT (pDrvCtrl, routine, arg)
SYS_INT_ENABLE(pDrvCtrl)
```

There are default values in the source code for these macros. They presume memory-mapped accesses to the device registers and the normal ***intConnect*( )**, and ***intEnable*( )** BSP functions. The first argument to each is the device controller structure. Thus, each has access back to all the device-specific information. Having the pointer in the macro facilitates the addition of new features to this driver.

SYSTEM RESOURCE USAGE

When implemented, this driver requires the following system resources:

– one mutual exclusion semaphore
– one interrupt vector

SEE ALSO **muxLib**

---

ns16550Sio

NAME **ns16550Sio** – NS 16550 UART tty driver

ROUTINES

***ns16550DevInit*( )** – intialize an NS16550 channel
***ns16550IntWr*( )** – handle a transmitter interrupt
***ns16550IntRd*( )** – handle a receiver interrupt
***ns16550IntEx*( )** – miscellaneous interrupt processing
***ns16550Int*( )** – interrupt level processing

DESCRIPTION

This is the driver for the NS16552 DUART. This device includes two universal asynchronous receiver/transmitters, a baud rate generator, and a complete modem control capability.

A **NS16550_CHAN** structure is used to describe the serial channel. This data structure is defined in **ns16550Sio.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.

USAGE

The BSP's ***sysHwInit*( )** routine typically calls ***sysSerialHwInit*( )**, which creates the **NS16550_CHAN** structure and initializes all the values in the 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**).

This driver handles setting of hardware options such as parity(odd, even) and number of data bits(5, 6, 7, 8). Hardware flow control is provided with the handshakes RTS/CTS. The function HUPCL(hang up on last close) is available. When hardware flow control is enabled, the signals RTS and DTR are set TRUE and remain set until a HUPCL is performed.

INCLUDE FILES **drv/sio/ns16552Sio.h**

ntEnd

NAME **ntEnd** – END network interface driver to ULIP for vxSim for Windows NT

ROUTINES
***ntLoad*( )** – initialize the driver and device
***ntParse*( )** – parse the init string
***ntMemInit*( )** – initialize memory for the chip
***ntPollStart*( )** – start polled mode operations
***ntPollStop*( )** – stop polled mode operations
***ntInt*( )** – handle controller interrupt

DESCRIPTION This driver provides a fake ethernet intface to the "ULIP" driver written by WRS. The driver essentially gets packets from vxWorks, and writes them directly to file, where the ULIP driver handles them.

The macro **SYS_ENET_ADDR_GET** is used to get the ethernet address (MAC) for the device. The single argument to this routine is the **NTEND_DEVICE** pointer. By default this routine copies the ethernet address stored in the global variable ntEnetAddr into the **NTEND_DEVICE** structure.

INCLUDES **end.h endLib.h etherMultiLib.h**

SEE ALSO **muxLib**, **endLib***Writing and Enhanced Network Driver*

ntPassFsLib

NAME **ntPassFsLib** – pass-through (to Windows NT) file system library

ROUTINES
***ntPassFsDevInit*( )** – associate a device with ntPassFs file system functions
***ntPassFsInit*( )** – prepare to use the ntPassFs library

DESCRIPTION This module is only used with VxSim simulated versions of VxWorks.

This library provides services for file-oriented device drivers to use the Windows NT file standard. In general, the routines in this library are not to be called directly by users, but rather by the VxWorks I/O System.

INITIALIZING PASSFSLIB

Before any other routines in **ntPassFsLib** can be used, the routine ***ntPassFsInit*( )** must be called to initialize this library. The ***ntPassFsDevInit*( )** routine associates a device name with the **ntPassFsLib**functions. The parameter expected by ***ntPassFsDevInit*( )** is a pointer to a name string, to be used to identify the volume/device. This will be part of the pathname for I/O operations which operate on the device. This name will appear in the I/O system device table, which may be displayed using the ***iosDevShow*( )** routine.

As an example:

```
ntPassFsInit (1);
ntPassFsDevInit ("host:");
```

After the ***ntPassFsDevInit*( )** call has been made, when **ntPassFsLib** receives a request from the I/O system, it calls the Windows NT I/O system to service the request. Only one volume may be created.

READING DIRECTORY ENTRIES

Directories on a ntPassFs volume may be searched using the ***opendir*( )**, ***readdir*( )**, ***rewinddir*( )**, and ***closedir*( )** routines. These calls allow the names of files and sub-directories to be determined.

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

FILE DATE AND TIME

Windows NT file date and time are passed through to VxWorks.

INCLUDE FILES **ntPassFsLib.h**

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

---

ospfLib

NAME **ospfLib** – OSPF version 2 (RFC 1583) routing facilities (OSPF Opt.)

ROUTINES

***m2OspfGeneralGroupGet*( )** – get values of OSPF general group objects (OSPF Opt.)
***m2OspfGeneralGroupSet*( )** – set values of OSPF general group objects (OSPF Opt.)
***m2OspfAreaEntryGet*( )** – get an entry from the OSPF area table (OSPF Opt.)
***m2OspfAreaEntrySet*( )** – set values in an OSPF area entry (OSPF Opt.)
***m2OspfStubAreaEntryGet*( )** – get an OSPF stub area entry (OSPF Opt.)
***m2OspfStubAreaEntrySet*( )** – set values in an OSPF stub area entry (OSPF Opt.)
***m2OspfLsdbEntryGet*( )** – get an OSPF link state database entry (OSPF Opt.)
***m2OspfAreaRangeEntryGet*( )** – get an OSPF area range entry (OSPF Opt.)
***m2OspfAreaRangeEntrySet*( )** – set values in an OSPF area range entry (OSPF Opt.)
***m2OspfHostEntryGet*( )** – get an OSPF host entry (OSPF Opt.)
***m2OspfHostEntrySet*( )** – set values in an OSPF host entry (OSPF Opt.)

***m2OspfIfEntryGet*( )** – get an OSPF interface entry (OSPF Opt.)
***m2OspfIfEntrySet*( )** – set values in an OSPF interface entry (OSPF Opt.)
***m2OspfIfMetricEntryGet*( )** – get an OSPF interface metric entry (OSPF Opt.)
***m2OspfIfMetricEntrySet*( )** – set OSPF interface metric entry values (OSPF Opt.)
***m2OspfVirtIfEntryGet*( )** – get an OSPF virtual interface entry (OSPF Opt.)
***m2OspfVirtIfEntrySet*( )** – set OSPF virtual interface entry values (OSPF Opt.)
***m2OspfNbrEntryGet*( )** – get an OSPF neighbor entry (OSPF Opt.)
***m2OspfNbrEntrySet*( )** – set values in an OSPF neighbor entry (OSPF Opt.)
***m2OspfVirtNbrEntryGet*( )** – get an OSPF virtual neighbor entry (OSPF Opt.)
***ospfExtRouteAdd*( )** – import external route into OSPF domain (OSPF Opt.)
***ospfExtRouteDelete*( )** – delete external route imported into OSPF (OSPF Opt.)
***ospfInit*( )** – function to initialize OSPF routing (OSPF Opt.)
***ospfNbmaDstAdd*( )** – add NBMA destination
***ospfNbmaDstDelete*( )** – delete NBMA destination
***ospfTerminate*( )** – free OSPF resources and delete OSPF tasks

DESCRIPTION

This module implements OSPF Version 2 as specified in (RFC 1583). In addition to implementing the routing tasks, this module includes RFC 1253 compliant interfaces that you can use to configure the OSPF MIBs. These may be invoked directly or called by the relevant method routines of an SNMP agent.

To include OSPF in your image you must first define the **INCLUDE_OSPF** in **configAll.h**. Once the system is up and running you need to invoke the ***ospfInit*( )** call. This call has the following structure:

```
STATUS ospfInit
(
    int priority,         /* priority of tasks */
    int options,          /* ospf task options */
    int stackSize,        /* task stack size */
    int routerId          /* the ID for this router */
    FUNCPTR ospfAuthHook  /* authentication hook */
)
```

After OSPF is up and running, you should configure the OSPF MIB by using the various **m2Ospf** routines. The parameters to these routines are specified in the OSPF MIB as defined in RFC 1253. Explanations for each of the variables may be obtained from the RFC. For additional information on the MIB-II interfaces, please see the manual pages.

EXAMPLE

This section presents a sample configuration as well as the code necessary to make the example work. In the example system, a router is attached to two subnets 160.10.10.00 and 160.10.11.00 with 0xffffff00 as the subnet mask. The interface addresses are 160.10.10.5 and 160.10.11.5.

```
---------------------------------- 160.10.11.0
             160.10.11.5 |
             --------------------------
             |      Interface A       |
             |                        |
             |         Router         |
             |                        |
             |      Interface B       |
             --------------------------
             160.10.10.5 |
 ----------------------------------- 160.10.10.0
```

To set this up programmatically, you would execute the following code:

```
void ospfSetup ()
    {
    /* This is a generic setup for all interfaces in the system. */
    M2_OSPF_AREA_ENTRY area;
    M2_OSPF_IF_ENTRY    intf;
    area.ospfAreaId = 0x2;   /* using area id 2 */
    area.ospfAuthType = 0;   /* no authentication */
    if (m2OspfAreaEntrySet (M2_OSPF_AREA_ID |
        M2_OSPF_AUTH_TYPE, &area) != OK)
        {
        return (ERROR);
        };
    /* First we set up Interface A */
    /* set the interface address */
    intf.ospfIfIpAddress = 0xa00a0a05; /* 160.10.10.5 */

    /* address less interface is false */
    intf.ospfAddressLessIf = 0;

    /* interface area id set to 2 */
    intf.ospfIfAreaId =  2;

    /* router priority */
    intf.ospfIfRtrPriority = 5;

    /* various time   intervals */
    intf.ospfIfTransitDelay   =   1;
    intf.ospfIfRetransInterval  =  3;
    intf.ospfIfHelloInterval = 10;
    intf.ospfIfRtrDeadInterval = 40;
    intf.ospfIfPollInterval = 30;

    /* enable OSPF on interface */
    intf.ospfIfAdminStat = M2_ospfAdminStat_enabled;
```

```
/* set the parameters for this interface */
if(m2OspfIfEntrySet (M2_OSPF_IF_AREA_ID |
    M2_OSPF_IF_RTR_PRIORITY |
    M2_OSPF_IF_RETRANS_INTERVAL |
    M2_OSPF_IF_HELLO_INTERVAL |
    M2_OSPF_IF_RTR_DEAD_INTERVAL |
    M2_OSPF_IF_POLL_INTERVAL |
    M2_OSPF_IF_ADMIN_STAT,
    &intf) != OK)
    {
    return (ERROR);
    }
/* similar sequence for Interface B */
intf.ospfIfIpAddress = 0xa00a0b05; /* 160.10.11.5 */
intf.ospfAddressLessIf  = 0;
intf.ospfIfAreaId   =    2;
intf.ospfIfRtrPriority =  0;
intf.ospfIfTransitDelay  =  1;
intf.ospfIfRetransInterval  =  3;
intf.ospfIfHelloInterval = 10;
intf.ospfIfRtrDeadInterval = 40;
intf.ospfIfPollInterval = 30;
intf.ospfIfAdminStat = 1;

if (m2OspfIfEntrySet (M2_OSPF_IF_AREA_ID |
    M2_OSPF_IF_RTR_PRIORITY |
    M2_OSPF_IF_RETRANS_INTERVAL |
    M2_OSPF_IF_HELLO_INTERVAL |
    M2_OSPF_IF_RTR_DEAD_INTERVAL |
    M2_OSPF_IF_POLL_INTERVAL |
    M2_OSPF_IF_ADMIN_STAT, &intf) != OK)
    {
    return (ERROR);
    }
```

After this code has executed, the system is set up to use OSPF to route between the two interfaces (A and B). The system will now continue to participate in the OSPF routing protocol until either the system is shut off or further calls are made into the system using the m2{*} interfaces. Note that it may not be necessary to set all the parameters as shown above if the default value of the parameter is acceptable for your configuration. Default values are as specified in the MIB (RFC 1253).

INCLUDE FILES **ospfLib.h**

SEE ALSO RFC 1583 and RFC 1253

1

passFsLib

NAME **passFsLib** – pass-through (to UNIX) file system library (VxSim)

ROUTINES ***passFsDevInit*( )** – associate a device with passFs file system functions
***passFsInit*( )** – prepare to use the passFs library

DESCRIPTION This module is only used with VxSim simulated versions of VxWorks.

This library provides services for file-oriented device drivers to use the UNIX file standard. This module takes care of all the buffering, directory maintenance, and file system details that are necessary. In general, the routines in this library are not to be called directly by users, but rather by the VxWorks I/O System.

INITIALIZING PASSFSLIB

Before any other routines in **passFsLib** can be used, the routine ***passFsInit*( )** must be called to initialize this library. The ***passFsDevInit*( )** routine associates a device name with the **passFsLib**functions. The parameter expected by ***passFsDevInit*( )** is a pointer to a name string, to be used to identify the volume/device. This will be part of the pathname for I/O operations which operate on the device. This name will appear in the I/O system device table, which may be displayed using the ***iosDevShow*( )** routine.

As an example:

```
passFsInit (1);
passFsDevInit ("host:");
```

After the ***passFsDevInit*( )** call has been made, when **passFsLib** receives a request from the I/O system, it calls the UNIX I/O system to service the request. Only one volume may be created.

READING DIRECTORY ENTRIES

Directories on a passFs volume may be searched using the ***opendir*( )**, ***readdir*( )**, ***rewinddir*( )**, and ***closedir*( )** routines. These calls allow the names of files and sub-directories to be determined.

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

FILE DATE AND TIME

UNIX file date and time are passed though to VxWorks.

INCLUDE FILES **passFsLib.h**

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

pccardLib

NAME **pccardLib** – PC CARD enabler library

ROUTINES

***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
***pccardTffsEnabler*( )** – enable the PCMCIA-TFFS driver

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

ROUTINES ***pcicInit*( )** – initialize the PCIC chip

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

ROUTINES ***pcicShow*( )** – show all configurations of the PCIC chip

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*( )**.

SEE ALSO **pcicShow**

pcmciaLib

NAME **pcmciaLib** – generic PCMCIA event-handling facilities

ROUTINES ***pcmciaInit*( )** – initialize the PCMCIA event-handling package
***pcmciad*( )** – handle task-level PCMCIA events

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

ROUTINES

*pcmciaShowInit***( )** – initialize all show routines for PCMCIA drivers
*pcmciaShow***( )** – show all configurations of the PCMCIA chip

DESCRIPTION This library provides a show routine that shows the status of the PCMCIA chip and the PC card.

pentiumALib

NAME **pentiumALib** – Pentium and PentiumPro specific routines

ROUTINES

***pentiumCr4Get*( )** – Get a content of CR4 register
***pentiumCr4Set*( )** – Set a specified value to CR4 register
***pentiumPmcStart*( )** – start both PMC0 and PMC1
***pentiumPmcStop*( )** – stop both PMC0 and PMC1
***pentiumPmcStop1*( )** – stop PMC1
***pentiumPmcGet*( )** – get contents of PMC0 and PMC1
***pentiumPmcGet0*( )** – get a content of PMC0
***pentiumPmcGet1*( )** – get a content of PMC1

***pentiumPmcReset*( )** – reset both PMC0 and PMC1
***pentiumPmcReset0*( )** – reset PMC0
***pentiumPmcReset1*( )** – reset PMC1
***pentiumTscGet64*( )** – get 64Bit TSC (Timestamp Counter)
***pentiumTscGet32*( )** – get a lower half of the 64Bit TSC (Timestamp Counter)
***pentiumTscReset*( )** – reset the TSC (Timestamp Counter)
***pentiumMsrGet*( )** – get a content of the specified MSR (Model Specific Register)
***pentiumMsrSet*( )** – set a value to the specified MSR (Model Specific Registers)
***pentiumTlbFlush*( )** – flush TLBs (Translation Lookaside Buffers)
***pentiumSerialize*( )** – execute a serializing instruction CPUID
***pentiumBts*( )** – execute atomic compare-and-exchange instruction to set a bit
***pentiumBtc*( )** – execute atomic compare-and-exchange instruction to clear a bit

DESCRIPTION This module contains Pentium and PentiumPro specific routines written in assembly language.

MCA (Machine Check Architecture)

The Pentium processor introduced a new exception called the machine-check exception (interrupt-18). This exception is used to signal hardware-related errors, such as a parity error on a read cycle. The PentiumPro processor extends the types of errors that can be detected and that generate a machine- check exception. It also provides a new machine-check architecture that records information about a machine-check error and provides the basis for an extended error logging capability.

MCA is enabled and its status registers are cleared zero in ***sysHwInit*( )**. Its registers are accessed by ***pentiumMsrSet*( )** and ***pentiumMsrGet*( )**.

PMC (Performance Monitoring Counters)

The PentiumPro processor has two performance-monitoring counters for use in monitoring internal hardware operations. These counters are duration or event counters that can be programmed to count any of approximately 100 different types of events, such as the number of instructions decoded, number of interrupts received, or number of cache loads.

There are nine routines to interface the PMC. These nine routines are:

```
STATUS pentiumPmcStart
       (
       int pmcEvtSel0;        /* performance event select register 0 */
       int pmcEvtSel1;        /* performance event select register 1 */
       )

void   pentiumPmcStop (void)

void   pentiumPmcStop1 (void)

void   pentiumPmcGet
```

```
        (
        long long int * pPmc0; /* performance monitoring counter 0 */
        long long int * pPmc1; /* performance monitoring counter 1 */
        )

void    pentiumPmcGet0
        (
        long long int * pPmc0; /* performance monitoring counter 0 */
        )

void    pentiumPmcGet1
        (
        long long int * pPmc1; /* performance monitoring counter 1 */
        )

void    pentiumPmcReset (void)

void    pentiumPmcReset0 (void)

void    pentiumPmcReset1 (void)
```

***pentiumPmcStart*( )** starts both PMC0 and PMC1. ***pentiumPmcStop*( )** stops them, and ***pentiumPmcStop1*( )** stops only PMC1. ***pentiumPmcGet*( )** gets contents of PMC0 and PMC1. ***pentiumPmcGet0*( )** gets a content of PMC0, and ***pentiumPmcGet1*( )** gets a content of PMC1. ***pentiumPmcReset*( )** resets both PMC0 and PMC1. ***pentiumPmcReset0*( )** resets PMC0, and ***pentiumPmcReset1*( )** resets PMC1. PMC is enabled in ***sysHwInit*( )**. Selected events in the default configuration are PMC0 = number of hardware interrupts received and PMC1 = number of misaligned data memory references.

MSR (Model Specific Register)

The concept of model-specific registers (MSRs) to control hardware functions in the processor or to monitor processor activity was introduced in the PentiumPro processor. The new registers control the debug extensions, the performance counters, the machine-check exception capability, the machine check architecture, and the MTRRs. The MSRs can be read and written to using the RDMSR and WRMSR instructions, respectively.

There are two routines to interface the MSR. These two routines are:

```
void pentiumMsrGet
     (
     int address,             /* MSR address */
     long long int * pData    /* MSR data */
     )

void pentiumMsrSet
```

```
(
int address,             /* MSR address */
long long int * pData    /* MSR data */
)
```

***pentiumMsrGet*( )** get a content of the specified MSR, and ***pentiumMsrSet*( )** set a value to the specified MSR.

TSC (Time Stamp Counter)

The PentiumPro processor provides a 64-bit time-stamp counter that is incremented every processor clock cycle. The counter is incremented even when the processor is halted by the HLT instruction or the external STPCLK# pin. The time-stamp counter is set to 0 following a hardware reset of the processor. The RDTSC instruction reads the time stamp counter and is guaranteed to return a monotonically increasing unique value whenever executed, except for 64-bit counter wraparound. Intel guarantees, architecturally, that the time-stamp counter frequency and configuration will be such that it will not wraparound within 10 years after being reset to 0. The period for counter wrap is several thousands of years in the PentiumPro and Pentium processors.

There are three routines to interface the TSC. These three routines are:

```
void pentiumTscReset (void)

void pentiumTscGet32 (void)

void pentiumTscGet64
     (
     long long int * pTsc     /* TSC */
     )
```

***pentiumTscReset*( )** reset the TSC. ***pentiumTscGet32*( )** gets a lower half of the 64Bit TSC, and ***pentiumTscGet64*( )** gets whole 64Bit TSC.

Four other routines are provided in this library. They are:

```
void   pentiumTlbFlush (void)

void   pentiumSerialize (void)

STATUS pentiumBts
       (
       char * pFlag                   /* flag address */
       )

STATUS pentiumBtc (pFlag)
       (
       char * pFlag                   /* flag address */
       )
```

***pentiumTlbFlush*( )** flushs TLBs (Translation Lookaside Buffers). ***pentiumSerialize*( )** does serialization by executing CPUID instruction. ***pentiumBts*( )** executes an atomic compare-and-exchange instruction to set a bit. ***pentiumBtc*( )** executes an atomic compare-and-exchange instruction to clear a bit.

SEE ALSO *Pentium, PentiumPro Family Developer's Manual*

pentiumLib

NAME **pentiumLib** – Pentium and PentiumPro library

ROUTINES
***pentiumMtrrEnable*( )** – enable MTRR (Memory Type Range Register)
***pentiumMtrrDisable*( )** – disable MTRR (Memory Type Range Register)
***pentiumMtrrGet*( )** – get MTRRs to a specified MTRR table
***pentiumMtrrSet*( )** – set MTRRs from specified MTRR table with WRMSR instruction.

DESCRIPTION This library provides Pentium and PentiumPro specific routines.

MTRR (Memory Type Range Register) are a new feature introduced in the PentiumPro processor that allow the processor to optimize memory operations for different types of memory, such as RAM, ROM, frame buffer memory, and memory-mapped IO. MTRRs configure an internal map of how physical address ranges are mapped to various types of memory. The processor uses this internal map to determine the cacheability of various physical memory locations and the optimal method of accessing memory locations. For example, if a memory location is specified in an MTRR as write-through memory, the processor handles accesses to this location as follows. It reads data from that location in lines and caches the read data or maps all writes to that location to the bus and updates the cache to maintain cache coherency. In mapping the physical address space with MTRRs, the processor recognizes five types of memory: uncacheable (UC), write-combining (WC), write-through (WT), write-protected (WP), and write-back (WB).

There are one table – **sysMtrr[]** in **sysLib.c** – and four routines to interface the MTRR. These four routines are:

```
void pentiumMtrrEnable (void)

void pentiumMtrrDisable (void)

void pentiumMtrrGet
    (
    MTRR * pMtrr              /* MTRR table */
    )

void pentiumMtrrSet (void)
```

```
(
MTRR * pMtrr               /* MTRR table */
)
```

***pentiumMtrrEnable*( )** enables MTRR, ***pentiumMtrrDisable*( )** disables MTRR. ***pentiumMtrrGet*( )** gets MTRRs to the specified MTRR table. ***pentiumMtrrGet*( )** sets MTRRs from the specified MTRR table. The MTRR table is defined as follows:

```
typedef struct mtrr_fix         /* MTRR - fixed range register */
    {
    char type[8];               /* address range: [0]=0-7 ... [7]=56-63 */
    } MTRR_FIX;

typedef struct mtrr_var         /* MTRR - variable range register */
    {
    long long int base;         /* base register */
    long long int mask;         /* mask register */
    } MTRR_VAR;

typedef struct mtrr             /* MTRR */
    {
    int cap[2];                 /* MTRR cap register */
    int deftype[2];             /* MTRR defType register */
    MTRR_FIX fix[11];           /* MTRR fixed range registers */
    MTRR_VAR var[8];            /* MTRR variable range registers */
    } MTRR;
```

Fixed Range Register's type array can be one of following memory types. **MTRR_UC** (uncacheable), **MTRR_WC** (write-combining), **MTRR_WT** (write-through), **MTRR_WP** (write-protected), and **MTRR_WB** (write-back). MTRR is enabled in ***sysHwInit*( )**.

SEE ALSO *Pentium, PentiumPro Family Developer's Manual*

pentiumShow

NAME **pentiumShow** – Pentium and PentiumPro specific show routines

ROUTINES ***pentiumMcaShow*( )** – show MCA (Machine Check Architecture) registers
***pentiumPmcShow*( )** – show PMCs (Performance Monitoring Counters)

DESCRIPTION This library provides Pentium and PentiumPro specific show routines.

***pentiumMcaShow*( )** shows Machine Check Global Control Registers and Error Reporting Register Banks. ***pentiumPmcShow*( )** shows PMC0 and PMC1, and reset them if the parameter zap is TRUE.

SEE ALSO *VxWorks Programmer's Guide: Configuration*

pingLib

NAME **pingLib** – Packet InterNet Grouper (PING) library

ROUTINES ***pingLibInit*( )** – initialize the ***ping*( )** utility
***ping*( )** – test that a remote host is reachable

DESCRIPTION This library contains the ***ping*( )** utility, which tests the reachability of a remote host.

The routine ***ping*( )** is typically called from the VxWorks shell to check the network connection to another VxWorks target or to a UNIX host. ***ping*( )** may also be used programmatically by applications that require such a test. The remote host must be running TCP/IP networking code that responds to ICMP echo request packets. The ***ping*( )** routine is re-entrant, thus may be called by many tasks concurrently.

The routine ***pingLibInit*( )** initializes the ***ping*( )** utility and allocates resources used by this library. It is called automatically when the configuration macro **INCLUDE_PING** is defined.

pipeDrv

NAME **pipeDrv** – pipe I/O driver

ROUTINES
***pipeDrv*( )** – initialize the pipe driver
***pipeDevCreate*( )** – create a pipe device

DESCRIPTION
The pipe driver provides a mechanism that lets tasks communicate with each other through the standard I/O interface. Pipes can be read and written with normal ***read*( )** and ***write*( )** calls. The pipe driver is initialized with ***pipeDrv*( )**. Pipe devices are created with ***pipeDevCreate*( )**.

The pipe driver uses the VxWorks message queue facility to do the actual buffering and delivering of messages. The pipe driver simply provides access to the message queue facility through the I/O system. The main differences between using pipes and using message queues directly are:

– pipes are named (with I/O device names).

– pipes use the standard I/O functions -- ***open*( )**, ***close*( )**, ***read*( )**, ***write*( )** -- while message queues use the functions ***msgQSend*( )** and ***msgQReceive*( )**.

– pipes respond to standard ***ioctl*( )** functions.

– pipes can be used in a ***select*( )** call.

– message queues have more flexible options for timeouts and message priorities.

– pipes are less efficient than message queues because of the additional overhead of the I/O system.

INSTALLING THE DRIVER
Before using the driver, it must be initialized and installed by calling ***pipeDrv*( )**. This routine must be called before any pipes are created. It is called automatically by the root task, ***usrRoot*( )**, in **usrConfig.c** when the configuration macro **INCLUDE_PIPES** is defined.

CREATING PIPES
Before a pipe can be used, it must be created with ***pipeDevCreate*( )**. For example, to create a device pipe "/pipe/demo" with up to 10 messages of size 100 bytes, the proper call is:

```
pipeDevCreate ("/pipe/demo", 10, 100);
```

USING PIPES
Once a pipe has been created it can be opened, closed, read, and written just like any other I/O device. Often the data that is read and written to a pipe is a structure of some type. Thus, the following example writes to a pipe and reads back the same data:

```
{
int fd;
struct msg outMsg;
```

```
struct msg inMsg;
int len;
fd = open ("/pipe/demo", O_RDWR);
write (fd, &outMsg, sizeof (struct msg));
len = read (fd, &inMsg, sizeof (struct msg));
close (fd);
}
```

The data written to a pipe is kept as a single message and will be read all at once in a single read. If ***read*()** is called with a buffer that is smaller than the message being read, the remainder of the message will be discarded. Thus, pipe I/O is "message oriented" rather than "stream oriented." In this respect, VxWorks pipes differ significantly from UNIX pipes which are stream oriented and do not preserve message boundaries.

WRITING TO PIPES FROM INTERRUPT SERVICE ROUTINES

Interrupt service routines (ISR) can write to pipes, providing one of several ways in which ISRs can communicate with tasks. For example, an interrupt service routine may handle the time-critical interrupt response and then send a message on a pipe to a task that will continue with the less critical aspects. However, the use of pipes to communicate from an ISR to a task is now discouraged in favor of the direct message queue facility, which offers lower overhead (see the manual entry for **msgQLib** for more information).

SELECT CALLS

An important feature of pipes is their ability to be used in a ***select*()** call. The ***select*()** routine allows a task to wait for input from any of a selected set of I/O devices. A task can use ***select*()** to wait for input from any combination of pipes, sockets, or serial devices. See the manual entry for ***select*()**.

IOCTL FUNCTIONS

Pipe devices respond to the following ***ioctl*()** functions. These functions are defined in the header file **ioLib.h**.

FIOGETNAME
Gets the file name of fd and copies it to the buffer referenced by *nameBuf*:

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

FIONREAD
Copies to *nBytesUnread* the number of bytes remaining in the first message in the pipe:

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

FIONMSGS
Copies to *nMessages* the number of discrete messages remaining in the pipe:

```
status = ioctl (fd, FIONMSGS, &nMessages);
```

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

ROUTINES
***ppc403DummyCallback*( )** – dummy callback routine
***ppc403DevInit*( )** – initialize the serial port unit
***ppc403IntWr*( )** – handle a transmitter interrupt
***ppc403IntRd*( )** – handle a receiver interrupt
***ppc403IntEx*( )** – handle error interrupts

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

ROUTINES

***ppc860DevInit*( )** – initialize the SMC
***ppc860Int*( )** – handle an SMC interrupt

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

ROUTINES

***pppHookAdd*( )** – add a hook routine on a unit basis
***pppHookDelete*( )** – delete a hook routine on a unit basis

DESCRIPTION

This library provides routines to add and delete connect and disconnect routines. The connect routine, added on a unit basis, is called before the initial phase of link option negotiation. The disconnect routine, added on a unit basis is called before the PPP connection is closed. These connect and disconnect routines can be used to hook up additional software. If either connect or disconnect hook returns ERROR, the connection is terminated immediately.

This library is automatically linked into the VxWorks system image when the configuration macro **INCLUDE_PPP** is defined.

INCLUDE FILES

pppLib.h

SEE ALSO

pppLib, *VxWorks Programmer's Guide: Network*

pppLib

NAME **pppLib** – Point-to-Point Protocol library

ROUTINES ***pppInit*( )** – initialize a PPP network interface
***pppDelete*( )** – delete a PPP network interface

DESCRIPTION This library implements the VxWorks Point-to-Point Protocol (PPP) facility. PPP allows VxWorks to communicate with other machines by sending encapsulated multi-protocol datagrams over a point-to-point serial link. VxWorks may have up to 16 PPP interfaces active at any one time. Each individual interface (or "unit") operates independent of the state of other PPP units.

USER-CALLABLE ROUTINES

PPP network interfaces are initialized using the ***pppInit*( )** routine. This routine's parameters specify the unit number, the name of the serial interface (*tty*) device, Internet (IP) addresses for both ends of the link, the interface baud rate, an optional pointer to a configuration options structure, and an optional pointer to a configuration options file. The ***pppDelete*( )** routine deletes a specified PPP interface.

DATA ENCAPSULATION

PPP uses HDLC-like framing, in which five header and three trailer octets are used to encapsulate each datagram. In environments where bandwidth is at a premium, the total encapsulation may be shortened to four octets with the available address/control and protocol field compression options.

LINK CONTROL PROTOCOL

PPP incorporates a link-layer protocol called Link Control Protocol (LCP), which is responsible for the link set up, configuration, and termination. LCP provides for automatic negotiation of several link options, including datagram encapsulation format, user authentication, and link monitoring (LCP echo request/reply).

NETWORK CONTROL PROTOCOLS

PPP's Network Control Protocols (NCP) allow PPP to support different network protocols. VxWorks supports only one NCP, the Internet Protocol Control Protocol (IPCP), which allows the establishment and configuration of IP over PPP links. IPCP supports the negotiation of IP addresses and TCP/IP header compression (commonly called "VJ" compression).

AUTHENTICATION The VxWorks PPP implementation supports two separate user authentication protocols: the Password Authentication Protocol (PAP) and the Challenge-Handshake Authentication Protocol (CHAP). While PAP only authenticates at the time of link establishment, CHAP may be configured to periodically require authentication throughout the life of the link. Both protocols are independent of one another, and either may be configured in through the PPP options structure or options file.

IMPLEMENTATION

Each VxWorks PPP interface is handled by two tasks: the daemon task (tPPP*unit*) and the write task (tPPP*unit*Wrt).

The daemon task controls the various PPP control protocols (LCP, IPCP, CHAP, and PAP). Each PPP interface has its own daemon task that handles link set up, negotiation of link options, link-layer user 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* ring buffer and processes it according to PPP framing rules. The packet is then queued either to the IP input queue or to the PPP daemon task input queue.

INCLUDE FILES

pppLib.h

SEE ALSO

ifLib, **tyLib**, **pppSecretLib**, **pppShow**, *VxWorks Programmer's Guide: Network*, *RFC-1332: The PPP Internet Protocol Control Protocol (IPCP)* , *RFC-1334: PPP Authentication Protocols* , *RFC-1548: The Point-to-Point Protocol (PPP)* , *RFC-1549: PPP in HDLC Framing*

ACKNOWLEDGEMENT

This program is based on original work done by Paul Mackerras of Australian National University, Brad Parker, Greg Christy, Drew D. Perkins, Rick Adams, and Chris Torek.

pppSecretLib

NAME **pppSecretLib** – PPP authentication secrets library

ROUTINES
***pppSecretAdd*( )** – add a secret to the PPP authentication secrets table
***pppSecretDelete*( )** – delete a secret from the PPP authentication secrets table

DESCRIPTION This library provides routines to create and manipulate a table of "secrets" for use with Point-to-Point Protocol (PPP) user authentication protocols. The secrets in the secrets table can be searched by peers on a PPP link so that one peer (client) can send a secret word to the other peer (server). If the client cannot find a suitable secret when required to do so, or the secret received by the server is not valid, the PPP link may be terminated.

This library is automatically linked into the VxWorks system image when the configuration macro **INCLUDE_PPP** is defined.

INCLUDE FILES **pppLib.h**

SEE ALSO **pppLib**, **pppShow**, *VxWorks Programmer's Guide: Network*

pppShow

NAME **pppShow** – Point-to-Point Protocol show routines

ROUTINES
***pppInfoShow*( )** – display PPP link status information
***pppInfoGet*( )** – get PPP link status information
***pppstatShow*( )** – display PPP link statistics
***pppstatGet*( )** – get PPP link statistics
***pppSecretShow*( )** – display the PPP authentication secrets table

DESCRIPTION This library provides routines to show Point-to-Point Protocol (PPP) link status information and statistics. Also provided are routines that programmatically access this same information.

This library is automatically linked into the VxWorks system image when the configuration macro **INCLUDE_PPP** is defined.

INCLUDE FILES **pppLib.h**

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

proxyArpLib

NAME

proxyArpLib – proxy Address Resolution Protocol (ARP) library

ROUTINES

***proxyArpLibInit*( )** – initialize proxy ARP
***proxyNetCreate*( )** – create a proxy ARP network
***proxyNetDelete*( )** – delete a proxy network
***proxyNetShow*( )** – show proxy ARP networks
***proxyPortFwdOn*( )** – enable broadcast forwarding for a particular port
***proxyPortFwdOff*( )** – disable broadcast forwarding for a particular port
***proxyPortShow*( )** – show enabled ports

DESCRIPTION

This library provides transparent network access by using the Address Resolution Protocol (ARP) to make logically distinct networks appear as one logical network (that is, 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 (that is, 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.

For more information on proxy ARP, see the *VxWorks Programmer's Guide: Network*

INCLUDE FILES

proxyArpLib.h

SEE ALSO

proxyLib, RFC 925, RFC 1027, RFC 826, *Network Programmer's Guide VxWorks Programmer's Guide: Network*

proxyLib

NAME **proxyLib** – proxy Address Resolution Protocol (ARP) client library

ROUTINES ***proxyReg*( )** – register a proxy client
***proxyUnreg*( )** – unregister a proxy client

DESCRIPTION This library implements the client side of the proxy Address Resolution Protocol (ARP). It allows a VxWorks target to register itself as a proxy client by calling ***proxyReg*( )** and to unregister itself by calling ***proxyUnreg*( )**.

Both commands take an interface name and an IP address as arguments. The interface, *ifName*, specifies the interface through which to send the message. *ifName* must be a backplane interface. *proxyAddr* is the IP address associated with the interface *ifName*.

INCLUDE FILES **proxyArpLib.h**

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

ptyDrv

NAME **ptyDrv** – pseudo-terminal driver

ROUTINES ***ptyDrv*( )** – initialize the pseudo-terminal driver
***ptyDevCreate*( )** – create a pseudo terminal

DESCRIPTION The pseudo-terminal driver provides a tty-like interface between a master and slave process, typically in network applications. The master process simulates the "hardware" side of the driver (e.g., a USART serial chip), while the slave process is the application program that normally talks to the driver.

USER-CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. However, the following routines must be called directly: ***ptyDrv*( )** to initialize the driver, 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**.

CAVEAT

Pseudo-terminal devices cannot be deleted and the associated memory cannot be reclaimed.

INCLUDE FILES

ioLib.h, **ptyDrv.h**

SEE ALSO

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

ramDrv

NAME

ramDrv – RAM disk driver

ROUTINES

***ramDrv*( )** – prepare a RAM disk driver for use (optional)
***ramDevCreate*( )** – create a RAM disk device

DESCRIPTION

This driver emulates a disk driver, but actually keeps all data in memory. The memory location and size are specified when the "disk" is created. The RAM disk feature is useful when data must be preserved between boots of VxWorks or when sharing data between CPUs.

USER-CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. Two routines, however, can be called directly by the user. The first, ***ramDrv( )***, provides no real function except to parallel the initialization function found in true disk device drivers. A call to ***ramDrv( )*** is not required to use the RAM disk driver. However, the second routine, ***ramDevCreate( )***, must be called directly to create RAM disk devices.

Once the device has been created, it must be associated with a name and file system (dosFs, rt11Fs, or rawFs). This is accomplished by passing the value returned by ***ramDevCreate( )***, a pointer to a block device structure, to the file system's device initialization routine or make-file-system routine. See the manual entry ***ramDevCreate( )*** for a more detailed discussion.

IOCTL FUNCTIONS

The RAM driver is called in response to ***ioctl( )*** codes in the same manner as a normal disk driver. When the file system is unable to handle a specific ***ioctl( )*** request, it is passed to the **ramDrv** driver. Although there is no physical device to be controlled, **ramDrv** does handle a **FIODISKFORMAT** request, which always returns OK. All other ***ioctl( )*** requests return an error and set the task's **errno** to **S_ioLib_UNKNOWN_REQUEST**.

INCLUDE FILE

ramDrv.h

SEE ALSO

dosFsDevInit( ), ***dosFsMkfs( )***, ***rt11FsDevInit( )***, ***rt11FsMkfs( )***, ***rawFsDevInit( )***, *VxWorks Programmer's Guide: I/O System, Local File Systems*

---

rawFsLib

NAME

rawFsLib – raw block device file system library

ROUTINES

rawFsDevInit( ) – associate a block device with raw volume functions
rawFsInit( ) – prepare to use the raw volume library
rawFsModeChange( ) – modify the mode of a raw device volume
rawFsReadyChange( ) – notify **rawFsLib** of a change in ready status
rawFsVolUnmount( ) – disable a raw device volume

DESCRIPTION

This library provides basic services for disk devices that do not use a standard file or directory structure. The disk volume is treated much like a large file. Portions of it may be read, written, or the current position within the disk may be changed. However, there is no high-level organization of the disk into files or directories.

USING THIS LIBRARY

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

The ***rawFsInit*( )** routine is the principal initialization function; it need only be called once, regardless of how many rawFs devices will be used.

A separate rawFs routine is used for device initialization. For each rawFs device, ***rawFsDevInit*( )** must be called to install the device.

Several routines are provided to inform the file system of changes in the system environment. The ***rawFsModeChange*( )** routine may be used to modify the readability or writability of a particular device. The ***rawFsReadyChange*( )** routine is used to inform the file system that a disk may have been swapped and that the next disk operation should first remount the disk. The ***rawFsVolUnmount*( )** routine informs the file system that a particular device should be synchronized and unmounted, generally in preparation for a disk change.

INITIALIZATION

Before any other routines in **rawFsLib** can be used, ***rawFsInit*( )** must be called to initialize the library. This call specifies the maximum number of raw device file descriptors that can be open simultaneously and allocates memory for that many raw file descriptors. Any attempt to open more raw device file descriptors than the specified maximum will result in errors from ***open*( )** or ***creat*( )**.

During the ***rawFsInit*( )** call, the raw device library is installed as a driver in the I/O system driver table. The driver number associated with it is then placed in a global variable, **rawFsDrvNum**.

This initialization is enabled when the configuration macro **INCLUDE_RAWFS** is defined; ***rawFsInit*( )** is then called from the root task, ***usrRoot*( )**, in **usrConfig.c**.

DEFINING A RAW DEVICE

To use this library for a particular device, the device structure used by the device driver must contain, as the very first item, a 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

ROUTINES ***reboot*( )** – reset network devices and transfer control to boot ROMs
***rebootHookAdd*( )** – add a routine to be called at reboot

DESCRIPTION This library provides reboot support. To restart VxWorks, the routine ***reboot*( )** can be called at any time by typing CTRL-X from the shell. Shutdown routines can be added with ***rebootHookAdd*( )**. These are typically used to reset or synchronize hardware. For example, **netLib** adds a reboot hook to cause all network interfaces to be reset. Once the reboot hooks have been run, ***sysToMonitor*( )** is called to transfer control to the boot ROMs. For more information, see the manual entry for bootInit.

DEFICIENCIES The order in which hooks are added is the order in which they are run. As a result, **netLib** will kill the network, and no user-added hook routines will be able to use the network. There is no ***rebootHookDelete*( )** routine.

INCLUDE FILES **rebootLib.h**

SEE ALSO **sysLib**, bootConfig, bootInit

remLib

NAME

remLib – remote command library

ROUTINES

***rcmd*( )** – execute a shell command on a remote machine
***rresvport*( )** – open a socket with a privileged port bound to it
***remCurIdGet*( )** – get the current user name and password
***remCurIdSet*( )** – set the remote user name and password
***iam*( )** – set the remote user name and password
***whoami*( )** – display the current remote identity
***bindresvport*( )** – bind a socket to a privileged IP port

DESCRIPTION

This library provides routines 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*

resolvLib

NAME

resolvLib – DNS resolver library

ROUTINES

***resolvInit*( )** – initialize the resolver library
***resolvGetHostByName*( )** – query the DNS server for the IP address of a host
***resolvGetHostByAddr*( )** – query the DNS server for the host name of an IP address
***resolvParamsSet*( )** – set the parameters which control the resolver library
***resolvParamsGet*( )** – get the parameters which control the resolver library
***resolvDNExpand*( )** – expand a DNS compressed name from a DNS packet
***resolvDNComp*( )** – compress a DNS name in a DNS packet
***resolvQuery*( )** – construct a query, send it, wait for a response
***resolvMkQuery*( )** – create all types of DNS queries
***resolvSend*( )** – send a pre-formatted query and return the answer

DESCRIPTION

This library provides the client-side services for DNS (Domain Name Service) queries. DNS queries come from applications that require translation of IP addresses to host names and back. If you include this library in VxWorks, it extends the services of the host library. The interface to this library is described in **hostLib**. The **hostLib** interface uses resolver services to get IP and host names. In addition, the resolver can query multiple DNS servers, if necessary, to add redundancy for queries.

There are two interfaces available for the resolver library. One is a high-level interface suitable for most applications. The other is also a low-level interface for more specialized applications, such as mail protocols.

USING THIS LIBRARY

By default, a VxWorks build does not include the resolver code. In addition, VxWorks is delivered with the resolver library disabled. To include the resolver library in the VxWorks image, edit **config/all/configAll.h** and include the definition:

```
#define INCLUDE_DNS_RESOLVER
```

To enable the resolver services, you need to redefine only one DNS server IP address, changing it from a place-holder value to an actual value. Additional DNS server IP addresses can be configured using ***resolvParamsSet*( )**. To do the initial configuration, edit **configAll.h**, and enter the correct IP address for your domain server in the definition:

```
#define RESOLVER_DOMAIN_SERVER  "90.0.0.3"
```

If you do not provide a valid IP address, resolver initialization fails. You also need to configure the domain to which your resolver belongs. To do this, edit **configAll.h** and enter the correct domain name for your organization in the definition:

```
#define RESOLVER_DOMAIN  "wrs.com"
```

The last and most important step is to make sure that you have a route to the configured DNS server. If your VxWorks image includes a routing protocol, such as RIP or OSPF, the routes are created for you automatically. Otherwise, you must use ***routeAdd*( )** or ***mRouteAdd*( )** to add the routes to the routing table.

The resolver library comes with a debug option. To turn on debugging, edit **configAll.h** to include the define:

```
#define INCLUDE_DNS_DEBUG
```

This include makes VxWorks print a log of the resolver queries to the console. This feature assumes a single task. Thus, if you are running multiple tasks, your output to the console is a garble of messages from all the tasks.

The resolver library uses UDP to send queries to the DNS server and expects the DNS server to handle recursion. You can change the resolver parameters at any time after the library has been initialized with ***resolvInit*( )**. However, it is strongly recommended that you change parameters only shortly after initialization, or when there are no other tasks accessing the resolver library.

Your procedure for changing any of the resolver parameter should start with a call to ***resolvParamsGet*( )** to retrieve the active parameters. Then you can change the query order (defaults to query DNS server only), the domain name, or add DNS server IP addresses. After the parameters are changed, call ***resolvParamsSet*( )**. For the values you can use when accessing resolver library services, see the header files **resolvLib.h**, **resolv/resolv.h**, and **resolv/nameser.h**.

INCLUDE FILES **resolvLib.h**

SEE ALSO **hostLib**

ripLib

NAME **ripLib** – Routing Information Protocol (RIP) v1 and v2 library

ROUTINES
***ripLibInit*( )** – initialize the RIP routing library
***ripRouteShow*( )** – display the internal routing table maintained by RIP
***ripAuthHookAdd*( )** – add an authentication hook to a RIP interface
***ripAuthHookDelete*( )** – remove an authentication hook from a RIP interface
***ripAuthHook*( )** – sample authentication hook
***ripLeakHookAdd*( )** – add a hook to bypass the RIP and kernel routing tables
***ripLeakHookDelete*( )** – remove a table bypass hook from a RIP interface
***ripSendHookAdd*( )** – add an update filter to a RIP interface
***ripSendHookDelete*( )** – remove an update filter from a RIP interface
***ripIfSearch*( )** – add new interfaces to the internal list
***ripIfReset*( )** – alter the RIP configuration after an interface changes
***ripFilterEnable*( )** – activate strict border gateway filtering
***ripFilterDisable*( )** – prevent strict border gateway filtering
***ripShutdown*( )** – terminate all RIP processing
***ripDebugLevelSet*( )** – specify amount of debugging output

DESCRIPTION This library implements versions 1 and 2 of the Routing Information Protocol (RIP). The protocol is intended to operate as an interior gateway protocol within a relatively small network with a longest path of 15 hops.

HIGH-LEVEL INTERFACE

The ***ripLibInit*( )** routine links this library into the VxWorks image and begins a RIP session. This happens automatically if **INCLUDE_RIP** is defined at the time the image is built. Once started, RIP will maintain the network routing table until deactivated by a call to the ***ripShutdown*( )** routine, which will remove all route entries and disable the RIP library routines. All RIP requests and responses are handled as defined in the RFC specifications. RFC 1058 defines the basic protocol operation and RFC 1723 details the extensions which implement version 2.

When acting as a supplier, outgoing route updates are filtered using simple split horizon. Split horizon with poisoned reverse is not currently available. Additional route entries may be excluded from the periodic update with the ***ripSendHookAdd*( )** routine.

If a RIP session is terminated, the networking subsystem may not function correctly until RIP is restarted with a new call to ***ripLibInit*( )** unless routing information is provided by some other method.

CONFIGURATION INTERFACE

By default, a RIP session only uses the network interfaces created before it started. The ***ripIfSearch*( )** routine allows RIP to recognize any interfaces added to the system after that point. If the address or netmask of an existing interface is changed during a RIP session, the ***ripIfReset*( )** routine must be used to update the RIP configuration appropriately. The current RIP implementation also automatically performs the border gateway filtering required by the RFC specification. Those restrictions provide correct operation in a mixed environment of RIP-1 and RIP-2 routers. The ***ripFilterDisable*( )** routine will remove those limitations, and may produce more efficient routing for some topologies. That routine must not be used if any version 1 routers are present. The ***ripFilterEnable*( )** routine will restore the default behavior.

AUTHENTICATION INTERFACE

By default, authentication is disabled, but may be activated by an SNMP agent on an interface-specific basis. While authentication is disabled, any RIP-2 messages containing authentication entries are discarded. When enabled, all RIP-2 messages without authentication entries are automatically rejected. To fully support authentication, an authentication routine should be specified with the ***ripAuthHookAdd*( )** routine. The specified function will be called to screen every RIP-1 message and all unverified RIP-2 messages containing authentication entries. It may be removed with the ***ripAuthHookDelete*( )** routine. All RIP-1 and unverified RIP-2 messages will be discarded while authentication is enabled unless a hook is present.

OPTIONAL INTERFACE

The ***ripLeakHookAdd*( )** routine allows the use of an alternative routing protocol which uses RIP as a transport mechanism. The specified function can prevent the RIP session from creating any table entries from the received messages. The ***ripLeakHookDelete*( )** routine will restore the default operation.

DEBUGGING INTERFACE

As required by the RFC specification, the obsolete traceon and traceoff messages are not supported by this implementation. The ***ripRouteShow*( )** routine will display the contents of the internal RIP routing table. Routines such as ***mRouteShow*( )** to display the corresponding kernel routing table will also be available if **INCLUDE_NET_SHOW** is defined when the image is built. If additional information is required, the ***ripDebugLevelSet*( )** routine will enable predefined debugging messages which will be sent to the standard output.

INCLUDE FILES **ripLib.h**

SEE ALSO RFC 1058, RFC 1723

rlogLib

NAME

rlogLib – remote login library

ROUTINES

***rlogInit*( )** – initialize the remote login facility
***rlogind*( )** – the VxWorks remote login daemon
***rlogin*( )** – log in to a remote host

DESCRIPTION

This library provides a remote login facility for VxWorks 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 the configuration macro **INCLUDE_RLOGIN** is defined. The remote login daemon accepts remote login requests from another VxWorks or UNIX system, and causes the shell's input and output to be redirected to the remote user.

Internally, ***rlogind*( )** provides a tty-like interface to the remote user through the use of the VxWorks pseudo-terminal driver ptyDrv.

INCLUDE FILES

rlogLib.h

SEE ALSO

ptyDrv, **telnetLib**, UNIX BSD 4.3 manual entries for **rlogin**, **rlogind**, and **pty**

rngLib

NAME

rngLib – ring buffer subroutine library

ROUTINES

***rngCreate*( )** – create an empty ring buffer
***rngDelete*( )** – delete a ring buffer
***rngFlush*( )** – make a ring buffer empty
***rngBufGet*( )** – get characters from a ring buffer
***rngBufPut*( )** – put bytes into a ring buffer
***rngIsEmpty*( )** – test if a ring buffer is empty
***rngIsFull*( )** – test if a ring buffer is full (no more room)
***rngFreeBytes*( )** – determine the number of free bytes in a ring buffer
***rngNBytes*( )** – determine the number of bytes in a ring buffer

***rngPutAhead*( )** – put a byte ahead in a ring buffer without moving ring pointers
***rngMoveAhead*( )** – advance a ring pointer by *n* bytes

DESCRIPTION

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

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

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

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

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

INCLUDE FILES

rngLib.h

---

routeLib

NAME

routeLib – network route manipulation library

ROUTINES

***routeAdd*( )** – add a route
***routeNetAdd*( )** – add a route to a destination that is a network
***routeDelete*( )** – delete a route
***mRouteAdd*( )** – add multiple routes to the same destination
***mRouteEntryAdd*( )** – add a protocol-specific route to the routing table
***mRouteEntryDelete*( )** – delete route from the routing table
***mRouteDelete*( )** – delete a route from the routing table
***routeProtoPrioritySet*( )** – set the priority of routes added by the routing protocol

DESCRIPTION

This library contains the routines for inspecting the routing table, as well as routines for adding and deleting routes from that table. If you do not configure VxWorks to include a routing protocol, such as RIP or OSPF, you can use these routines to maintain the routing tables manually.

INCLUDE FILES **routeLib.h**

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

rpcLib

NAME **rpcLib** – Remote Procedure Call (RPC) support library

ROUTINES
***rpcInit*( )** – initialize the RPC package
***rpcTaskInit*( )** – initialize a task's access to the RPC package

DESCRIPTION This library supports Sun Microsystems' Remote Procedure Call (RPC) facility. RPC provides facilities for implementing distributed client/server-based architectures. The underlying communication mechanism can be completely hidden, permitting applications to be written without any reference to network sockets. The package is structured such that lower-level routines can optionally be accessed, allowing greater control of the communication protocols.

For more information and a tutorial on RPC, see Sun Microsystems' *Remote Procedure Call Programming Guide.* For an example of RPC usage, see **/target/unsupported/demo/sprites**.

The RPC facility is enabled when the configuration macro **INCLUDE_RPC** is defined.

VxWorks supports Network File System (NFS), which is built on top of RPC. If NFS is configured into the VxWorks system, RPC is automatically included as well.

IMPLEMENTATION A task must call ***rpcTaskInit*( )** before making any calls to other routines in the RPC library. This routine creates task-specific data structures required by RPC. These task-specific data structures are automatically deleted when the task exits.

Because each task has its own RPC context, RPC-related objects (such as SVCXPRTs and CLIENTs) cannot be shared among tasks; objects created by one task cannot be passed to another for use. Such additional objects must be explicitly deleted (for example, using task deletion hooks).

INCLUDE FILES **rpc.h**

SEE ALSO **nfsLib**, **nfsDrv**, Sun Microsystems' *Remote Procedure Call Programming Guide*

rt11FsLib

NAME

rt11FsLib – RT-11 media-compatible file system library

ROUTINES

***rt11FsDevInit*( )** – initialize the rt11Fs device descriptor
***rt11FsInit*( )** – prepare to use the rt11Fs library
***rt11FsMkfs*( )** – initialize a device and create an rt11Fs file system
***rt11FsDateSet*( )** – set the rt11Fs file system date
***rt11FsReadyChange*( )** – notify rt11Fs of a change in ready status
***rt11FsModeChange*( )** – modify the mode of an rt11Fs volume

DESCRIPTION

This library provides services for file-oriented device drivers which use the RT-11 file standard. This module takes care of all the necessary buffering, directory maintenance, and RT-11-specific details.

USING THIS LIBRARY

The various routines provided by the VxWorks RT-11 file system (rt11Fs) may be separated into three broad groups: general initialization, device initialization, and file system operation.

The ***rt11FsInit*( )** routine is the principal initialization function; it need only be called once, regardless of how many rt11Fs devices will be used.

Other rt11Fs routines are used for device initialization. For each rt11Fs device, either ***rt11FsDevInit*( )** or ***rt11FsMkfs*( )** must be called to install the device and define its configuration.

Several functions are provided to inform the file system of changes in the system environment. The ***rt11FsDateSet*( )** routine is used to set the date. The ***rt11FsModeChange*( )** routine is used to modify the readability or writability of a particular device. The ***rt11FsReadyChange*( )** routine is used to inform the file system that a disk may have been swapped, and that the next disk operation should first remount the disk.

INITIALIZING RT11FSLIB

Before any other routines in **rt11FsLib** can be used, ***rt11FsInit*( )** must be called to initialize this library. This call specifies the maximum number of rt11Fs files that can be open simultaneously and allocates memory for that many rt11Fs file descriptors. Attempts to open more files than the specified maximum will result in errors from ***open*( )** or ***creat*( )**.

This initialization is enabled when the configuration macro **INCLUDE_RT11FS** is defined.

DEFINING AN RT-11 DEVICE

To use this library for a particular device, the device structure must contain, as the very first item, a **BLK_DEV** structure. This must be initialized before calling ***rt11FsDevInit*( )**.

In the **BLK_DEV** structure, the driver includes the addresses of five routines which it must supply: one that reads one or more sectors, one that writes one or more sectors, one that performs I/O control on the device (using ***ioctl*( )**), one that checks the status of the device, and one that resets the device. This structure also specifies various physical aspects of the device (e.g., number of sectors, sectors per track, whether the media is removable). For more information about defining block devices, see the *VxWorks Programmer's Guide: I/O System.*

The device is associated with the rt11Fs file system by the ***rt11FsDevInit*( )** call. The arguments to ***rt11FsDevInit*( )** include the name to be used for the rt11Fs volume, a pointer to the **BLK_DEV** structure, whether the device uses RT-11 standard skew and interleave, and the maximum number of files that can be contained in the device directory.

Thereafter, when the file system receives a request from the I/O system, it simply calls the provided routines in the device driver to fulfill the request.

RTFMT

The RT-11 standard defines a peculiar software interleave and track-to-track skew as part of the format. The *rtFmt* parameter passed to ***rt11FsDevInit*( )** should be TRUE if this formatting is desired. This should be the case if strict RT-11 compatibility is desired, or if files must be transferred between the development and target machines using the VxWorks-supplied RT-11 tools. Software interleave and skew will automatically be dealt with by **rt11FsLib**.

When *rtFmt* has been passed as TRUE and the maximum number of files is specified **RT_FILES_FOR_2_BLOCK_SEG**, the driver does not need to do anything else to maintain RT-11 compatibility (except to add the track offset as described above).

Note that if the number of files specified is different than **RT_FILES_FOR_2_BLOCK_SEG** under either a VxWorks system or an RT-11 system, compatibility is lost because VxWorks allocates a contiguous directory, whereas RT-11 systems create chained directories.

MULTIPLE LOGICAL DEVICES AND RT-11 COMPATIBILITY

The sector number passed to the sector read and write routines is an absolute number, starting from sector 0 at the beginning of the device. If desired, the driver may add an offset from the beginning of the physical device before the start of the logical device. This would normally be done by keeping an offset parameter in the device-specific structure of the driver, and adding the proper number of sectors to the sector number passed to the read and write routines.

The RT-11 standard defines the disk to start on track 1. Track 0 is set aside for boot information. Therefore, in order to retain true compatibility with RT-11 systems, a one-track offset (i.e., the number of sectors in one track) needs to be added to the sector numbers passed to the sector read and write routines, and the device size needs to be declared as one track smaller than it actually is. This must be done by the driver using **rt11FsLib**; the library does not add such an offset automatically.

In the VxWorks RT-11 implementation, the directory is a fixed size, able to contain at least as many files as specified in the call to ***rt11FsDevInit*( )**. If the maximum number of files is specified to be **RT_FILES_FOR_2_BLOCK_SEG**, strict RT-11 compatibility is maintained, because this is the initial allocation in the RT-11 standard.

RT-11 FILE NAMES File names in the RT-11 file system use six characters, followed by a period (.), followed by an optional three-character extension.

DIRECTORY ENTRIES

An ***ioctl*( )** call with the **FIODIRENTRY** function returns information about a particular directory entry. A pointer to a **REQ_DIR_ENTRY** structure is passed as the parameter. The field **entryNum** in the **REQ_DIR_ENTRY** structure must be set to the desired entry number. The name of the file, its size (in bytes), and its creation date are returned in the structure. If the specified entry is empty (i.e., if it represents an unallocated section of the disk), the name will be an empty string, the size will be the size of the available disk section, and the date will be meaningless. Typically, entries are accessed sequentially, starting with **entryNum** = 0, until the terminating entry is reached, indicated by a return of **ERROR**.

DIRECTORIES IN MEMORY

A copy of the directory for each volume is kept in memory (in the **RT_VOL_DESC** structure). This speeds up directory accesses, but requires that **rt11FsLib**be notified when disks are changed (i.e., floppies are swapped). If the driver can find this out (by interrogating controller status or by receiving an interrupt), the driver simply calls ***rt11FsReadyChange*( )** when a disk is inserted or removed. The library **rt11FsLib** will automatically try to remount the device next time it needs it.

If the driver does not have access to the information that disk volumes have been changed, the *changeNoWarn* parameter should be set to TRUE when the device is defined using ***rt11FsDevInit*( )**. This will cause the disk to be automatically remounted before each ***open*( )**, ***creat*( )**, ***delete*( )**, and directory listing.

The routine ***rt11FsReadyChange*( )** can also be called by user tasks, by issuing an ***ioctl*( )** call with **FIODISKCHANGE** as the function code.

ACCESSING THE RAW DISK

As a special case in ***open*( )** and ***creat*( )** calls, **rt11FsLib** recognizes a NULL file name to indicate access to the entire "raw" disk, as opposed to a file on the disk. Access in raw mode is useful for a disk that has no file system. For example, to initialize a new file system on the disk, use an ***ioctl*( )** call with **FIODISKINIT**. To read the directory of a disk for which no file names are known, open the raw disk and use an ***ioctl*( )** call with the function **FIODIRENTRY**.

HINTS The RT-11 file system is much simpler than the more common UNIX or MS-DOS file systems. The advantage of RT-11 is its speed; file access is made in at most one seek because all files are contiguous. Some of the most common errors for users with a UNIX background are:

– Only a single create at a time may be active per device.

– File size is set by the first create and close sequence; use ***lseek*( )** to ensure a specific file size; there is no append function to expand a file.

– Files are strictly block oriented; unused portions of a block are filled with NULLs -- there is no end-of-file marker other than the last block.

IOCTL FUNCTIONS The rt11Fs file system supports the following ***ioctl*( )** functions. The functions listed are defined in the header **ioLib.h**. Unless stated otherwise, the file descriptor used for these functions can be any file descriptor open to a file or to the volume itself.

FIODISKFORMAT
Formats the entire disk with appropriate hardware track and sector marks. No file system is initialized on the disk by this request. Note that this is a driver-provided function:

```
fd = open ("DEV1:", O_WRONLY);
status = ioctl (fd, FIODISKFORMAT, 0);
```

FIODISKINIT
Initializes an rt11Fs file system on the disk volume. This routine does not format the disk; formatting must be done by the driver. The file descriptor should be obtained by opening the entire volume in raw mode:

```
fd = open ("DEV1:", O_WRONLY);
status = ioctl (fd, FIODISKINIT, 0);
```

FIODISKCHANGE
Announces a media change. It performs the same function as ***rt11FsReadyChange*( )**. This function may be called from interrupt level:

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

FIOGETNAME
Gets the file name of the file descriptor and copies it to the buffer *nameBuf*:

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

FIORENAME
Renames the file to the string *newname*:

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

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

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

FIOFLUSH
Flushes the file output buffer. It guarantees that any output that has been requested is actually written to the device.

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

FIOSEEK
Sets the current byte offset in the file to the position specified by *newOffset*:

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

FIOWHERE
Returns the current byte position in the file. This is the byte offset of the next byte to be read or written. It takes no additional argument:

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

FIOSQUEEZE
Coalesces fragmented free space on an rt11Fs volume:

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

FIODIRENTRY
Copies information about the specified directory entries to a **REQ_DIR_ENTRY** structure that is defined in **ioLib.h**. The argument *req*is a pointer to a **REQ_DIR_ENTRY** structure. On entry, the structure contains the number of the directory entry for which information is requested. On return, the structure contains the information on the requested entry. For example, after the following:

```
REQ_DIR_ENTRY req;
req.entryNum = 0;
status = ioctl (fd, FIODIRENTRY, &req);
```

The request structure contains the name, size, and creation date of the file in the first entry (0) of the directory.

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

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

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

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

Any other ***ioctl*( )** function codes are passed to the block device driver for handling.

INCLUDE FILES **rt11FsLib.h**

SEE ALSO **ioLib**, **iosLib**, **ramDrv**, *VxWorks Programmer's Guide: I/O System, Local File Systems*

sa1100Sio

NAME **sa1100Sio** – Digital Semiconductor SA-1100 UART tty driver

ROUTINES ***sa1100DevInit*( )** – initialise an SA1100 channel
***sa1100Int*( )** – handle an interrupt

DESCRIPTION This is the device driver for the Digital Semiconductor SA-1100 UARTs. This chip contains 5 serial ports, but only ports 1 and 3 are usable as UARTs, the others support Universal Serial Bus (USB), SDLC, IrDA Infrared Communications Port (ICP) and Multimedia Communications Port (MCP)/Synchronous Serial Port (SSP).

The UARTs are identical in design. They contain a universal asynchronous receiver/transmitter, and a baud-rate generator, The UARTs contain an 8-entry, 8-bit FIFO to buffer outgoing data and a 12-entry 11-bit FIFO to buffer incoming data. If a framing, overrun or parity error occurs during reception, the appropriate error bits are stored in the receive FIFO along with the received data. The only mode of operation supported is with the FIFOs enabled.

The UART design does not support modem control input or output signals e.g. DTR, RI, RTS, DCD, CTS and DSR.

An interrupt is generated when a framing, parity or receiver overrun error is present within the bottom four entries of the receive FIFO, when the transmit FIFO is half-empty or receive FIFO is one- to two-thirds full, when a begin and end of break is detected on the receiver, and when the receive FIFO is partially full and the receiver is idle for three or more frame periods.

Only asynchronous serial operation is supported by the UARTs which supports 7 or 8 bit word lengths with or without parity and with one or two stop bits. The only serial word format supported by the driver is 8 data bits, 1 stop bit, no parity, The default baud rate is determined by the BSP by filling in the **SA1100_CHAN** structure before calling ***sa1100DevInit*( )**.

The UART supports baud rates from 56.24 to 230.4 kbps.

DATA STRUCTURES An **SA1100_CHAN** data structure is used to describe each channel, this structure is described in **h/drv/sio/sa1100Sio.h**.

CALLBACKS Servicing a "transmitter ready" interrupt involves making a callback to a higher level library in order to get a character to transmit. By default, this driver installs dummy callback routines which do nothing. A higher layer library that wants to use this driver (e.g. **ttyDrv**) will install its own callback routine using the **SIO_INSTALL_CALLBACK** ioctl command. Likewise, a receiver interrupt handler makes a callback to pass the character to the higher layer library.

MODES

This driver supports both polled and interrupt modes.

USAGE

The driver is typically only called by the BSP. The directly callable routines in this modules are ***sa1100DevInit( )***, and ***sa1100Int( )***.

The BSP's ***sysHwInit( )*** routine typically calls ***sysSerialHwInit( )***, which initialises the hardware-specific fields in the **SA1100_CHAN** structure (e.g. register I/O addresses etc) before calling ***sa1100DevInit( )*** which resets the device and installs the driver function pointers. After this the UART will be enabled and ready to generate interrupts, but those interrupts will be disabled in the interrupt controller.

The following example shows the first parts of the initialization:

```
#include "drv/sio/sa1100Sio.h"
LOCAL SA1100_CHAN sa1100Chan[N_SA1100_UART_CHANS];
void sysSerialHwInit (void)
    {
    int i;
    for (i = 0; i < N_SA1100_UART_CHANNELS; i++)
        {
        sa1100Chan[i].regs = devParas[i].baseAdrs;
        sa1100Chan[i].baudRate = CONSOLE_BAUD_RATE;
        sa1100Chan[i].xtal = UART_XTAL_FREQ;
        sa1100Chan[i].level = devParas[i].intLevel;
        /* set up GPIO pins and UART pin reassignment */
        ...
        /*
         * Initialise driver functions, getTxChar, putRcvChar
         * and channelMode and initialise UART
         */
        sa1100DevInit(&sa1100Chan[i]);
        }
    }
```

The BSP's ***sysHwInit2( )*** routine typically calls ***sysSerialHwInit2( )***, which connects the chip's interrupts via ***intConnect( )*** and enables those interrupts, as in the following:

```
void sysSerialHwInit2 (void)
    {
    int i;
    for (i = 0; i < N_SA1100_UART_CHANNELS; i++)
        {
        /* connect and enable interrupts */
        (void)intConnect (INUM_TO_IVEC(devParas[i].vector),
                          sa1100Int, (int) &sa1100Chan[i]);
        intEnable (devParas[i].intLevel);
        }
    }
```

BSP

By convention all the BSP-specific serial initialisation is performed in a file called **sysSerial.c**, which is #include'ed by **sysLib.c**. **sysSerial.c** implements at least four functions, ***sysSerialHwInit*( )**, ***sysSerialHwInit2*( )**, ***sysSerialChanGet*( )**, and ***sysSerialReset*( )**. The first two have been described above, the others work as follows:

sysSerialChanGet is called by usrRoot to get the serial channel descriptor associated with a serial channel number. The routine takes a single parameter which is a channel number ranging between zero and **NUM_TTY**. It returns a pointer to the corresponding channel descriptor, **SIO_CHAN** *, which is just the address of the **SA1100_CHAN** structure.

sysSerialReset is called from ***sysToMonitor*( )** and should reset the serial devices to an inactive state (prevent them from generating any interrupts).

INCLUDE FILES

drv/sio/sa1100Sio.h sioLib.h

SEE ALSO

Digital StrongARM SA-1100 Portable Communications Microcontroller, Data Sheet, *Digital Semiconductor StrongARM SA-1100 Microprocessor Evaluation Platform,* User's Guide

---

saIoLib

NAME

saIoLib – default transport routines for SNMP subagent

ROUTINES

***snmpSaInit*( )** – initialize the subagent
***saIoWrite*( )** – send a packet to the master agent's message queue
***saIpcFree*( )** – free the specified IPC mechanism
***saMsgBuild*( )** – build and encode a message and send it to the master agent
***hdrBlkBuild*( )** – create the header block and the demuxer information
***envoy_now*( )** – return the number of clock ticks elapsed since the timer was set
***envoy_call_timer*( )** – execute the specified function when the timer expires

DESCRIPTION

This library implements the subagent side of the IPC mechanism used to pass messages between the SNMP master agent and its subagents. In the shipped version of this library, the IPC mechanism is a message queue. However, it is a relatively simple matter to replace the message queue with a socket if you cannot use message queues.

To set up the IPC mechanism and spawn a task to monitor it, call ***snmpSaInit*( )**. To send a message to the master agent, you can call ***saIoWrite*( )**. However, you will likely never call this function directly. Instead, you will call ***hdrBlkBuild*( )**. Internally, ***hdrBlkBuild*( )** calls ***saMsgBuild*( )**, which calls ***snmpSubEncode*( )** and finally ***saIoWrite*( )**.

The first message you will transmit using ***hdrBlkBuild*( )** will be a registration message that registers objects and instances as a group in the master agent's MIB tree. If successful, the response to this message will contain a group ID. Make sure that you store this ID so that you can later remove the group from the MIB tree when you want to deregister the subagent. You also need this ID if you want to register instances of the object just registered.

Exactly how and when you register a subagent is up to you, but keep in mind that you can do so only after the master agent is up an running.

SEE ALSO **saIoLib**

schedPxLib

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

ROUTINES
***sched_setparam*( )** – set a task's priority (POSIX)
***sched_getparam*( )** – get the scheduling parameters for a specified task (POSIX)
***sched_setscheduler*( )** – set scheduling policy and scheduling parameters (POSIX)
***sched_getscheduler*( )** – get the current scheduling policy (POSIX)
***sched_yield*( )** – relinquish the CPU (POSIX)
***sched_get_priority_max*( )** – get the maximum priority (POSIX)
***sched_get_priority_min*( )** – get the minimum priority (POSIX)
***sched_rr_get_interval*( )** – get the current time slice (POSIX)

DESCRIPTION
This library provides POSIX-compliance scheduling routines. The routines in this library allow the user to get and set priorities and scheduling schemes, get maximum and minimum priority values, and get the time slice if round-robin scheduling is enabled.

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

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

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

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

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

INCLUDE FILES **sched.h**

SEE ALSO POSIX 1003.1b document, **taskLib**

scsi1Lib

NAME **scsi1Lib** – Small Computer System Interface (SCSI) library (SCSI-1)

ROUTINES 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 devices, additional higher-level software is necessary to map user commands to SCSI transactions.

CONFIGURING SCSI CONTROLLERS

The routines to create and initialize a specific SCSI controller are particular to the controller and normally are found in its library module. The normal calling sequence is:

```
xxCtrlCreate (...); /* parameters are controller specific */
xxCtrlInit (...);   /* parameters are controller specific */
```

The conceptual difference between the two routines is that ***xxCtrlCreate*( )** calloc's memory for the **xx_SCSI_CTRL** data structure and initializes information that is never expected to change (for example, clock rate). The remaining fields in the **xx_SCSI_CTRL** structure are initialized by ***xxCtrlInit*( )** and any necessary registers are written on the SCSI controller to effect the desired initialization. This routine can be called multiple times, although this is rarely required. For example, the bus ID of the SCSI controller can be changed without rebooting the VxWorks system.

CONFIGURING PHYSICAL SCSI DEVICES

Before a device can be used, it must be "created," that is, declared. This is done with ***scsiPhysDevCreate*( )** and can only be done after a **SCSI_CTRL** structure exists and has been properly initialized.

```
SCSI_PHYS_DEV *scsiPhysDevCreate
    (
    SCSI_CTRL * pScsiCtrl,/* ptr to SCSI controller info */
    int devBusId,        /* device's SCSI bus ID */
    int devLUN,          /* device's logical unit number */
    int reqSenseLength,  /* length of REQUEST SENSE data dev returns */
    int devType,         /* type of SCSI device */
    BOOL removable,       /* whether medium is removable */
    int numBlocks,       /* number of blocks on device */
    int blockSize        /* size of a block in bytes */
    )
```

Several of these parameters can be left unspecified, as follows:

reqSenseLength
If 0, issue a **REQUEST_SENSE** to determine a request sense length.

devType
If -1, issue an INQUIRY to determine the device type.

numBlocks, *blockSize*
If 0, issue a **READ_CAPACITY** to determine the number of blocks.

The above values are recommended, unless the device does not support the required commands, or other non-standard conditions prevail.

LOGICAL PARTITIONS ON BLOCK DEVICES

It is possible to have more than one logical partition on a SCSI block device. This capability is currently not supported for removable media devices. A partition is an array of contiguously addressed blocks with a specified starting block address and a specified number of blocks. The ***scsiBlkDevCreate*( )** routine is called once for each block device partition. Under normal usage, logical partitions should not overlap.

```
SCSI_BLK_DEV *scsiBlkDevCreate
    (
    SCSI_PHYS_DEV *  pScsiPhysDev,  /* ptr to SCSI physical device info */
    int              numBlocks,     /* number of blocks in block device */
    int              blockOffset    /* address of first block in volume */
    )
```

Note that if *numBlocks* is 0, the rest of the device is used.

ATTACHING FILE SYSTEMS TO LOGICAL PARTITIONS

Files cannot be read or written to a disk partition until a file system (such as dosFs or rt11Fs) has been initialized on the partition. For more information, see the documentation in **dosFsLib** or **rt11FsLib**.

TRANSMITTING ARBITRARY COMMANDS TO SCSI DEVICES

The **scsi1Lib** library provides routines that implement many common SCSI commands. Still, there are situations that require commands that are not supported by **scsi1Lib** (for example, writing software to control non-direct access devices). Arbitrary commands are handled with the **FIOSCSICOMMAND** option to ***scsiIoctl*( )**. The *arg* parameter for **FIOSCSICOMMAND** is a pointer to a valid **SCSI_TRANSACTION** structure. Typically, a call to ***scsiIoctl*( )** is written as a subroutine of the form:

```
STATUS myScsiCommand
    (
    SCSI_PHYS_DEV *  pScsiPhysDev,  /* ptr to SCSI physical device     */
    char *           buffer,        /* ptr to data buffer              */
    int              bufLength,     /* length of buffer in bytes       */
    int              someParam      /* param. specifiable in cmd block */
```

```
)
{
SCSI_COMMAND myScsiCmdBlock;        /* SCSI command byte array */
SCSI_TRANSACTION myScsiXaction;     /* info on a SCSI transaction */
/* fill in fields of SCSI_COMMAND structure */
myScsiCmdBlock [0] = MY_COMMAND_OPCODE;     /* the required opcode */
.
myScsiCmdBlock [X] = (UINT8) someParam;     /* for example */
.
myScsiCmdBlock [N-1] = MY_CONTROL_BYTE;     /* typically == 0 */
/* fill in fields of SCSI_TRANSACTION structure */
myScsiXaction.cmdAddress    = myScsiCmdBlock;
myScsiXaction.cmdLength     = <# of valid bytes in myScsiCmdBlock>;
myScsiXaction.dataAddress   = (UINT8 *) buffer;
myScsiXaction.dataDirection = <O_RDONLY (0) or O_WRONLY (1)>;
myScsiXaction.dataLength    = bufLength;
myScsiXaction.cmdTimeout    = timeout in usec;
/* if dataDirection is O_RDONLY, and the length of the input data is
 * variable, the following parameter specifies the byte # (min == 0)
 * of the input data which will specify the additional number of
 * bytes available
 */
myScsiXaction.addLengthByte = X;
if (scsiIoctl (pScsiPhysDev, FIOSCSICOMMAND, &myScsiXaction) == OK)
    return (OK);
else
    /* optionally perform retry or other action based on value of
     *  myScsiXaction.statusByte
     */
    return (ERROR);
}
```

INCLUDE FILES **scsiLib.h**, **scsi1Lib.h**

SEE ALSO **dosFsLib**, **rt11FsLib**, *American National Standards for Information Systems – Small Computer System Interface (SCSI)*, ANSI X3.131-1986, *VxWorks Programmer's Guide: I/O System, Local File Systems*

scsi2Lib

NAME

scsi2Lib – Small Computer System Interface (SCSI) library (SCSI-2)

ROUTINES

***scsi2IfInit*( )** – initialize the SCSI-2 interface to **scsiLib**
***scsiTargetOptionsSet*( )** – set options for one or all SCSI targets
***scsiTargetOptionsGet*( )** – get options for one or all SCSI targets
***scsiPhysDevShow*( )** – show status information for a physical device
***scsiCacheSynchronize*( )** – synchronize the caches for data coherency
***scsiIdentMsgBuild*( )** – build an identification message
***scsiIdentMsgParse*( )** – parse an identification message
***scsiMsgOutComplete*( )** – perform post-processing after a SCSI message is sent
***scsiMsgOutReject*( )** – perform post-processing when an outgoing message is rejected
***scsiMsgInComplete*( )** – handle a complete SCSI message received from the target
***scsiSyncXferNegotiate*( )** – initiate or continue negotiating transfer parameters
***scsiWideXferNegotiate*( )** – initiate or continue negotiating wide parameters
***scsiThreadInit*( )** – perform generic SCSI thread initialization
***scsiCacheSnoopEnable*( )** – inform SCSI that hardware snooping of caches is enabled
***scsiCacheSnoopDisable*( )** – inform SCSI that hardware snooping of caches is disabled

DESCRIPTION

This library implements the Small Computer System Interface (SCSI) protocol in a controller-independent manner. It implements only the SCSI initiator function as defined in the SCSI-2 ANSI specification. This library does not support a VxWorks target acting as a SCSI target.

The implementation is transaction based. A transaction is defined as the selection of a SCSI device by the initiator, the issuance of a SCSI command, and the sequence of data, status, and message phases necessary to perform the command. A transaction normally completes with a "Command Complete" message from the target, followed by disconnection from the SCSI bus. If the status from the target is "Check Condition," the transaction continues; the initiator issues a "Request Sense" command to gain more information on the exception condition reported.

Many of the subroutines in **scsi2Lib** facilitate the transaction of frequently used SCSI commands. Individual command fields are passed as arguments from which SCSI Command Descriptor Blocks are constructed, and fields of a **SCSI_TRANSACTION** structure are filled in appropriately. This structure, along with the **SCSI_PHYS_DEV** structure associated with the target SCSI device, is passed to the routine whose address is indicated by the **scsiTransact** field of the **SCSI_CTRL** structure associated with the relevant SCSI controller. The above mentioned structures are defined in **scsi2Lib.h**.

The function variable **scsiTransact** is set by the individual SCSI controller driver. For off-board SCSI controllers, this routine rearranges the fields of the **SCSI_TRANSACTION** structure into the appropriate structure for the specified hardware, which then carries out the transaction through firmware control. Drivers for an on-board SCSI-controller chip can use the ***scsiTransact*( )** routine in **scsiLib** (which invokes the ***scsi2Transact*( )** routine in **scsi2Lib**), as long as they provide the other functions specified in the **SCSI_CTRL** structure.

SCSI TRANSACTION TIMEOUT

Associated with each transaction is a time limit (specified in microseconds, but measured with the resolution of the system clock). If the transaction has not completed within this time limit, the SCSI library aborts it; the called routine fails with a corresponding error code. The timeout period includes time spent waiting for the target device to become free to accept the command.

The semantics of the timeout should guarantee that the caller waits no longer than the transaction timeout period, but in practice this may depend on the state of the SCSI bus and the connected target device when the timeout occurs. If the target behaves correctly according to the SCSI specification, proper timeout behavior results. However, in certain unusual cases--for example, when the target does not respond to an asserted ATN signal--the caller may remain blocked for longer than the timeout period.

If the transaction timeout causes problems in your system, you can set the value of either or both the global variables "scsi{Min,Max}Timeout". These specify (in microseconds) the global minimum and maximum timeout periods, which override (clip) the value specified for a transaction. They may be changed at any time and affect all transactions issued after the new values are set. The range of both these variable is 0 to 0xffffffff (zero to about 4295 seconds).

SCSI TRANSACTION PRIORITY

Each transaction also has an associated priority used by the SCSI library when selecting the next command to issue when the SCSI system is idle. It chooses the highest priority transaction that can be dispatched on an available physical device. If there are several equal-priority transactions available, the SCSI library uses a simple round-robin scheme to avoid favoring the same physical device.

Priorities range from 0 (highest) to 255 (lowest), which is the same as task priorities. The priority **SCSI_THREAD_TASK_PRIORITY** can be used to give the transaction the same priority as the calling task (this is the method used internally by this SCSI-2 library).

SUPPORTED SCSI DEVICES

This library requires peripherals that conform to the SCSI-2 ANSI standard; in particular, the **INQUIRY**, **REQUEST SENSE**, and **TEST UNIT READY** commands must be supported as specified by this standard. In general, the SCSI library is self-configuring to work with any device that meets these requirements.

Peripherals that support identification and the SCSI message protocol are strongly recommended as these provide maximum performance.

In theory, all classes of SCSI devices are supported. **scsiLib** provides the capability to transact any SCSI command on any SCSI device through the **FIOSCSICOMMAND** function of the ***scsiIoctl*( )** routine (which invokes the ***scsi2Ioctl*( )** routine in **scsi2Lib**).

Only direct-access devices (disks) are supported by file systems like dosFs, rt11Fs and rawFs. These file systems employ routines in **scsiDirectLib** (most of which are described in **scsiLib** but defined in **scsiDirectLib**). In the case of sequential-access devices (tapes), higher-level tape file systems, like tapeFs, make use of **scsiSeqLib**. For other types of devices, additional, higher-level software is necessary to map user-level commands to SCSI transactions.

DISCONNECT/RECONNECT SUPPORT The target device can be disconnected from the SCSI bus while it carries out a SCSI command; in this way, commands to multiple SCSI devices can be overlapped to improve overall SCSI throughput. There are no restrictions on the number of pending, disconnected commands or the order in which they are resumed. The SCSI library serializes access to the device according to the capabilities and status of the device (see the following section).

Use of the disconnect/reconnect mechanism is invisible to users of the SCSI library. It can be enabled and disabled separately for each target device (see ***scsiTargetOptionsSet( )***). Note that support for disconnect/reconnect depends on the capabilities of the controller and its driver (see below).

TAGGED COMMAND QUEUEING SUPPORT

If the target device conforms to the ANSI SCSI-2 standard and indicates (using the INQUIRY command) that it supports command queuing, the SCSI library allows new commands to be started on the device whenever the SCSI bus is idle. That is, it executes multiple commands concurrently on the target device. By default, commands are tagged with a SIMPLE QUEUE TAG message. Up to 256 commands can be executing concurrently.

The SCSI library correctly handles contingent allegiance conditions that arise while a device is executing tagged commands. (A contingent allegiance condition exists when a target device is maintaining sense data that the initiator should use to correctly recover from an error condition.) It issues an untagged REQUEST SENSE command, and stops issuing tagged commands until the sense recovery command has completed.

For devices that do not support command queuing, the SCSI library only issues a new command when the previous one has completed. These devices can only execute a single command at once.

Use of tagged command queuing is normally invisible to users of the SCSI library. If necessary, the default tag type and maximum number of tags may be changed on a per-target basis, using ***scsiTargetOptionsSet( )***.

SYNCHRONOUS TRANSFER PROTOCOL SUPPORT

If the SCSI controller hardware supports the synchronous transfer protocol, **scsiLib** negotiates with the target device to determine whether to use synchronous or asynchronous transfers. Either VxWorks or the target device may start a round of negotiation. Depending on the controller hardware, synchronous transfer rates up to the maximum allowed by the SCSI-2 standard (10 Mtransfers/second) can be used.

Again, this is normally invisible to users of the SCSI library, but synchronous transfer parameters may be set or disabled on a per-target basis by using ***scsiTargetOptionsSet*( )**.

WIDE DATA TRANSFER SUPPORT

If the SCSI controller supports the wide data transfer protocol, **scsiLib**negotiates wide data transfer parameters with the target device, if that device also supports wide transfers. Either VxWorks or the target device may start a round of negotiation. Wide data transfer parameters are negotiated prior to the synchronous data transfer parameters, as specified by the SCSI-2 ANSI specification. In conjunction with synchronous transfer, up to a maximum of 20MB/sec. can be attained.

Wide data transfer negotiation is invisible to users of this library, but it is possible to enable or disable wide data transfers and the parameters on a per-target basis by using ***scsiTargetOptionsSet*( )**.

SCSI BUS RESET

The SCSI library implements the ANSI "hard reset" option. Any transactions in progress when a SCSI bus reset is detected fail with an error code indicating termination due to bus reset. Any transactions waiting to start executing are then started normally.

CONFIGURING SCSI CONTROLLERS

The routines to create and initialize a specific SCSI controller are particular to the controller and normally are found in its library module. The normal calling sequence is:

```
xxCtrlCreate (...); /* parameters are controller specific */
xxCtrlInit (...);   /* parameters are controller specific */
```

The conceptual difference between the two routines is that ***xxCtrlCreate*( )** calloc's memory for the xx_SCSI_CTRL data structure and initializes information that is never expected to change (for example, clock rate). The remaining fields in the xx_SCSI_CTRL structure are initialized by ***xxCtrlInit*( )** and any necessary registers are written on the SCSI controller to effect the desired initialization. This routine can be called multiple times, although this is rarely required. For example, the bus ID of the SCSI controller can be changed without rebooting the VxWorks system.

CONFIGURING PHYSICAL SCSI DEVICES

Before a device can be used, it must be "created," that is, declared. This is done with ***scsiPhysDevCreate*( )** and can only be done after a **SCSI_CTRL** structure exists and has been properly initialized.

```
SCSI_PHYS_DEV *scsiPhysDevCreate
    (
    SCSI_CTRL * pScsiCtrl,/* ptr to SCSI controller info */
    int devBusId,         /* device's SCSI bus ID */
    int devLUN,           /* device's logical unit number */
    int reqSenseLength,   /* length of REQUEST SENSE data dev returns */
    int devType,          /* type of SCSI device */
    BOOL removable,        /* whether medium is removable */
```

```
    int numBlocks,       /* number of blocks on device */
    int blockSize        /* size of a block in bytes */
    )
```

Several of these parameters can be left unspecified, as follows:

reqSenseLength
: If 0, issue a **REQUEST_SENSE** to determine a request sense length.

devType
: This parameter is ignored: an INQUIRY command is used to ascertain the device type. A value of NONE (-1) is the recommended placeholder.

numBlocks, *blockSize*
: If 0, issue a **READ_CAPACITY** to determine the number of blocks.

The above values are recommended, unless the device does not support the required commands, or other non-standard conditions prevail.

LOGICAL PARTITIONS ON DIRECT-ACCESS BLOCK DEVICES

It is possible to have more than one logical partition on a SCSI block device. This capability is currently not supported for removable media devices. A partition is an array of contiguously addressed blocks with a specified starting block address and specified number of blocks. The ***scsiBlkDevCreate*( )** routine is called once for each block device partition. Under normal usage, logical partitions should not overlap.

```
SCSI_BLK_DEV *scsiBlkDevCreate
    (
    SCSI_PHYS_DEV *  pScsiPhysDev,  /* ptr to SCSI physical device info */
    int              numBlocks,     /* number of blocks in block device */
    int              blockOffset    /* address of first block in volume */
    )
```

Note that if *numBlocks* is 0, the rest of the device is used.

ATTACHING DISK FILE SYSTEMS TO LOGICAL PARTITIONS

Files cannot be read or written to a disk partition until a file system (for example, dosFs, rt11Fs, or rawFs) has been initialized on the partition. For more information, see the relevant documentation in **dosFsLib**, **rt11FsLib**, or **rawFsLib**.

USING A SEQUENTIAL-ACCESS BLOCK DEVICE

The entire volume (tape) on a sequential-access block device is treated as a single raw file. This raw file is made available to higher-level layers like tapeFs by the ***scsiSeqDevCreate*( )** routine, described in **scsiSeqLib**. The ***scsiSeqDevCreate*( )** routine is called once for a given SCSI physical device.

```
SEQ_DEV *scsiSeqDevCreate
    (
    SCSI_PHYS_DEV *pScsiPhysDev  /* ptr to SCSI physical device info */
    )
```

TRANSMITTING ARBITRARY COMMANDS TO SCSI DEVICES

The **scsi2Lib**, **scsiCommonLib**, **scsiDirectLib**, and **scsiSeqLib** libraries collectively provide routines that implement all mandatory SCSI-2 direct-access and sequential-access commands. Still, there are situations that require commands 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)

ROUTINES No Callable Routines

DESCRIPTION This library contains commands common to all SCSI devices. The content of this library is separated from the other SCSI libraries in order to create an additional layer for better support of all SCSI devices.

Commands in this library include:

Command	Op Code
INQUIRY	(0x12)
REQUEST SENSE	(0x03)
TEST UNIT READY	(0x00)

INCLUDE FILES **scsiLib.h**, **scsi2Lib.h**

SEE ALSO **dosFsLib**, **rt11FsLib**, **rawFsLib**, **tapeFsLib**, **scsi2Lib**, *VxWorks Programmer's Guide: I/O System, Local File Systems*

scsiCtrlLib

NAME **scsiCtrlLib** – SCSI thread-level controller library (SCSI-2)

ROUTINES No Callable Routines

DESCRIPTION The purpose of the SCSI controller library is to support basic SCSI controller drivers that rely on a higher level of software in order to manage SCSI transactions. More advanced

SCSI I/O processors do not require this protocol engine since software support for SCSI transactions is provided at the SCSI I/O processor level.

This library provides all the high-level routines that manage the state of the SCSI threads and guide the SCSI I/O transaction through its various stages:

– selecting a SCSI peripheral device;

– sending the identify message in order to establish the ITL nexus;

– cycling through information transfer, message and data, and status phases;

– handling bus-initiated reselects.

The various stages of the SCSI I/O transaction are reported to the SCSI manager as SCSI events. Event selection and management is handled by routines in this library.

INCLUDE FILES **scsiLib.h**, **scsi2Lib.h**

SEE ALSO **scsiLib**, **scsi2Lib**, **scsiCommonLib**, **scsiDirectLib**, **scsiSeqLib**, **scsiMgrLib**, *American National Standard for Information Systems – Small Computer System Interface (SCSI-2)*, ANSI X3T9, *VxWorks Programmer's Guide: I/O System, Local File Systems*

scsiDirectLib

NAME **scsiDirectLib** – SCSI library for direct access devices (SCSI-2)

ROUTINES

***scsiStartStopUnit*( )** – issue a **START_STOP_UNIT** command to a SCSI device
***scsiReserve*( )** – issue a RESERVE command to a SCSI device
***scsiRelease*( )** – issue a RELEASE command to a SCSI device

DESCRIPTION This library contains commands common to all direct-access SCSI devices. These routines are separated from **scsi2Lib** in order to create an additional layer for better support of all SCSI direct-access devices.

Commands in this library include:

Command	Op Code
FORMAT UNIT	(0x04)
READ (6)	(0x08)
READ (10)	(0x28)
READ CAPACITY	(0x25)
RELEASE	(0x17)
RESERVE	(0x16)
MODE SELECT (6)	(0x15)

Command	Op Code
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*

scsiLib

NAME **scsiLib** – Small Computer System Interface (SCSI) library

ROUTINES
***scsiPhysDevDelete*( )** – delete a SCSI physical-device structure
***scsiPhysDevCreate*( )** – create a SCSI physical device structure
***scsiPhysDevIdGet*( )** – return a pointer to a **SCSI_PHYS_DEV** structure
***scsiAutoConfig*( )** – configure all devices connected to a SCSI controller
***scsiShow*( )** – list the physical devices attached to a SCSI controller
***scsiBlkDevCreate*( )** – define a logical partition on a SCSI block device
***scsiBlkDevInit*( )** – initialize fields in a SCSI logical partition
***scsiBlkDevShow*( )** – show the **BLK_DEV** structures on a specified physical device
***scsiBusReset*( )** – pulse the reset signal on the SCSI bus
***scsiIoctl*( )** – perform a device-specific I/O control function
***scsiFormatUnit*( )** – issue a **FORMAT_UNIT** command to a SCSI device
***scsiModeSelect*( )** – issue a **MODE_SELECT** command to a SCSI device
***scsiModeSense*( )** – issue a **MODE_SENSE** command to a SCSI device
***scsiReadCapacity*( )** – issue a **READ_CAPACITY** command to a SCSI device
***scsiRdSecs*( )** – read sector(s) from a SCSI block device
***scsiWrtSecs*( )** – write sector(s) to a SCSI block device
***scsiTestUnitRdy*( )** – issue a **TEST_UNIT_READY** command to a SCSI device
***scsiInquiry*( )** – issue an INQUIRY command to a SCSI device
***scsiReqSense*( )** – issue a **REQUEST_SENSE** command to a SCSI device and read results

DESCRIPTION The purpose of this library is to switch SCSI function calls (the common SCSI-1 and SCSI-2 calls listed above) to either **scsi1Lib** or **scsi2Lib**, depending upon the SCSI configuration in the Board Support Package (BSP). The normal usage is to configure SCSI-2. However, SCSI-1 is configured when device incompatibilities exist. VxWorks can be configured with either SCSI-1 or SCSI-2, but not both SCSI-1 and SCSI-2 simultaneously.

For more information about SCSI-1 functionality, refer to **scsi1Lib**. For more information about SCSI-2, refer to **scsi2Lib**.

INCLUDE FILES **scsiLib.h**, **scsi1Lib.h**, **scsi2Lib.h**

SEE ALSO **dosFsLib**, **rt11FsLib**, **rawFsLib**, **scsi1Lib**, **scsi2Lib**, *VxWorks Programmer's Guide: I/O System, Local File Systems*

scsiMgrLib

NAME **scsiMgrLib** – SCSI manager library (SCSI-2)

ROUTINES
***scsiMgrEventNotify*( )** – notify the SCSI manager of a SCSI (controller) event
***scsiMgrBusReset*( )** – handle a controller-bus reset event
***scsiMgrCtrlEvent*( )** – send an event to the SCSI controller state machine
***scsiMgrThreadEvent*( )** – send an event to the thread state machine
***scsiMgrShow*( )** – show status information for the SCSI manager

DESCRIPTION This SCSI-2 library implements the SCSI manager. The 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)

ROUTINES

***scsiSeqDevCreate*( )** – create a SCSI sequential device
***scsiErase*( )** – issue an ERASE command to a SCSI device
***scsiTapeModeSelect*( )** – issue a **MODE_SELECT** command to a SCSI tape device
***scsiTapeModeSense*( )** – issue a **MODE_SENSE** command to a SCSI tape device
***scsiSeqReadBlockLimits*( )** – issue a **READ_BLOCK_LIMITS** command to a SCSI device
***scsiRdTape*( )** – read bytes or blocks from a SCSI tape device
***scsiWrtTape*( )** – write data to a SCSI tape device
***scsiRewind*( )** – issue a REWIND command to a SCSI device
***scsiReserveUnit*( )** – issue a RESERVE UNIT command to a SCSI device
***scsiReleaseUnit*( )** – issue a RELEASE UNIT command to a SCSI device
***scsiLoadUnit*( )** – issue a LOAD/UNLOAD command to a SCSI device
***scsiWrtFileMarks*( )** – write file marks to a SCSI sequential device
***scsiSpace*( )** – move the tape on a specified physical SCSI device
***scsiSeqStatusCheck*( )** – detect a change in media
***scsiSeqIoctl*( )** – perform an I/O control function for sequential access devices

DESCRIPTION This library contains commands common to all sequential-access SCSI devices. Sequential-access SCSI devices are usually SCSI tape devices. These routines are separated from **scsi2Lib** in order to create an additional layer for better support of all SCSI sequential devices. SCSI commands in this library include:

Command	Op Code
ERASE	(0x19)

Command	**Op Code**
MODE SELECT (6)	(0x15)
MODE_SENSE (6)	(0x1a)
READ (6)	(0x08)
READ BLOCK LIMITS	(0x05)
RELEASE UNIT	(0x17)
RESERVE UNIT	(0x16)
REWIND	(0x01)
SPACE	(0x11)
WRITE (6)	(0x0a)
WRITE FILEMARKS	(0x10)
LOAD/UNLOAD	(0x1b)

The SCSI routines implemented here operate mostly on a **SCSI_SEQ_DEV** structure. This structure acts as an interface between this library and a higher-level layer. The **SEQ_DEV** structure is analogous to the **BLK_DEV** structure for block devices.

The ***scsiSeqDevCreate*( )** routine creates a **SCSI_SEQ_DEV** structure whose first element is a **SEQ_DEV**, operated upon by higher layers. This routine publishes all functions to be invoked by higher layers and maintains some state information (for example, block size) for tracking SCSI-sequential-device information.

INCLUDE FILES **scsiLib.h**, **scsi2Lib.h**

SEE ALSO **tapeFsLib**, **scsi2Lib**, *VxWorks Programmer's Guide: I/O System*, *Local File Systems*

selectLib

NAME **selectLib** – UNIX BSD 4.3 select library

ROUTINES

***selectInit*( )** – initialize the select facility
***select*( )** – pend on a set of file descriptors
***selWakeup*( )** – wake up a task pended in ***select*( )**
***selWakeupAll*( )** – wake up all tasks in a ***select*( )** wake-up list
***selNodeAdd*( )** – add a wake-up node to a ***select*( )** wake-up list
***selNodeDelete*( )** – find and delete a node from a ***select*( )** wake-up list
***selWakeupListInit*( )** – initialize a ***select*( )** wake-up list
***selWakeupListLen*( )** – get the number of nodes in a ***select*( )** wake-up list
***selWakeupType*( )** – get the type of a ***select*( )** wake-up node

DESCRIPTION This library provides a BSD 4.3 compatible *select* facility to wait for activity on a set of file descriptors. **selectLib** provides a mechanism that gives a driver the ability to detect pended tasks that are awaiting activity on the driver's device. This allows a driver's interrupt service routine to wake up such tasks directly, eliminating the need for polling. The maximum number of file descriptors supported is 256.

Applications can use ***select*( )** with pipes and serial devices, in addition to sockets. Also, ***select*( )** examines *write* file descriptors in addition to *read* file descriptors; however, exception file descriptors remain unsupported.

Typically, application developers need concern themselves only with the ***select*( )** call. However, driver developers should become familiar with the other routines that may be used with ***select*( )**, if they wish to support the ***select*( )** mechanism.

INCLUDE FILES **selectLib.h**

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

semBLib

NAME **semBLib** – binary semaphore library

ROUTINES ***semBCreate*( )** – create and initialize a binary semaphore

DESCRIPTION This library provides the interface to VxWorks binary semaphores. Binary semaphores are the most versatile, efficient, and conceptually simple type of semaphore. They can be used to: (1) control mutually exclusive access to shared devices or data structures, or (2) synchronize multiple tasks, or task-level and interrupt-level processes. Binary semaphores form the foundation of numerous VxWorks facilities.

A binary semaphore can be viewed as a cell in memory whose contents are in one of two states, full or empty. When a task takes a binary semaphore, using ***semTake*( )**, subsequent action depends on the state of the semaphore:

(1) If the semaphore is full, the semaphore is made empty, and the calling task continues executing.

(2) If the semaphore is empty, the task will be blocked, pending the availability of the semaphore. If a timeout is specified and the timeout expires, the pended task will be removed from the queue of pended tasks and enter the ready state with an ERROR status. A pended task is ineligible for CPU allocation. Any number of tasks may be pended simultaneously on the same binary semaphore.

When a task gives a binary semaphore, using ***semGive*( )**, the next available task in the pend queue is unblocked. If no task is pending on this semaphore, the semaphore becomes full. Note: if a semaphore is given that unblocks a task that is of higher priority than the task that called ***semGive*( )**, the unblocked task will preempt the calling task.

MUTUAL EXCLUSION

To use a binary semaphore as a means of mutual exclusion, first create it with an initial state of full. For example:

```
SEM_ID semMutex;
/* create a binary semaphore that is initially full */
semMutex = semBCreate (SEM_Q_PRIORITY, SEM_FULL);
```

Then guard a critical section or resource by taking the semaphore with ***semTake*( )**, and exit the section or release the resource by giving the semaphore with ***semGive*( )**. For example:

```
semTake (semMutex, WAIT_FOREVER);
    ...  /* critical region, accessible only by one task at a time */

semGive (semMutex);
```

While there is no restriction on the same semaphore being given, taken, or flushed by multiple tasks, it is important to ensure the proper functionality of the mutual-exclusion construct. While there is no danger in any number of processes taking a semaphore, the giving of a semaphore should be more carefully controlled. If a semaphore is given by a task that did not take it, mutual exclusion could be lost.

SYNCHRONIZATION

To use a binary semaphore as a means of synchronization, create it with an initial state of empty. A task blocks by taking a semaphore at a synchronization point, and it remains blocked until the semaphore is given by another task or interrupt service routine.

Synchronization with interrupt service routines is a particularly common need. Binary semaphores can be given, but not taken, from interrupt level. Thus, a task can block at a synchronization point with ***semTake*( )**, and an interrupt service routine can unblock that task with ***semGive*( )**.

In the following example, when ***init*( )** is called, the binary semaphore is created, an interrupt service routine is attached to an event, and a task is spawned to process the event. Task 1 will run until it calls ***semTake*( )**, at which point it will block until an event causes the interrupt service routine to call ***semGive*( )**. When the interrupt service routine completes, task 1 can execute to process the event.

```
SEM_ID semSync;    /* ID of sync semaphore */
init ()
    {
    intConnect (..., eventInterruptSvcRout, ...);
    semSync = semBCreate (SEM_Q_FIFO, SEM_EMPTY);
    taskSpawn (..., task1);
    }
task1 ()
    {
    ...
    semTake (semSync, WAIT_FOREVER);    /* wait for event */
```

```
        ...    /* process event */
        }
    eventInterruptSvcRout ()
        {
        ...
        semGive (semSync);    /* let task 1 process event */
        ...
        }
```

A ***semFlush*( )** on a binary semaphore will atomically unblock all pended tasks in the semaphore queue, i.e., all tasks will be unblocked at once, before any actually execute.

CAVEATS

There is no mechanism to give back or reclaim semaphores automatically when tasks are suspended or deleted. Such a mechanism, though desirable, is not currently feasible. Without explicit knowledge of the state of the guarded resource or region, reckless automatic reclamation of a semaphore could leave the resource in a partial state. Thus, if a task ceases execution unexpectedly, as with a bus error, currently owned semaphores will not be given back, effectively leaving a resource permanently unavailable. The mutual-exclusion semaphores provided by **semMLib** offer protection from unexpected task deletion.

INCLUDE FILES

semLib.h

SEE ALSO

semLib, **semCLib**, **semMLib**, *VxWorks Programmer's Guide: Basic OS*

semCLib

NAME

semCLib – counting semaphore library

ROUTINES

***semCCreate*( )** – create and initialize a counting semaphore

DESCRIPTION

This library provides the interface to VxWorks counting semaphores. Counting semaphores are useful for guarding multiple instances of a resource.

A counting semaphore may be viewed as a cell in memory whose contents keep track of a count. When a task takes a counting semaphore, using ***semTake*( )**, subsequent action depends on the state of the count:

(1) If the count is non-zero, it is decremented and the calling task continues executing.

(2) If the count is zero, the task will be blocked, pending the availability of the semaphore. If a timeout is specified and the timeout expires, the pended task will be removed from the queue of pended tasks and enter the ready state with an ERROR status. A pended task is ineligible for CPU allocation. Any number of tasks may be pended simultaneously on the same counting semaphore.

When a task gives a semaphore, using ***semGive*( )**, the next available task in the pend queue is unblocked. If no task is pending on this semaphore, the semaphore count is incremented. Note that if a semaphore is given, and a task is unblocked that is of higher priority than the task that called ***semGive*( )**, the unblocked task will preempt the calling task.

A ***semFlush*( )** on a counting semaphore will atomically unblock all pended tasks in the semaphore queue. So all tasks will be made ready before any task actually executes. The count of the semaphore will remain unchanged.

INTERRUPT USAGE Counting semaphores may be given but not taken from interrupt level.

CAVEATS There is no mechanism to give back or reclaim semaphores automatically when tasks are suspended or deleted. Such a mechanism, though desirable, is not currently feasible. Without explicit knowledge of the state of the guarded resource or region, reckless automatic reclamation of a semaphore could leave the resource in a partial state. Thus, if a task ceases execution unexpectedly, as with a bus error, currently owned semaphores will not be given back, effectively leaving a resource permanently unavailable. The mutual-exclusion semaphores provided by **semMLib** offer protection from unexpected task deletion.

INCLUDE FILES **semLib.h**

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

---

semLib

NAME **semLib** – general semaphore library

ROUTINES

***semGive*( )** – give a semaphore
***semTake*( )** – take a semaphore
***semFlush*( )** – unblock every task pended on a semaphore
***semDelete*( )** – delete a semaphore

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

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

Semaphore creation and initialization is handled by other libraries, depending on the type of semaphore used. These libraries contain full functional descriptions of the semaphore types:

semBLib – binary semaphores
semCLib – counting semaphores
semMLib – mutual exclusion semaphores
semSmLib – shared memory semaphores

Binary semaphores offer the greatest speed and the broadest applicability.

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

SEMAPHORE CONTROL

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

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

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

SEMAPHORE DELETION

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

SEMAPHORE INFORMATION

The ***semInfo*( )** call is a useful debugging aid, reporting all tasks blocked on a specified semaphore. It provides a snapshot of the queue at the time of the call, but because semaphores are dynamic, the information may be out of date by the time it is available. As with the current state of the semaphore, use of the queue of pended tasks should be restricted to debugging uses only.

INCLUDE FILES **semLib.h**

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

---

semMLib

NAME **semMLib** – mutual-exclusion semaphore library

ROUTINES ***semMCreate*( )** – create and initialize a mutual-exclusion semaphore
***semMGiveForce*( )** – give a mutual-exclusion semaphore without restrictions

DESCRIPTION This library provides the interface to VxWorks mutual-exclusion semaphores. Mutual-exclusion semaphores offer convenient options suited for situations requiring mutually exclusive access to resources. Typical applications include sharing devices and protecting data structures. Mutual-exclusion semaphores are used by many higher-level VxWorks facilities.

The mutual-exclusion semaphore is a specialized version of the binary semaphore, designed to address issues inherent in mutual exclusion, such as recursive access to resources, priority inversion, and deletion safety. The fundamental behavior of the mutual-exclusion semaphore is identical to the binary semaphore (see the manual entry for **semBLib**), except for the following restrictions:

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

These last two operations have no meaning in mutual-exclusion situations.

RECURSIVE RESOURCE ACCESS

A special feature of the mutual-exclusion semaphore is that it may be taken "recursively," i.e., it can be taken more than once by the task that owns it before finally being released. Recursion is useful for a set of routines that need mutually exclusive access to a resource, but may need to call each other.

Recursion is possible because the system keeps track of which task currently owns a mutual-exclusion semaphore. Before being released, a mutual-exclusion semaphore taken recursively must be given the same number of times it has been taken; this is tracked by means of a count which is incremented with each ***semTake*( )** and decremented with each ***semGive*( )**.

The example below illustrates recursive use of a mutual-exclusion semaphore. Function A requires access to a resource which it acquires by taking **semM**; function A may also need to call function B, which also requires **semM**:

```
SEM_ID semM;
semM = semMCreate (...);
funcA ()
    {
    semTake (semM, WAIT_FOREVER);
    ...
    funcB ();
    ...
    semGive (semM);
    }
funcB ()
    {
    semTake (semM, WAIT_FOREVER);
    ...
    semGive (semM);
    }
```

PRIORITY-INVERSION SAFETY

If the option **SEM_INVERSION_SAFE** is selected, the library adopts a priority-inheritance protocol to resolve potential occurrences of "priority inversion," a problem stemming from the use semaphores for mutual exclusion. Priority inversion arises when a higher-priority task is forced to wait an indefinite period of time for the completion of a lower-priority task.

Consider the following scenario: T1, T2, and T3 are tasks of high, medium, and low priority, respectively. T3 has acquired some resource by taking its associated semaphore. When T1 preempts T3 and contends for the resource by taking the same semaphore, it becomes blocked. If we could be assured that T1 would be blocked no longer than the time it normally takes T3 to finish with the resource, the situation would not be problematic. However, the low-priority task is vulnerable to preemption by medium-priority tasks; a preempting task, T2, could inhibit T3 from relinquishing the resource. This condition could persist, blocking T1 for an indefinite period of time.

The priority-inheritance protocol solves the problem of priority inversion by elevating the priority of T3 to the priority of T1 during the time T1 is blocked on T3. This protects T3, and indirectly T1, from preemption by T2. Stated more generally, the priority-inheritance protocol assures that a task which owns a resource will execute at the priority of the highest priority task blocked on that resource. Once the task priority has been elevated, it remains at the higher level until all mutual-exclusion semaphores that the task owns are released; then the task returns to its normal, or standard, priority. Hence, the "inheriting" task is protected from preemption by any intermediate-priority tasks.

The priority-inheritance protocol also takes into consideration a task's ownership of more than one mutual-exclusion semaphore at a time. Such a task will execute at the priority of the highest priority task blocked on any of its owned resources. The task will return to its normal priority only after relinquishing all of its mutual-exclusion semaphores that have the inversion-safety option enabled.

SEMAPHORE DELETION

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

TASK-DELETION SAFETY

If the option **SEM_DELETE_SAFE** is selected, the task owning the semaphore will be protected from deletion as long as it owns the semaphore. This solves another problem endemic to mutual exclusion. Deleting a task executing in a critical region can be catastrophic. The resource could be left in a corrupted state and the semaphore guarding the resource would be unavailable, effectively shutting off all access to the resource.

As discussed in **taskLib**, the primitives ***taskSafe( )*** and ***taskUnsafe( )*** offer one solution, but as this type of protection goes hand in hand with mutual exclusion, the mutual-exclusion semaphore provides the option **SEM_DELETE_SAFE**, which enables an implicit ***taskSafe( )*** with each ***semTake( )***, and a ***taskUnsafe( )*** with each ***semGive( )***. This convenience is also more efficient, as the resulting code requires fewer kernel entrances.

CAVEATS

There is no mechanism to give back or reclaim semaphores automatically when tasks are suspended or deleted. Such a mechanism, though desirable, is not currently feasible. Without explicit knowledge of the state of the guarded resource or region, reckless automatic reclamation of a semaphore could leave the resource in a partial state. Thus if a task ceases execution unexpectedly, as with a bus error, currently owned semaphores will not be given back, effectively leaving a resource permanently unavailable. The **SEM_DELETE_SAFE** option partially protects an application, to the extent that unexpected deletions will be deferred until the resource is released.

Because the priority of a task which has been elevated by the taking of a mutual-exclusion semaphore remains at the higher priority until all mutexes held by that task are released, unbounded priority inversion situations can result when nested mutexes are involved. If nested mutexes are required, consider the following alternatives:

1. Avoid overlapping critical regions.
2. Adjust priorities of tasks so that there are no tasks at intermediate priority levels.
3. Adjust priorities of tasks so that priority inheritance protocol is not needed.
4. Manually implement a static priority ceiling protocol using a non-inversion-save mutex. This involves setting all blockers on a mutex to the ceiling priority, then taking the mutex. After semGive, set the priorities back to the base priority. Note that this implementation reduces the queue to a fifo queue.

INCLUDE FILES

semLib.h

SEE ALSO

semLib, **semBLib**, **semCLib**, *VxWorks Programmer's Guide: Basic OS*

semOLib

NAME

semOLib – release 4.x binary semaphore library

ROUTINES

***semCreate*( )** – create and initialize a release 4.x binary semaphore
***semInit*( )** – initialize a static binary semaphore
***semClear*( )** – take a release 4.x semaphore, if the semaphore is available

DESCRIPTION

This library is provided for backward compatibility with VxWorks 4.x semaphores. The semaphores are identical to 5.0 binary semaphores, except that timeouts -- missing or specified -- are ignored.

For backward compatibility, ***semCreate*( )** operates as before, allocating and initializing a 4.x-style semaphore. Likewise, ***semClear*( )** has been implemented as a ***semTake*( )**, with a timeout of **NO_WAIT**.

For more information on of the behavior of binary semaphores, see the manual entry for **semBLib**.

INCLUDE FILES

semLib.h

SEE ALSO

semLib, **semBLib**, *VxWorks Programmer's Guide: Basic OS*

semPxLib

NAME

semPxLib – semaphore synchronization library (POSIX)

ROUTINES

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

DESCRIPTION

This library implements the POSIX 1003.1b semaphore interface. For alternative semaphore routines designed expressly for VxWorks, see the manual page for **semLib** and other semaphore libraries mentioned there. POSIX semaphores are counting semaphores; as such they are most similar to the **semCLib** VxWorks-specific semaphores.

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

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

TERMINOLOGY

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

SEMAPHORE DELETION

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

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

INCLUDE FILES

semaphore.h

SEE ALSO

POSIX 1003.1b document, **semLib**, *VxWorks Programmer's Guide: Basic OS*

semPxShow

NAME **semPxShow** – POSIX semaphore show library

ROUTINES ***semPxShowInit*( )** – initialize the POSIX semaphore show facility

DESCRIPTION This library provides a show routine for POSIX semaphore objects.

semShow

NAME **semShow** – semaphore show routines

ROUTINES
***semShowInit*( )** – initialize the semaphore show facility
***semInfo*( )** – get a list of task IDs that are blocked on a semaphore
***semShow*( )** – show information about a semaphore

DESCRIPTION This library provides routines to show semaphore statistics, such as semaphore type, semaphore queuing method, tasks pended, etc.

The routine ***semShowInit*( )** links the semaphore show facility into the VxWorks system. It is called automatically when the semaphore show facility is configured into VxWorks using either of the following methods:

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

– If you use the Tornado project facility, select **INCLUDE_SEM_SHOW**.

INCLUDE FILES **semLib.h**

SEE ALSO **semLib**, *VxWorks Programmer's Guide: Basic OS*

semSmLib

NAME

semSmLib – shared memory semaphore library (VxMP Opt.)

ROUTINES

***semBSmCreate*( )** – create and initialize a shared memory binary semaphore (VxMP Opt.)
***semCSmCreate*( )** – create and initialize a shared memory counting semaphore (VxMP Opt.)

DESCRIPTION

This library provides the interface to VxWorks shared memory binary and counting semaphores. Once a shared memory semaphore is created, the generic semaphore-handling routines provided in **semLib** are used to manipulate it. Shared memory binary semaphores are created using ***semBSmCreate*( )**. Shared memory counting semaphores are created using ***semCSmCreate*( )**.

Shared memory binary semaphores are used to: (1) control mutually exclusive access to multiprocessor-shared data structures, or (2) synchronize multiple tasks running in a multiprocessor system. For general information about binary semaphores, see the manual entry **semBLib**.

Shared memory counting semaphores are used for guarding multiple instances of a resource used by multiple CPUs. For general information about shared counting semaphores, see the manual entry for **semCLib**.

For information about the generic semaphore-handling routines, see the manual entry for **semLib**.

MEMORY REQUIREMENTS

The semaphore structure is allocated from a dedicated shared memory partition.

The shared semaphore dedicated shared memory partition is initialized by the shared memory objects master CPU. The size of this partition is defined by the maximum number of shared semaphores, defined by **SM_OBJ_MAX_SEM** in the configuration header file.

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** when the configuration macro **INCLUDE_SM_OBJ** is defined.

AVAILABILITY
This module is distributed as a component of the unbundled shared memory support option, VxMP.

INCLUDE FILES
semSmLib.h

SEE ALSO
semLib, **semBLib**, **semCLib**, **smObjLib**, **semShow**, ***usrSmObjInit*( )**, *VxWorks Programmer's Guide: Shared Memory Objects, Basic OS*

shellLib

NAME
shellLib – shell execution routines

ROUTINES
***shellInit*( )** – start the shell
***shell*( )** – the shell entry point
***shellScriptAbort*( )** – signal the shell to stop processing a script
***shellHistory*( )** – display or set the size of shell history
***shellPromptSet*( )** – change the shell prompt
***shellOrigStdSet*( )** – set the shell's default input/output/error file descriptors
***shellLock*( )** – lock access to the shell

DESCRIPTION
This library contains the execution support routines for the VxWorks shell. It provides the basic programmer's interface to VxWorks. It is a C-expression interpreter, containing no built-in commands.

The nature, use, and syntax of the shell are more fully described in the *VxWorks Programmer's Guide: Target Shell.*

INCLUDE FILES
shellLib.h

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

sigLib

NAME

sigLib – software signal facility library

ROUTINES

***sigInit*( )** – initialize the signal facilities
***sigqueueInit*( )** – initialize the queued signal facilities
***sigemptyset*( )** – initialize a signal set with no signals included (POSIX)
***sigfillset*( )** – initialize a signal set with all signals included (POSIX)
***sigaddset*( )** – add a signal to a signal set (POSIX)
***sigdelset*( )** – delete a signal from a signal set (POSIX)
***sigismember*( )** – test to see if a signal is in a signal set (POSIX)
***signal*( )** – specify the handler associated with a signal
***sigaction*( )** – examine and/or specify the action associated with a signal (POSIX)
***sigprocmask*( )** – examine and/or change the signal mask (POSIX)
***sigpending*( )** – retrieve the set of pending signals blocked from delivery (POSIX)
***sigsuspend*( )** – suspend the task until delivery of a signal (POSIX)
***pause*( )** – suspend the task until delivery of a signal (POSIX)
***sigtimedwait*( )** – wait for a signal
***sigwaitinfo*( )** – wait for real-time signals
***sigvec*( )** – install a signal handler
***sigsetmask*( )** – set the signal mask
***sigblock*( )** – add to a set of blocked signals
***raise*( )** – send a signal to the caller's task
***kill*( )** – send a signal to a task (POSIX)
***sigqueue*( )** – send a queued signal to a task

DESCRIPTION

This library provides a signal interface for tasks. Signals are used to alter the flow control of tasks by communicating asynchronous events within or between task contexts. Any task or interrupt service can "raise" (or send) a signal to a particular task. The task being signaled will immediately suspend its current thread of execution and invoke a task-specified "signal handler" routine. The signal handler is a user-supplied routine that is bound to a specific signal and performs whatever actions are necessary whenever the signal is received. Signals are most appropriate for error and exception handling, rather than as a general purpose intertask communication mechanism.

This library has both a BSD 4.3 and POSIX signal interface. The POSIX interface provides a standardized interface which is more functional than the traditional BSD 4.3 interface. The chart below shows the correlation between BSD 4.3 and POSIX 1003.1 functions. An application should use only one form of interface and not intermix them.

BSD 4.3	POSIX 1003.1
***sigmask*( )**	***sigemptyset*( )**, ***sigfillset*( )**, ***sigaddset*( )**, ***sigdelset*( )**, ***sigismember*( )**
***sigblock*( )**	***sigprocmask*( )**
***sigsetmask*( )**	***sigprocmask*( )**

BSD 4.3	POSIX 1003.1
***pause*( )**	***sigsuspend*( )**
***sigvec*( )**	***sigaction*( )**
(none)	***sigpending*( )**
***signal*( )**	***signal*( )**
***kill*( )**	***kill*( )**

POSIX 1003.1b (Real-Time Extensions) also specifies a queued-signal facility that involves four additional routines: ***sigqueue*( )**, ***sigwaitinfo*( )**, and ***sigtimedwait*( )**.

In many ways, signals are analogous to hardware interrupts. The signal facility provides a set of 31 distinct signals. A signal can be raised by calling ***kill*( )**, which is analogous to an interrupt or hardware exception. A signal handler is bound to a particular signal with ***sigaction*( )** in much the same way that an interrupt service routine is connected to an interrupt vector with ***intConnect*( )**. Signals are blocked for the duration of the signal handler, just as interrupts are locked out for the duration of the interrupt service routine. Tasks can block the occurrence of certain signals with ***sigprocmask*( )**, just as the interrupt level can be raised or lowered to block out levels of interrupts. If a signal is blocked when it is raised, its handler routine will be called when the signal becomes unblocked.

Several routines (***sigprocmask*( )**, ***sigpending*( )**, and ***sigsuspend*( )**) take **sigset_t** data structures as parameters. These data structures are used to specify signal set masks. Several routines are provided for manipulating these data structures: ***sigemptyset*( )** clears all the bits in a **segset_t**, ***sigfillset*( )** sets all the bits in a **sigset_t**, ***sigaddset*( )** sets the bit in a **sigset_t** corresponding to a particular signal number, ***sigdelset*( )** resets the bit in a **sigset_t** corresponding to a particular signal number, and ***sigismember*( )** tests to see if the bit corresponding to a particular signal number is set.

FUNCTION RESTARTING

If a task is pended (for instance, by waiting for a semaphore to become available) and a signal is sent to the task for which the task has a handler installed, then the handler will run before the semaphore is taken. When the handler is done, the task will go back to being pended (waiting for the semaphore). If there was a timeout used for the pend, then the original value will be used again when the task returns from the signal handler and goes back to being pended.

Signal handlers are typically defined as:

```
void sigHandler
    (
    int sig, /* signal number */
    )
    {
        ...
    }
```

In VxWorks, the signal handler is passed additional arguments and can be defined as:

```
void sigHandler
    (
    int sig,                        /* signal number               */
    int code,                       /* additional code             */
    struct sigcontext *pSigContext  /* context of task before signal */
    )
    {
        ...
    }
```

The parameter *code* is valid only for signals caused by hardware exceptions. In this case, it is used to distinguish signal variants. For example, both numeric overflow and zero divide raise **SIGFPE** (floating-point exception) but have different values for *code*. (Note that when the above VxWorks extensions are used, the compiler may issue warnings.)

SIGNAL HANDLER DEFINITION

Signal handling routines must follow one of two specific formats, so that they may be correctly called by the operating system when a signal occurs.

Traditional signal handlers receive the signal number as the sole input parameter. However, certain signals generated by routines which make up the POSIX Real-Time Extensions (P1003.1b) support the passing of an additional application-specific value to the handler routine. These include signals generated by the ***sigqueue*( )** call, by asynchronous I/O, by POSIX real-time timers, and by POSIX message queues.

If a signal handler routine is to receive these additional parameters, **SA_SIGINFO** must be set in the sa_flags field of the sigaction structure which is a parameter to the ***sigaction*( )** routine. Such routines must take the following form:

```
void sigHandler (int sigNum, siginfo_t * pInfo, void * pContext);
```

Traditional signal handling routines must not set **SA_SIGINFO** in the **sa_flags** field, and must take the form of:

```
void sigHandler (int sigNum);
```

EXCEPTION PROCESSING

Certain signals, defined below, are raised automatically when hardware exceptions are encountered. This mechanism allows user-defined exception handlers to be installed. This is useful for recovering from catastrophic events such as bus or arithmetic errors. Typically, ***setjmp*( )** is called to define the point in the program where control will be restored, and ***longjmp*( )** is called in the signal handler to restore that context. Note that ***longjmp*( )** restores the state of the task's signal mask. If a user-defined handler is not installed or the installed handler returns for a signal raised by a hardware exception, then the task is suspended and a message is logged to the console.

The following is a list of hardware exceptions caught by VxWorks and delivered to the offending task. The user may include the higher-level header file **sigCodes.h** in order to access the appropriate architecture-specific header file containing the code value.

Motorola 68K

Signal	Code	Exception
SIGSEGV	**NULL**	bus error
SIGBUS	**BUS_ADDERR**	address error
SIGILL	**ILL_ILLINSTR_FAULT**	illegal instruction
SIGFPE	**FPE_INTDIV_TRAP**	zero divide
SIGFPE	**FPE_CHKINST_TRAP**	chk trap
SIGFPE	**FPE_TRAPV_TRAP**	trapv trap
SIGILL	**ILL_PRIVVIO_FAULT**	privilege violation
SIGTRAP	**NULL**	trace exception
SIGEMT	**EMT_EMU1010**	line 1010 emulator
SIGEMT	**EMT_EMU1111**	line 1111 emulator
SIGILL	**ILL_ILLINSTR_FAULT**	coprocessor protocol violation
SIGFMT	**NULL**	format error
SIGFPE	**FPE_FLTBSUN_TRAP**	compare unordered
SIGFPE	**FPE_FLTINEX_TRAP**	inexact result
SIGFPE	**FPE_FLTDIV_TRAP**	divide by zero
SIGFPE	**FPE_FLTUND_TRAP**	underflow
SIGFPE	**FPE_FLTOPERR_TRAP**	operand error
SIGFPE	**FPE_FLTOVF_TRAP**	overflow
SIGFPE	**FPE_FLTNAN_TRAP**	signaling "Not A Number"

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
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

Signal	Code	Exception
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

Signal	Code	Exception
SIGFPE	**FPE_FLTUND_TRAP**	FPA underflow exception
SIGFPE	**FPE_FLTINEX_TRAP**	FPA inexact operation

Intel i386/i486

Signal	Code	Exception
SIGILL	**ILL_DIVIDE_ERROR**	divide error
SIGEMT	**EMT_DEBUG**	debugger call
SIGILL	**ILL_NON_MASKABLE**	NMI interrupt
SIGEMT	**EMT_BREAKPOINT**	breakpoint
SIGILL	**ILL_OVERFLOW**	INTO-detected overflow
SIGILL	**ILL_BOUND**	bound range exceeded
SIGILL	**ILL_INVALID_OPCODE**	invalid opcode
SIGFPE	**FPE_NO_DEVICE**	device not available
SIGILL	**ILL_DOUBLE_FAULT**	double fault
SIGFPE	**FPE_CP_OVERRUN**	coprocessor segment overrun
SIGILL	**ILL_INVALID_TSS**	invalid task state segment
SIGBUS	**BUS_NO_SEGMENT**	segment not present
SIGBUS	**BUS_STACK_FAULT**	stack exception
SIGILL	**ILL_PROTECTION_FAULT**	general protection
SIGBUS	**BUS_PAGE_FAULT**	page fault
SIGILL	**ILL_RESERVED**	(intel reserved)
SIGFPE	**FPE_CP_ERROR**	coprocessor error
SIGBUS	**BUS_ALIGNMENT**	alignment check

PowerPC

Signal	Code	Exception
SIGBUS	**_EXC_OFF_MACH**	machine check
SIGBUS	**_EXC_OFF_INST**	instruction access
SIGBUS	**_EXC_OFF_ALIGN**	alignment
SIGILL	**_EXC_OFF_PROG**	program
SIGBUS	**_EXC_OFF_DATA**	data access
SIGFPE	**_EXC_OFF_FPU**	floating point unavailable
SIGTRAP	**_EXC_OFF_DBG**	debug exception (PPC403)
SIGTRAP	**_EXC_OFF_INST_BRK**	inst. breakpoint (PPC603, PPCEC603, PPC604)
SIGTRAP	**_EXC_OFF_TRACE**	trace (PPC603, PPCEC603, PPC604, PPC860)
SIGBUS	**_EXC_OFF_CRTL**	critical interrupt (PPC403)
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.)

ROUTINES

***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

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 the configuration header file.

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 a call to ***usrSmObjInit*( )**, which is found in **src/config/usrSmObj.c**. This is done automatically from the root task, ***usrRoot*( )**, in **usrConfig.c**, when the configuration macro **INCLUDE_SM_OBJ** is defined.

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.)

ROUTINES ***smMemShow*( )** – show the shared memory system partition blocks and statistics (VxMP Opt.)

DESCRIPTION This library provides routines to show the statistics on a shared memory system partition.

General shared memory management routines are provided by **smMemLib**.

CONFIGURATION The routines in this library are included by default if **INCLUDE_SM_OBJ** is defined in **configAll.h**.

AVAILABILITY This module is distributed as a component of the unbundled shared memory objects support option, VxMP.

INCLUDE FILES **smLib.h**, **smObjLib.h**, **smMemLib.h**

SEE ALSO **smMemLib**, *VxWorks Programmer's Guide: Shared Memory Objects*

smNameLib

NAME **smNameLib** – shared memory objects name database library (VxMP Opt.)

ROUTINES ***smNameAdd*( )** – add a name to the shared memory name database (VxMP Opt.)
***smNameFind*( )** – look up a shared memory object by name (VxMP Opt.)
***smNameFindByValue*( )** – look up a shared memory object by value (VxMP Opt.)
***smNameRemove*( )** – remove an object from the shared memory objects name database (VxMP Opt.)

DESCRIPTION This library provides facilities for managing the shared memory objects name database. The shared memory objects name database associates a name and object type with a value and makes that information available to all CPUs. A name is an arbitrary, null-terminated string. An object type is a small integer, and its value is a global (shared) ID or a global shared memory address.

Names are added to the shared memory name database with ***smNameAdd*( )**. They are removed by ***smNameRemove*( )**.

Objects in the database can be accessed by either name or value. The routine ***smNameFind*( )** searches the shared memory name database for an object of a specified name. The routine ***smNameFindByValue*( )** searches the shared memory name database for an object of a specified identifier or address.

Name database contents can be viewed using ***smNameShow*( )**.

The maximum number of names to be entered in the database is **SM_OBJ_MAX_NAME**, defined in the configuration header file. 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 "semSmLib.h"
#include "stdio.h"
testSmSem1 (void)
    {
    SEM_ID smSemId;
    /* create a shared semaphore */
    if ((smSemId = semBSmCreate(SEM_Q_FIFO, SEM_EMPTY)) == NULL)
        {
        printf ("Shared semaphore creation error.");
        return (ERROR);
        }
    /*
     * make created semaphore Id available to all CPUs in
     * the system by entering its name in shared name database.
     */
    if (smNameAdd ("smSem", smSemId, T_SM_SEM_B) != OK )
        {
        printf ("Cannot add smSem into shared database.");
        return (ERROR);
        }
```

```
        ...
    /* now use the semaphore */
    semGive (smSemId);
        ...
    }
```

On CPU 2 :

```
#include "vxWorks.h"
#include "semLib.h"
#include "smNameLib.h"
#include "stdio.h"
testSmSem2 (void)
    {
    SEM_ID smSemId;
    int    objType;       /* place holder for smNameFind() object type */
    /* get semaphore ID from name database */

    smNameFind ("smSem", (void **) &smSemId, &objType, WAIT_FOREVER);
        ...
    /* now that we have the shared semaphore ID, take it */

    semTake (smSemId, WAIT_FOREVER);
        ...
    }
```

CONFIGURATION

Before routines in this library can be called, the shared memory object facility must be initialized by calling ***usrSmObjInit*( )**, which is found in **src/config/usrSmObj.c**. This is done automatically from the root task, ***usrRoot*( )**, in **usrConfig.c** when the configuration macro **INCLUDE_SM_OBJ** is defined.

AVAILABILITY

This module is distributed as a component of the unbundled shared memory objects support option, VxMP.

INCLUDE FILES

smNameLib.h

SEE ALSO

smNameShow, **smObjLib**, **smObjShow**, ***usrSmObjInit*( )**,
VxWorks Programmer's Guide: Shared Memory Objects

smNameShow

NAME **smNameShow** – shared memory objects name database show routines (VxMP Opt.)

ROUTINES ***smNameShow*( )** – show the contents of the shared memory objects name database

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 distributed as a component of the unbundled shared memory objects support option, VxMP.

INCLUDE FILES **smNameLib.h**

SEE ALSO **smObjLib**, *VxWorks Programmer's Guide: Shared Memory Objects*

smNetLib

NAME **smNetLib** – VxWorks interface to the shared memory network (backplane) driver

ROUTINES ***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

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

ROUTINES ***smNetShow( )*** – show information about a shared memory network

DESCRIPTION This library provides show routines for the shared memory network interface driver.

The ***smNetShow( )*** routine is provided as a diagnostic aid to show current shared memory network status.

INCLUDE FILES **smPktLib.h**

SEE ALSO *VxWorks Programmer's Guide: Network*

smObjLib

NAME **smObjLib** – shared memory objects library (VxMP Opt.)

ROUTINES ***smObjLibInit( )*** – install the shared memory objects facility (VxMP Opt.)
smObjSetup( ) – initialize the shared memory objects facility (VxMP Opt.)
smObjInit( ) – initialize a shared memory objects descriptor (VxMP Opt.)
smObjAttach( ) – attach the calling CPU to shared memory objects facility (VxMP Opt.)
smObjLocalToGlobal( ) – convert a local address to a global address (VxMP Opt.)
smObjGlobalToLocal( ) – convert a global address to a local address (VxMP Opt.)
smObjTimeoutLogEnable( ) – enable/disable logging of failed attempts to take a spin-lock (VxMP Opt.)

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 the configuration macro **INCLUDE_SM_OBJ** is defined, 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 distributed as a component of the unbundled shared memory objects support option, VxMP.

INCLUDE FILES

smObjLib.h

SEE ALSO

smObjShow, **semSmLib**, **msgQSmLib**, **smMemLib**, **smNameLib**, ***usrSmObjInit*( )**, *VxWorks Programmer's Guide: Shared Memory Objects*

smObjShow

NAME
smObjShow – shared memory objects show routines (VxMP Opt.)

ROUTINES
***smObjShow*( )** – display the current status of shared memory objects (VxMP Opt.)

DESCRIPTION
This library provides routines to show shared memory object statistics, such as the current number of shared tasks, semaphores, message queues, etc.

CONFIGURATION
The routines in this library are included by default if **INCLUDE_SM_OBJ** is defined in **configAll.h**.

AVAILABILITY
This module is distributed as a component of the unbundled shared memory objects support option, VxMP.

INCLUDE FILES
smObjLib.h

SEE ALSO
smObjLib, *VxWorks Programmer's Guide: Shared Memory Objects*

sn83932End

NAME
sn83932End – Nat. Semi DP83932B SONIC Ethernet driver

ROUTINES
***sn83932EndLoad*( )** – initialize the driver and device

DESCRIPTION
This module implements the National Semiconductor DP83932 SONIC Ethernet network interface driver.

This driver is designed to be moderately generic. Thus, it operates unmodified across the range of architectures and targets supported by VxWorks. To achieve this, the driver load routine requires several target-specific parameters. The driver also depends on a few external support routines. These parameters and support routines are described below. If any of the assumptions stated below are not true for your particular hardware, this driver probably cannot function correctly with that hardware. 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 END external interface. Thus, the only normal external interface is the ***sn83932EndLoad*( )** routine, although ***snEndClkEnable*( )** and ***snEndClkDisable*( )**

are provided for the use (optional) of the internal clock. All required parameters are passed into the load function by means of a single colon-delimited string. The ***sn83932Load*( )** function uses ***strtok*( )** to parse the string, which it expects to be of the following format:

unit_ID:*devIO_addr*:*ivec*:*e_addr*

The entry point for ***sn83932EndLoad*( )** is defined within the **endDevTbl** in **configNet.h**.

TARGET-SPECIFIC PARAMETERS

unit_ID
A convenient holdover from the former model, this is only used in the string name for the driver.

devIO_addr
Denotes the base address of the device's I/O register set.

ivec
Denotes the interrupt vector to be used by the driver to service an interrupt from the SONIC device. The driver connects the interrupt handler to this vector by calling ***intConnect*( )**.

e_addr
This parameter is obtained by calling ***sysEnetAddrGet*( )**, 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 the following external support routines:

***sysEnetInit*( )**
```
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, during the unit start-up phase.

***sysEnetAddrGet*( )**
```
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, during the unit start-up phase.

***sysEnetIntEnable*( )**
```
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, during the unit start-up phase.

sysEnetIntAck( )

```
void sysEnetIntAck (int unit)
```

This routine performs any interrupt acknowledgment 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, **snEndDcr** and **snEndDcr2**, 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 **snEndDcr** and **snEndDcr2** variables should be modified before loading the driver. See the SONIC manual for information on appropriate values for 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 can 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**

sntpcLib

NAME **sntpcLib** – Simple Network Time Protocol (SNTP) client library

ROUTINES ***sntpcTimeGet*( )** – retrieve the current time from a remote source

DESCRIPTION This library implements the client side of the Simple Network Time Protocol (SNTP), a protocol that allows a system to maintain the accuracy of its internal clock based on time values reported by one or more remote sources. The library is included in the VxWorks image if **INCLUDE_SNTPC** is defined at the time the image is built.

USER INTERFACE
The ***sntpcTimeGet*( )** routine retrieves the time reported by a remote source and converts that value for POSIX-compliant clocks. The routine will either send a request and extract the time from the reply, or it will wait until a message is received from an SNTP/NTP server executing in broadcast mode.

INCLUDE FILES
sntpcLib.h

SEE ALSO
clockLib, RFC 1769

---

sntpsLib

NAME
sntpsLib – Simple Network Time Protocol (SNTP) server library

ROUTINES
***sntpsClockSet*( )** – assign a routine to access the reference clock
***sntpsNsecToFraction*( )** – convert portions of a second to NTP format
***sntpsConfigSet*( )** – change SNTP server broadcast settings

DESCRIPTION
This library implements the server side of the Simple Network Time Protocol (SNTP), a protocol that allows a system to maintain the accuracy of its internal clock based on time values reported by one or more remote sources. The library is included in the VxWorks image if **INCLUDE_SNTPS** is defined at the time the image is built.

USER INTERFACE
The routine ***sntpsInit*( )** is called automatically during system startup when the SNTP server library is included in the VxWorks image. Depending on the value of **SNTPS_MODE**, the server executes in either a passive or an active mode. When **SNTPS_MODE** is set to **SNTP_PASSIVE** (0x2), the server waits for requests from clients, and sends replies containing an NTP timestamp. When the mode is set to **SNTP_ACTIVE** (0x1), the server transmits NTP timestamp information at fixed intervals.

When executing in active mode, the SNTP server uses the **SNTPS_DSTADDR** and **SNTPS_INTERVAL** definitions to determine the target IP address and broadcast interval. By default, the server will transmit the timestamp information to the local subnet broadcast address every 64 seconds. These settings can be changed with a call to the ***sntpsConfigSet*( )** routine. The SNTP server operating in active mode will still respond to client requests.

The **SNTP_PORT** definition in assigns the source and destination UDP port. The default port setting is 123 as specified by the relevant RFC. Finally, the SNTP server requires access to a reliable external time source. The **SNTPS_TIME_HOOK** constant specifies the name of a routine with the following interface:

```
STATUS sntpsTimeHook (int request, void *pBuffer);
```

This routine can be assigned directly by altering the value of **SNTPS_TIME_HOOK** or can be installed by a call to the ***sntpsClockSet*( )** routine. The manual pages for ***sntpsClockSet*( )** describe the parameters and required operation of the timestamp retrieval routine. Until this routine is specified, the SNTP server will not provide timestamp information.

INCLUDE FILES **sntpsLib.h**

SEE ALSO **sntpcLib**, RFC 1769

sockLib

NAME **sockLib** – generic socket library

ROUTINES

***socket*( )** – open a socket
***bind*( )** – bind a name to a socket
***listen*( )** – enable connections to a socket
***accept*( )** – accept a connection from a socket
***connect*( )** – initiate a connection to a socket
***connectWithTimeout*( )** – try to connect 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

DESCRIPTION This library provides UNIX BSD 4.4 compatible socket calls. Use these calls to open, close, read, and write sockets. These sockets can join processes on the same CPU or on different CPUs between which there is a network connection. The calling sequences of these routines are identical to their equivalents under UNIX BSD 4.4.

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

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

SIOCATMARK
Reports whether there is out-of-band data to be read from the socket. On the return of ***ioctl( )***, *atMark* is TRUE (1) if there is out-of-band data. Otherwise, it is FALSE (0).

```
status = ioctl (sFd, SIOCATMARK, &atMark);
```

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

ROUTINES ***spyLibInit( )*** – initialize task cpu utilization tool package

DESCRIPTION This library provides a facility to monitor tasks' use of the CPU. The primary interface routine, ***spy( )***, periodically calls ***spyReport( )*** to display the amount of CPU time utilized by each task, the amount of time spent at interrupt level, the amount of time spent in the kernel, and the amount of idle time. It also displays the total usage since the start of ***spy( )*** (or the last call to ***spyClkStart( )***), and the change in usage since the last ***spyReport( )***.

CPU usage can also be monitored manually by calling ***spyClkStart( )*** and ***spyReport( )***, instead of ***spy( )***. In this case, ***spyReport( )*** provides a one-time report of the same information provided by ***spy( )***.

Data is gathered by an interrupt-level routine that is connected by ***spyClkStart( )*** to the auxiliary clock. Currently, this facility cannot be used with CPUs that have no auxiliary clock. Interrupts that are at a higher level than the auxiliary clock's interrupt level cannot be monitored.

All user interface routine except ***spyLibInit( )*** are available through **usrLib**.

EXAMPLE

The following call:

```
-> spy 10, 200
```

will generate a report in the following format every 10 seconds, gathering data at the rate of 200 times per second.

```
NAME          ENTRY        TID   PRI total % (ticks)  delta % (ticks)
--------     --------     -----  ---  ---------------  ---------------
tExcTask     _excTask     fbb58    0    0% (       0)    0% (       0)
tLogTask     _logTask     fa6e0    0    0% (       0)    0% (       0)
tShell       _shell       e28a8    1    0% (       4)    0% (       0)
tRlogind     _rlogind     f08dc    2    0% (       0)    0% (       0)
tRlogOutTask _rlogOutTa   e93e0    2    2% (     173)    2% (      46)
tRlogInTask  _rlogInTas   e7f10    2    0% (       0)    0% (       0)
tSpyTask     _spyTask     ffe9c    5    1% (     116)    1% (      28)
tNetTask     _netTask     f3e2c   50    0% (       4)    0% (       1)
tPortmapd    _portmapd    ef240  100    0% (       0)    0% (       0)
KERNEL                                  1% (     105)    0% (      10)
INTERRUPT                               0% (       0)    0% (       0)
IDLE                                   95% (    7990)   95% (    1998)
TOTAL                                  99% (    8337)   98% (    2083)
```

The "total" column reflects CPU activity since the initial call to ***spy*( )** or the last call to ***spyClkStart*( )**. The "delta" column reflects activity since the previous report. A call to ***spyReport*( )** will produce a single report; however, the initial auxiliary clock interrupts and data collection must first be started using ***spyClkStart*( )**.

Data collection/clock interrupts and periodic reporting are stopped by calling:

```
-> spyStop
```

INCLUDE FILES **spyLib.h**

SEE ALSO **usrLib**

sramDrv

NAME **sramDrv** – PCMCIA SRAM device driver

ROUTINES

***sramDrv*( )** – install a PCMCIA SRAM memory driver
***sramMap*( )** – map PCMCIA memory onto a specified ISA address space
***sramDevCreate*( )** – create a PCMCIA memory disk device

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: ***sramDrv*( )** 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 ***sramDrv*( )**. 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

---

st16552Sio

NAME

st16552Sio – ST 16C552 DUART tty driver

ROUTINES

***st16552DevInit*( )** – initialise an ST16552 channel
***st16552IntWr*( )** – handle a transmitter interrupt
***st16552IntRd*( )** – handle a receiver interrupt
***st16552IntEx*( )** – miscellaneous interrupt processing
***st16552Int*( )** – interrupt level processing
***st16552MuxInt*( )** – multiplexed interrupt level processing

DESCRIPTION

This is the device driver for the Startech ST16C552 DUART, similar, but not quite identical to the National Semiconductor 16550 UART.

The chip is a dual universal asynchronous receiver/transmitter with 16 byte transmit and receive FIFOs and a programmable baud-rate generator. Full modem control capability is included and control over the four interrupts that can be generated: Tx, Rx, Line status, and modem status. Only the Rx and Tx interrupts are used by this driver. The FIFOs are enabled for both Tx and Rx by this driver.

Only asynchronous serial operation is supported by the UART which supports 5 to 8 bit bit word lengths with or without parity and with one or two stop bits. The only serial word format supported by the driver is 8 data bits, 1 stop bit, no parity, The default baud rate is determined by the BSP by filling in the **ST16552_CHAN** structure before calling ***ambaDevInit*( )**.

The exact baud rates supported by this driver will depend on the crystal fitted (and consequently the input clock to the baud-rate generator), but in general, baud rates from about 50 to about 115200 are possible.

DATA STRUCTURES

An **ST16552_CHAN** data structure is used to describe the two channels of the chip and, if necessary, an **ST16552_MUX** structure is used to describe the multiplexing of the interrupts for the two channels of the DUART. These structures are described in **h/drv/sio/ambaSio.h**.

CALLBACKS

Servicing a "transmitter ready" interrupt involves making a callback to a higher level library in order to get a character to transmit. By default, this driver installs dummy callback routines which do nothing. A higher layer library that wants to use this driver (e.g. **ttyDrv**) will install its own callback routine using the **SIO_INSTALL_CALLBACK** ioctl command. Likewise, a receiver interrupt handler makes a callback to pass the character to the higher layer library.

MODES

This driver supports both polled and interrupt modes.

USAGE

The driver is typically called only by the BSP. This module's directly callable routines are ***st16552DevInit*( )**, ***st16552Int*( )**, ***st16552IntRd*( )**, ***st16552IntWr*( )**, and ***st16552MuxInt*( )**.

The BSP's ***sysHwInit*( )** routine typically calls ***sysSerialHwInit*( )**, which initialises all the hardware-specific values in the **ST16552_CHAN** structure before calling ***st16552DevInit*( )** which resets the device and installs the driver function pointers. After this the UART will be enabled and ready to generate interrupts, but those interrupts will be disabled in the interrupt controller.

The following example shows the first parts of the initialization:

```
#include "drv/sio/st16552Sio.h"
LOCAL ST16552_CHAN st16552Chan[N_16552_CHANNELS];
void sysSerialHwInit (void)
    {
    int i;
    for (i = 0; i < N_16552_CHANNELS; i++)
        {
        st16552Chan[i].regDelta = devParas[i].regSpace;
        st16552Chan[i].regs = devParas[i].baseAdrs;
        st16552Chan[i].baudRate = CONSOLE_BAUD_RATE;
        st16552Chan[i].xtal = UART_XTAL_FREQ;
        st16552Chan[i].level = devParas[i].intLevel;
        /*
         * Initialise driver functions, getTxChar, putRcvChar and
         * channelMode and init UART.
         */
        st16552DevInit(&st16552Chan[i]);
        }
    }
```

The BSP's ***sysHwInit2*( )** routine typically calls ***sysSerialHwInit2*( )**, which connects the chips interrupts via ***intConnect*( )** (either the single interrupt **st16552Int**, the three interrupts **st16552IntWr**, **st16552IntRd**, and **st16552IntEx**, or the multiplexed interrupt handler **st16552MuxInt** which will cope with both channels of a DUART producing the same interrupt). It then enables those interrupts in the interrupt controller as shown in the following example:

```
void sysSerialHwInit2 (void)
    {
    /* Connect the multiplexed interrupt handler */
    (void) intConnect (INUM_TO_IVEC(devParas[0].vector),
                        st16552MuxInt, (int) &st16552Mux);
    intEnable (devParas[0].intLevel);
    }
```

BSP

By convention all the BSP-specific serial initialisation is performed in a file called **sysSerial.c**, which is #include'ed by **sysLib.c**. **sysSerial.c** implements at least four functions, ***sysSerialHwInit*( )**, ***sysSerialHwInit2*( )**, ***sysSerialChanGet*( )**, and ***sysSerialReset*( )**. The first two have been described above, the others work as follows:

***sysSerialChanGet*( )** is called by ***usrRoot*( )** to get the serial channel descriptor associated with a serial channel number. The routine takes a single parameter which is a channel number ranging between zero and **NUM_TTY**. It returns a pointer to the corresponding channel descriptor, **SIO_CHAN ***, which is just the address of the **ST16552_CHAN** strucure.

***sysSerialReset*( )** is called from ***sysToMonitor*( )** and should reset the serial devices to an inactive state (prevent them from generating any interrupts).

INCLUDE FILES

drv/sio/st16552Sio.h, **sioLib.h**

SEE ALSO

Startech ST16C552 Data Sheet

---

subagentLib

NAME

subagentLib – encode, decode, and process agent and subagent messages

ROUTINES

***snmpSubEncode*( )** – encode a packet for transmission to master agent or subagent
***snmpSaHandlerAsync*( )** – asynchronous message processing routine for the subagent
***snmpSaHandlerWR*( )** – provide ***snmpSaHandlerAsync*( )** functionality synchronously
***snmpSaHandlerContinue*( )** – subagent continuation function
***snmpSaHandlerFinish*( )** – encode packet for subagent I/O completion

***snmpSaHandlerCleanup*( )** – cleanup routine for subagent
***snmpMasterHandlerAsync*( )** – process messages from the subagent asynchronously
***snmpMasterHandlerWR*( )** – synchronous version of ***snmpMasterHandlerAsync*( )**
***snmpMasterQueryHandler*( )** – handles replies from the subagent
***snmpMasterCleanup*( )** – free up resources after a query times out

DESCRIPTION

This module provides the core routines for processing the messages passed between the SNMP master agent and its subagents. Thus, this library includes routines for encoding and decoding a package. It also includes the routines used to sort the messages according to type and then respond to each specific message appropriately.

symLib

NAME

symLib – symbol table subroutine library

ROUTINES

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

DESCRIPTION

This library provides facilities for managing symbol tables. A symbol table associates a name and type with a value. A name is simply an arbitrary, null-terminated string. A symbol type is a small integer (typedef **SYM_TYPE**), and its value is a 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 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

ROUTINES

***symSyncLibInit*( )** – initialize host/target symbol table synchronization
***symSyncTimeoutSet*( )** – set WTX timeout
***syncTgtSafeModCheck*( )** – check if a target module can be safely used

DESCRIPTION

This module provides host/target symbol table synchronization. With synchronization, every module or symbol added to the run-time system from either the target or host side can be seen by facilities on both the target and the host. Symbol-table synchronization makes it possible to use host tools to debug application modules loaded with the target loader or from a target file system. To enable synchronization, two actions must be performed:

1 The module is initialized by ***symSyncLibInit*( )**, which is called automatically when the configuration macro **INCLUDE_SYM_TBL_SYNC** is defined.

2 The target server is launched with the **-s** option.

If synchronization is enabled, **symSyncLib** spawns a synchronization task on the target, **tSymSync**. This task behaves as a WTX tool and attaches itself to the target server. When the task starts, it synchronizes target and host symbol tables so that every module loaded on the target before the target server was started can be seen by the host tools. This feature is particularly useful if VxWorks is started with a target-based startup script before the target server has been launched.

The **tSymSync** task also assures synchronization as new symbols are added by either the target or the host tools. The task waits for synchronization events on two channels: a WTX event from the host or a message queue additon from the target.

The **tSymSync** task, like all WTX tools, must be able to connect to the WTX registry. To make the WTX registry accissible from the target, do one of the following:

1 Boot the target from a host on the same subnet as the registry.

2 Start the registry on the same host the target boots from.

3 Add the needed routes with ***routeAdd*( )** calls, possibly in a startup script.

Neither the host tools nor the target loader wait for synchronization completion to return. To know when the synchronization is complete, you can wait for the corresponding event sent by the target server, or, if your target server was started with the **-V** option, it prints a message indicating synchronization has 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*( )**, and ***symRemove*( )**; however, if an application sends more than 15 synchronization events, it will fill the message queue and then need to wait for a synchronization event to be processed by **tSymSync**. Also, waiting for host synchronization events is done by polling; thus there may be some impact on performance if there are lower-priority tasks than **tSymSync**. If no more synchronization is needed, **tSymSync** can be suspended.

Known problem: Modules with undefined symbols that are loaded from the target are not synchronized; however, they are synchronized if they are loaded from the host.

SEE ALSO

tgtsvr

sysLib

NAME

sysLib – system-dependent library

ROUTINES

***sysClkConnect*( )** – connect a routine to the system clock interrupt
***sysClkDisable*( )** – turn off system clock interrupts
***sysClkEnable*( )** – turn on system clock interrupts
***sysClkRateGet*( )** – get the system clock rate
***sysClkRateSet*( )** – set the system clock rate
***sysAuxClkConnect*( )** – connect a routine to the auxiliary clock interrupt
***sysAuxClkDisable*( )** – turn off auxiliary clock interrupts
***sysAuxClkEnable*( )** – turn on auxiliary clock interrupts
***sysAuxClkRateGet*( )** – get the auxiliary clock rate
***sysAuxClkRateSet*( )** – set the auxiliary clock rate
***sysIntDisable*( )** – disable a bus interrupt level
***sysIntEnable*( )** – enable a bus interrupt level
***sysBusIntAck*( )** – acknowledge a bus interrupt
***sysBusIntGen*( )** – generate a bus interrupt
***sysMailboxConnect*( )** – connect a routine to the mailbox interrupt
***sysMailboxEnable*( )** – enable the mailbox interrupt
***sysNvRamGet*( )** – get the contents of non-volatile RAM
***sysNvRamSet*( )** – write to non-volatile RAM
***sysModel*( )** – return the model name of the CPU board
***sysBspRev*( )** – return the BSP version and revision number
***sysHwInit*( )** – initialize the system hardware
***sysPhysMemTop*( )** – get the address of the top of memory
***sysMemTop*( )** – get the address of the top of logical memory
***sysToMonitor*( )** – transfer control to the ROM monitor
***sysProcNumGet*( )** – get the processor number
***sysProcNumSet*( )** – set the processor number
***sysBusTas*( )** – test and set a location across the bus
***sysScsiBusReset*( )** – assert the RST line on the SCSI bus (Western Digital WD33C93 only)
***sysScsiInit*( )** – initialize an on-board SCSI port
***sysScsiConfig*( )** – system SCSI configuration
***sysLocalToBusAdrs*( )** – convert a local address to a bus address
***sysBusToLocalAdrs*( )** – convert a bus address to a local address
***sysSerialHwInit*( )** – initialize the BSP serial devices to a quiesent state
***sysSerialHwInit2*( )** – connect BSP serial device interrupts
***sysSerialReset*( )** – reset all SIO devices to a quiet state
***sysSerialChanGet*( )** – get the **SIO_CHAN** device associated with a serial channel

DESCRIPTION

This library provides board-specific routines.

NOTE: This is a generic reference entry for a BSP-specific library; this description contains general information only. For features and capabilities specific to the system library included in your BSP, see your BSP's reference entry for **sysLib**.

The file **sysLib.c** provides the board-level interface on which VxWorks and application code can be built in a hardware-independent manner. The functions addressed in this file include:

Initialization functions
– initialize the hardware to a known state
– identify the system
– initialize drivers, such as SCSI or custom drivers

Memory/address space functions
– get the on-board memory size
– make on-board memory accessible to external bus
– map local and bus address spaces
– enable/disable cache memory
– set/get nonvolatile RAM (NVRAM)
– define board's memory map (optional)
– virtual-to-physical memory map declarations for processors with MMUs

Bus interrupt functions
– enable/disable bus interrupt levels
– generate bus interrupts

Clock/timer functions
– enable/disable timer interrupts
– set the periodic rate of the timer

Mailbox/location monitor functions
– enable mailbox/location monitor interrupts for VME-based boards

The **sysLib** library does not support every feature of every board; a particular board may have various extensions to the capabilities described here. Conversely, some boards do not support every function provided by this library. Some boards provide some of the functions of this library by means of hardware switches, jumpers, or PALs, instead of software-controllable registers.

Typically, most functions in this library are not called by the user application directly. The configuration modules **usrConfig.c** and **bootConfig.c** are responsible for invoking the routines at the appropriate time. Device drivers may use some of the memory mapping routines and bus functions.

INCLUDE FILES **sysLib.h**

SEE ALSO *VxWorks Programmer's Guide: Configuration and Build,* BSP-specific reference entry for **sysLib**

tapeFsLib

NAME

tapeFsLib – tape sequential device file system library

ROUTINES

***tapeFsDevInit*( )** – associate a sequential device with tape volume functions
***tapeFsInit*( )** – initialize the tape volume library
***tapeFsReadyChange*( )** – notify **tapeFsLib** of a change in ready status
***tapeFsVolUnmount*( )** – disable a tape device volume

DESCRIPTION

This library provides basic services for tape devices that do not use a standard file or directory structure on tape. The tape volume is treated much like a large file. The tape may either be read or written. However, there is no high-level organization of the tape into files or directories, which must be provided by a higher-level layer.

USING THIS LIBRARY

The various routines provided by the VxWorks tape file system, or tapeFs, can be categorized into three broad groupings: general initialization, device initialization, and file system operation.

The ***tapeFsInit*( )** routine is the principal general initialization function; it needs to be called only once, regardless of how many tapeFs devices are used.

To initialize devices, ***tapeFsDevInit*( )** must be called for each tapeFs device.

Use of this library typically occurs through standard use of the I/O system routines ***open*( )**, ***close*( )**, ***read*( )**, ***write*( )** and ***ioctl*( )**. Besides these standard I/O system operations, several routines are provided to inform the file system of changes in the system environment. The ***tapeFsVolUnmount*( )** routine informs the file system that a particular device should be unmounted; any synchronization should be done prior to invocation of this routine, in preparation for a tape volume change. The ***tapeFsReadyChange*( )** routine is used to inform the file system that a tape may have been swapped and that the next tape operation should first remount the tape. Information about a ready-change is also obtained from the driver using the **SEQ_DEV** device structure. Note that ***tapeFsVolUnmount*( )** and ***tapeFsReadyChange*( )** should be called only after a file has been closed.

INITIALIZATION OF THE FILE SYSTEM

Before any other routines in **tapeFsLib** can be used, ***tapeFsInit*( )** must be called to initialize the library. This implementation of the tape file system assumes only one file descriptor per volume. However, this constraint can be changed in case a future implementation demands multiple file descriptors per volume.

During the ***tapeFsInit*( )** call, the tape device library is installed as a driver in the I/O system driver table. The driver number associated with it is then placed in a global variable, **tapeFsDrvNum**.

To enable this initialization, define **INCLUDE_TAPEFS** in the BSP, or simply start using the tape file system with a call to ***tapeFsDevInit*( )** and ***tapeFsInit*( )** will be called automatically if it has not been called before.

DEFINING A TAPE DEVICE

To use this library for a particular device, the device structure used by the device driver must contain, as the very first item, a sequential device description structure (**SEQ_DEV**). The **SEQ_DEV** must be initialized before calling ***tapeFsDevInit*( )**. The driver places in the **SEQ_DEV** structure the addresses of routines that it must supply: one that reads one or more blocks, one that writes one or more blocks, one that performs I/O control (***ioctl*( )**) on the device, one that writes file marks on a tape, one that rewinds the tape volume, one that reserves a tape device for use, one that releases a tape device after use, one that mounts/unmounts a volume, one that spaces forward or backwards by blocks or file marks, one that erases the tape, one that resets the tape device, and one that checks the status of the device. The **SEQ_DEV** structure also contains fields that describe the physical configuration of the device. For more information about defining sequential devices, see the *VxWorks Programmer's Guide: I/O System.*

INITIALIZATION OF THE DEVICE

The ***tapeFsDevInit*( )** routine is used to associate a device with the **tapeFsLib**functions. The **volName** parameter expected by ***tapeFsDevInit*( )** is a pointer to a name string which identifies the device. This string serves as the pathname for I/O operations which operate on the device and appears in the I/O system device table, which can be displayed using ***iosDevShow*( )**.

The **pSeqDev** parameter expected by ***tapeFsDevInit*( )** is a pointer to the **SEQ_DEV** structure describing the device and containing the addresses of the required driver functions.

The **pTapeConfig** parameter is a pointer to a **TAPE_CONFIG** structure that contains information specifying how the tape device should be configured. The configuration items are fixed/variable block size, rewind/no-rewind device, and number of file marks to be written. For more information about the **TAPE_CONFIG** structure, look at the header file **tapeFsLib.h**.

The syntax of the ***tapeFsDevInit*( )** routine is as follows:

```
tapeFsDevInit
    (
    char *        volName,     /* name to be used for volume   */
    SEQ_DEV *     pSeqDev,     /* pointer to device descriptor */
    TAPE_CONFIG * pTapeConfig  /* pointer to tape config info  */
    )
```

When **tapeFsLib** receives a request from the I/O system, after ***tapeFsDevInit*( )** has been called, it calls the device driver routines (whose addresses were passed in the **SEQ_DEV** structure) to access the device.

OPENING AND CLOSING A FILE

A tape volume is opened by calling the I/O system routine ***open*( )**. A file can be opened only with the **O_RDONLY** or **O_WRONLY** flags. The **O_RDWR** mode is not used by this library. A call to ***open*( )** initializes the file descriptor buffer and state information, reserves the tape device, rewinds the tape device if it was configured as a rewind device, and mounts a volume. Once a tape volume has been opened, that tape device is reserved, disallowing any other system from accessing that device until the tape volume is closed. Also, the single file descriptor is marked "in use" until the file is closed, making sure that a file descriptor is not opened multiple times.

A tape device is closed by calling the I/O system routine ***close*( )**. Upon a ***close*( )** request, any unwritten buffers are flushed, the device is rewound (if it is a rewind device), and, finally, the device is released.

UNMOUNTING VOLUMES (CHANGING TAPES)

A tape volume should be unmounted before it is removed. When unmounting a volume, make sure that any open file is closed first. A tape may be unmounted by calling ***tapeFsVolUnmount*( )** directly.

If a file is open, it is not correct to change the medium and continue with the same file descriptor still open. Since tapeFs assumes only one file descriptor per device, to reuse that device, the file must be closed and opened later for the new tape volume.

Before ***tapeFsVolUnmount*( )** is called, the device should be synchronized by invoking the ***ioctl*( )** **FIOSYNC** or **FIOFLUSH**. It is the responsibility of the higher-level layer to synchronize the tape file system before unmounting. Failure to synchronize the volume before unmounting may result in loss of data.

IOCTL FUNCTIONS

The VxWorks tape sequential device file system supports the following ***ioctl*( )** functions. The functions listed are defined in the header files **ioLib.h** and **tapeFsLib.h**.

FIOFLUSH
Writes all modified file descriptor buffers to the physical device.

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

FIOSYNC
Performs the same function as FIOFLUSH.

FIOBLKSIZEGET
Returns the value of the block size set on the physical device. This value is compared against the **sd_blkSize** value set in the **SEQ_DEV** device structure.

FIOBLKSIZESET
Sets a specified block size value on the physical device and also updates the value in the **SEQ_DEV** and **TAPE_VOL_DESC** structures, unless the supplied value is zero, in which case the device structures are updated but the device is not set to zero. This is because zero implies variable block operations, therefore the device block size is ignored.

MTIOCTOP
Allows use of the standard UNIX MTIO **ioctl** operations by means of the MTOP structure. The MTOP structure appears as follows:

```
typedef struct mtop
    {
    short       mt_op;                  /* operation */
    int         mt_count;               /* number of operations */
    } MTOP;
```

Use these ***ioctl*( )** operations as follows:

```
    MTOP mtop;
    mtop.mt_op    = MTWEOF;
    mtop.mt_count = 1;
    status = ioctl (fd, MTIOCTOP, (int) &mtop);
```

The 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 the status of the device, thus setting the appropriate **SEQ_DEV** fields.

MTRETEN
Retension 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

ROUTINES
***taskSRSet*( )** – set the task status register (MC680x0, MIPS, i386/i486)
***taskSRInit*( )** – initialize the default task status register (MIPS)

DESCRIPTION This library provides architecture-specific task management routines that set and examine architecture-dependent registers. For information about architecture-independent task management facilities, see the manual entry for **taskLib**.

NOTE There are no application-level routines in **taskArchLib** for SPARC.

INCLUDE FILES **regs.h**, **taskArchLib.h**

SEE ALSO **taskLib**

taskHookLib

NAME **taskHookLib** – task hook library

ROUTINES
***taskHookInit*( )** – initialize task hook facilities
***taskCreateHookAdd*( )** – add a routine to be called at every task create
***taskCreateHookDelete*( )** – delete a previously added task create routine

***taskSwitchHookAdd*( )** – add a routine to be called at every task switch
***taskSwitchHookDelete*( )** – delete a previously added task switch routine
***taskDeleteHookAdd*( )** – add a routine to be called at every task delete
***taskDeleteHookDelete*( )** – delete a previously added task delete routine

DESCRIPTION

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

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

***taskSwitchHookAdd*( )** and ***taskSwitchHookDelete*( )**
Add and delete routines to be called when a task is switched.

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

This facility is used by **dbgLib** to provide task-specific breakpoints and single-stepping. It is used by **taskVarLib** for the "task variable" mechanism. It is also used by **fppLib** for floating-point coprocessor support.

NOTE

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

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

INCLUDE FILES

taskHookLib.h

SEE ALSO

dbgLib, **fppLib**, **taskLib**, **taskVarLib**, *VxWorks Programmer's Guide: Basic OS*

taskHookShow

NAME **taskHookShow** – task hook show routines

ROUTINES
taskHookShowInit( ) – initialize the task hook show facility
taskCreateHookShow( ) – show the list of task create routines
taskSwitchHookShow( ) – show the list of task switch routines
taskDeleteHookShow( ) – show the list of task delete routines

DESCRIPTION This library provides routines which summarize the installed kernel hook routines. There is one routine dedicated to the display of each type of kernel hook: task operation, task switch, and task deletion.

The routine ***taskHookShowInit( )*** links the task hook show facility into the VxWorks system. It is called automatically when this show facility is configured into VxWorks using either of the following methods:

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

INCLUDE FILES **taskHookLib.h**

SEE ALSO **taskHookLib**, *VxWorks Programmer's Guide: Basic OS*

taskInfo

NAME **taskInfo** – task information library

ROUTINES
taskOptionsSet( ) – change task options
taskOptionsGet( ) – examine task options
taskRegsGet( ) – get a task's registers from the TCB
taskRegsSet( ) – set a task's registers
taskName( ) – get the name associated with a task ID
taskNameToId( ) – look up the task ID associated with a task name
taskIdDefault( ) – set the default task ID
taskIsReady( ) – check if a task is ready to run
taskIsSuspended( ) – check if a task is suspended
taskIdListGet( ) – get a list of active task IDs

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

Task information is crucial as a debugging aid and user-interface convenience during the development cycle of an application. The routines ***taskOptionsGet( )***, ***taskRegsGet( )***,

***taskName*( )**, ***taskNameToId*( )**, ***taskIsReady*( )**, ***taskIsSuspended*( )**, and ***taskIdListGet*( )** are used to obtain task information. Three routines -- ***taskOptionsSet*( )**, ***taskRegsSet*( )**, and ***taskIdDefault*( )** -- provide programmatic access to debugging features.

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

Task management and control routines are provided by **taskLib**. Higher-level task information display routines are provided by taskShow.

INCLUDE FILES **taskLib.h**

SEE ALSO **taskLib**, **taskShow**, **taskHookLib**, **taskVarLib**, **semLib**, **kernelLib**, *VxWorks Programmer's Guide: Basic OS*

---

taskLib

NAME **taskLib** – task management library

ROUTINES

***taskSpawn*( )** – spawn a task
***taskInit*( )** – initialize a task with a stack at a specified address
***taskActivate*( )** – activate a task that has been initialized
***exit*( )** – exit a task (ANSI)
***taskDelete*( )** – delete a task
***taskDeleteForce*( )** – delete a task without restriction
***taskSuspend*( )** – suspend a task
***taskResume*( )** – resume a task
***taskRestart*( )** – restart a task
***taskPrioritySet*( )** – change the priority of a task
***taskPriorityGet*( )** – examine the priority of a task
***taskLock*( )** – disable task rescheduling
***taskUnlock*( )** – enable task rescheduling
***taskSafe*( )** – make the calling task safe from deletion
***taskUnsafe*( )** – make the calling task unsafe from deletion
***taskDelay*( )** – delay a task from executing
***taskIdSelf*( )** – get the task ID of a running task
***taskIdVerify*( )** – verify the existence of a task
***taskTcb*( )** – get the task control block for a task ID

DESCRIPTION This library provides the interface to the VxWorks task management facilities. Task control services are provided by the VxWorks kernel, which is comprised of **kernelLib**, **taskLib**, **semLib**, **tickLib**, **msgQLib**, and **wdLib**. Programmatic access to task information and debugging features is provided by **taskInfo**. Higher-level task information display routines are provided by **taskShow**.

TASK CREATION

Tasks are created with the general-purpose routine ***taskSpawn( )***. Task creation consists of the following: allocation of memory for the stack and task control block (**WIND_TCB**), initialization of the **WIND_TCB**, and activation of the **WIND_TCB**. Special needs may require the use of the lower-level routines ***taskInit( )*** and ***taskActivate( )***, which are the underlying primitives of ***taskSpawn( )***.

Tasks in VxWorks execute in the most privileged state of the underlying architecture. In a shared address space, processor privilege offers no protection advantages and actually hinders performance.

There is no limit to the number of tasks created in VxWorks, as long as sufficient memory is available to satisfy allocation requirements.

The routine ***sp( )*** is provided in **usrLib** as a convenient abbreviation for spawning tasks. It calls ***taskSpawn( )*** with default parameters.

TASK DELETION

If a task exits its "main" routine, specified during task creation, the kernel implicitly calls ***exit( )*** to delete the task. Tasks can be deleted with the ***taskDelete( )*** or ***exit( )*** routine.

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

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

The **sigLib** facility may also be used to allow a task to execute clean-up code before actually expiring.

TASK CONTROL

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

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

TASK SCHEDULING VxWorks schedules tasks on the basis of priority. Tasks may have priorities ranging from 0, the highest priority, to 255, the lowest priority. The priority of a task in VxWorks is dynamic, and an existing task's priority can be changed using ***taskPrioritySet*( )**.

INCLUDE FILES **taskLib.h**

SEE ALSO **taskInfo**, **taskShow**, **taskHookLib**, **taskVarLib**, **semLib**, **semMLib**, **kernelLib**, *VxWorks Programmer's Guide: Basic OS*

---

taskShow

NAME **taskShow** – task show routines

ROUTINES
***taskShowInit*( )** – initialize the task show routine facility
***taskInfoGet*( )** – get information about a task
***taskShow*( )** – display task information from TCBs
***taskRegsShow*( )** – display the contents of a task's registers
***taskStatusString*( )** – get a task's status as a string

DESCRIPTION This library provides routines to show task-related information, such as register values, task status, etc.

The ***taskShowInit*( )** routine links the task show facility into the VxWorks system. It is called automatically when this show facility is configured into VxWorks using either of the following methods:

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

Task information is crucial as a debugging aid and user-interface convenience during the development cycle of an application. The routines ***taskInfoGet*( )**, ***taskShow*( )**, ***taskRegsShow*( )**, and ***taskStatusString*( )** are used to display task information.

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

Task management and control routines are provided by **taskLib**. Programmatic access to task information and debugging features is provided by **taskInfo**.

INCLUDE FILES **taskLib.h**

SEE ALSO **taskLib**, **taskInfo**, **taskHookLib**, **taskVarLib**, **semLib**, **kernelLib**, *VxWorks Programmer's Guide: Basic OS, Target Shell, Tornado User's Guide: Shell*

taskVarLib

NAME

taskVarLib – task variables support library

ROUTINES

***taskVarInit*( )** – initialize the task variables facility
***taskVarAdd*( )** – add a task variable to a task
***taskVarDelete*( )** – remove a task variable from a task
***taskVarGet*( )** – get the value of a task variable
***taskVarSet*( )** – set the value of a task variable
***taskVarInfo*( )** – get a list of task variables of a task

DESCRIPTION

VxWorks provides a facility called "task variables," which allows 4-byte variables to be added to a task's context, and the variables' values to be switched each time a task switch occurs to or from the calling task. Typically, several tasks declare the same variable (4-byte memory location) as a task variable and treat that memory location as their own private variable. For example, this facility can be used when a routine must be spawned more than once as several simultaneous tasks.

The routines ***taskVarAdd*( )** and ***taskVarDelete*( )** are used to add or delete a task variable. The routines ***taskVarGet*( )** and ***taskVarSet*( )** are used to get or set the value of a task variable.

NOTE

If you are using task variables in a task delete hook (see **taskHookLib**), refer to the manual entry for ***taskVarInit*( )** for warnings on proper usage.

INCLUDE FILES

taskVarLib.h

SEE ALSO

taskHookLib, *VxWorks Programmer's Guide: Basic OS*

tcic

NAME

tcic – Databook TCIC/2 PCMCIA host bus adaptor chip driver

ROUTINES

***tcicInit*( )** – initialize the TCIC chip

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

ROUTINES ***tcicShow*( )** – show all configurations of the TCIC chip

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*( )**.

tcpShow

NAME **tcpShow** – TCP information display routines

ROUTINES ***tcpShowInit*( )** – initialize TCP show routines
***tcpDebugShow*( )** – display debugging information for the TCP protocol
***tcpstatShow*( )** – display all statistics for the TCP protocol

DESCRIPTION This library provides routines to show TCP related statistics.

Interpreting these statistics requires detailed knowledge of Internet network protocols. Information on these protocols can be found in the following books:

– *TCP/IP Illustrated Volume II, The Implementationa,* by Richard Stevens

– *The Design and Implementation of the 4.4 BSD UNIX Operating System,* by Leffler, McKusick, Karels and Quarterman

The ***tcpShowInit*( )** routine links the TCP show facility into the VxWorks system. This is performed automatically if **INCLUDE_NET_SHOW** is defined in **configAll.h**.

SEE ALSO **netLib**, **netShow**, *VxWorks Programmer's Guide: Network*

telnetLib

NAME

telnetLib – telnet server library

ROUTINES

***telnetInit*( )** – initialize the telnet daemon
***telnetd*( )** – VxWorks telnet daemon

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 the configuration macro **INCLUDE_TELNET** is defined.

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

ROUTINES

***tftpdInit*( )** – initialize the TFTP server task
***tftpdTask*( )** – TFTP server daemon task
***tftpdDirectoryAdd*( )** – add a directory to the access list
***tftpdDirectoryRemove*( )** – delete a directory from the access list

DESCRIPTION

This library implements the VxWorks Trivial File Transfer Protocol (TFTP) server module. The server can respond to both read and write requests. It is started by a call to ***tftpdInit*( )**.

The server has access to a list of directories that can either be provided in the initial call to ***tftpdInit*( )** or changed dynamically using the ***tftpdDirectoryAdd*( )** and ***tftpDirectoryRemove*( )** calls. Requests for files not in the directory trees specified in the access list will be rejected, unless the list is empty, in which case all requests will be allowed. By default, the access list contains the directory given in the global variable **tftpdDirectory**. It is possible to remove the default by calling ***tftpdDirectoryRemove*( )**.

For specific information about the TFTP protocol, see RFC 783, "TFTP Protocol."

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

ROUTINES

***tftpXfer*( )** – transfer a file via TFTP using a stream interface
***tftpCopy*( )** – transfer a file via TFTP
***tftpInit*( )** – initialize a TFTP session
***tftpModeSet*( )** – set the TFTP transfer mode
***tftpPeerSet*( )** – set the TFTP server address
***tftpPut*( )** – put a file to a remote system
***tftpGet*( )** – get a file from a remote system
***tftpInfoShow*( )** – get TFTP status information
***tftpQuit*( )** – quit a TFTP session
***tftpSend*( )** – send a TFTP message to the remote system

DESCRIPTION

This library implements the VxWorks Trivial File Transfer Protocol (TFTP) client library. TFTP is a simple file transfer protocol (hence the name "trivial") implemented over UDP. TFTP was designed to be small and easy to implement; therefore it is limited in functionality in comparison with other file transfer protocols, such as FTP. TFTP provides only the read/write capability to and from a remote server.

TFTP provides no user authentication; therefore the remote files must have "loose" permissions before requests for file access will be granted by the remote TFTP server (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

ROUTINES
***tickAnnounce*( )** – announce a clock tick to the kernel
***tickSet*( )** – set the value of the kernel's tick counter
***tickGet*( )** – get the value of the kernel's tick counter

DESCRIPTION This library is the interface to the VxWorks kernel routines that announce a clock tick to the kernel, get the current time in ticks, and set the current time in ticks.

Kernel facilities that rely on clock ticks include ***taskDelay*( )**, ***wdStart*( )**, ***kernelTimeslice*( )**, and semaphore timeouts. In each case, the specified timeout is relative to the current time, also referred to as "time to fire." Relative timeouts are not affected by calls to ***tickSet*( )**, which only changes absolute time. The routines ***tickSet*( )** and ***tickGet*( )** keep track of absolute time in isolation from the rest of the kernel.

Time-of-day clocks or other auxiliary time bases are preferable for lengthy timeouts of days or more. The accuracy of such time bases is greater, and some external time bases even calibrate themselves periodically.

INCLUDE FILES **tickLib.h**

SEE ALSO **kernelLib**, **taskLib**, **semLib**, **wdLib**, *VxWorks Programmer's Guide: Basic OS*

timerLib

NAME **timerLib** – timer library (POSIX)

ROUTINES
***timer_cancel*( )** – cancel a timer
***timer_connect*( )** – connect a user routine to the timer signal
***timer_create*( )** – allocate a timer using the specified clock for a timing base (POSIX)
***timer_delete*( )** – remove a previously created timer (POSIX)
***timer_gettime*( )** – get the remaining time before expiration and the reload value (POSIX)
***timer_getoverrun*( )** – return the timer expiration overrun (POSIX)

***timer_settime*( )** – set the time until the next expiration and arm timer (POSIX)
***nanosleep*( )** – suspend the current task until the time interval elapses (POSIX)

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

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

The **clockLib** library should be installed and ***clock_settime*( )** set before the use of any timer routines.

ADDITIONS Two non-POSIX functions are provided for user convenience:

***timer_cancel*( )** quickly disables a timer by calling ***timer_settime*( )**.
***timer_connect*( )** easily hooks up a user routine by calling ***sigaction*( )**.

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

When a timer expires and the task has previously exited, ***logMsg*( )** indicates the expected task is not present. Similarly, ***logMsg*( )** indicates when a task arms a timer without installing a signal handler. Timers may be armed but not created or deleted at interrupt level.

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

INCLUDE FILES **timers.h**

SEE ALSO **clockLib**, ***sigaction*( )**, POSIX 1003.1b documentation, *VxWorks Programmer's Guide: Basic OS*

---

timexLib

NAME **timexLib** – execution timer facilities

ROUTINES ***timexInit*( )** – include the execution timer library
***timexClear*( )** – clear the list of function calls to be timed
***timexFunc*( )** – specify functions to be timed
***timexHelp*( )** – display synopsis of execution timer facilities

***timex*()** – time a single execution of a function or functions
***timexN*()** – time repeated executions of a function or group of functions
***timexPost*()** – specify functions to be called after timing
***timexPre*()** – specify functions to be called prior to timing
***timexShow*()** – display the list of function calls to be timed

DESCRIPTION

This library contains routines for timing the execution of programs, individual functions, and groups of functions. The VxWorks system clock is used as a time base. Functions that have a short execution time relative to this time base can be called repeatedly to establish an average execution time with an acceptable percentage of error.

Up to four functions can be specified to be timed as a group. Additionally, sets of up to four functions can be specified as pre- or post-timing functions, to be executed before and after the timed functions. The routines ***timexPre*()** and ***timexPost*()** are used to specify the pre- and post-timing functions, while ***timexFunc*()** specifies the functions to be timed.

The routine ***timex*()** is used to time a single execution of a function or group of functions. If called with no arguments, ***timex*()** uses the functions in the lists created by calls to ***timexPre*()**, ***timexPost*()**, and ***timexFunc*()**. If called with arguments, ***timex*()** times the function specified, instead of the previous list. The routine ***timexN*()** works in the same manner as ***timex*()** except that it iterates the function calls to be timed.

EXAMPLES

The routine ***timex*()** can be used to obtain the execution time of a single routine:

```
-> timex myFunc, myArg1, myArg2, ...
```

The routine ***timexN*()** calls a function repeatedly until a 2% or better tolerance is obtained:

```
-> timexN myFunc, myArg1, myArg2, ...
```

The routines ***timexPre*()**, ***timexPost*()**, and ***timexFunc*()** are used to specify a list of functions to be executed as a group:

```
-> timexPre 0, myPreFunc1, preArg1, preArg2, ...
-> timexPre 1, myPreFunc2, preArg1, preArg2, ...
-> timexFunc 0, myFunc1, myArg1, myArg2, ...
-> timexFunc 1, myFunc2, myArg1, myArg2, ...
-> timexFunc 2, myFunc3, myArg1, myArg2, ...
-> timexPost 0, myPostFunc, postArg1, postArg2, ...
```

The list is executed by calling ***timex*()** or ***timexN*()** without arguments:

```
-> timex
```

or

```
-> timexN
```

In this example, *myPreFunc1* and *myPreFunc2* are called with their respective arguments. *myFunc1*, *myFunc2*, and *myFunc3* are then called in sequence and timed. If ***timexN*()** was used, the sequence is called repeatedly until a 2% or better error tolerance is achieved.

Finally, *myPostFunc* is called with its arguments. The timing results are reported after all post-timing functions are called.

NOTE The timings measure the execution time of the routine body, without the usual subroutine entry and exit code (usually LINK, UNLINK, and RTS instructions). Also, the time required to set up the arguments and call the routines is not included in the reported times. This is because these timing routines automatically calibrate themselves by timing the invocation of a null routine, and thereafter subtracting that constant overhead.

INCLUDE FILES **timexLib.h**

SEE ALSO **spyLib**

ttyDrv

NAME **ttyDrv** – provide terminal device access to serial channels

ROUTINES
***ttyDrv*( )** – initialize the tty driver
***ttyDevCreate*( )** – create a VxWorks device for a serial channel

DESCRIPTION This library provides the OS-dependent functionality of a serial device, including canonical processing and the interface to the VxWorks I/O system.

The BSP provides "raw" serial channels which are accessed via an **SIO_CHAN** data structure. These raw devices provide only low level access to the devices to send and receive characters. This library builds on that functionality by allowing the serial channels to be accessed via the VxWorks I/O system using the standard read/write interface. It also provides the canonical processing support of **tyLib**.

The routines in this library are typically called by ***usrRoot*( )** in **usrConfig.c** to create VxWorks serial devices at system startup time.

INCLUDE FILES **ttyLib.h**

SEE ALSO **tyLib**, **sioLib.h**

tyLib

NAME

tyLib – tty driver support library

ROUTINES

***tyDevInit*( )** – initialize the tty device descriptor
***tyAbortFuncSet*( )** – set the abort function
***tyAbortSet*( )** – change the abort character
***tyBackspaceSet*( )** – change the backspace character
***tyDeleteLineSet*( )** – change the line-delete character
***tyEOFSet*( )** – change the end-of-file character
***tyMonitorTrapSet*( )** – change the trap-to-monitor character
***tyIoctl*( )** – handle device control requests
***tyWrite*( )** – do a task-level write for a tty device
***tyRead*( )** – do a task-level read for a tty device
***tyITx*( )** – interrupt-level output
***tyIRd*( )** – interrupt-level input

DESCRIPTION

This library provides routines used to implement drivers for serial devices. It provides all the necessary device-independent functions of a normal serial channel, including:

– ring buffering of input and output
– raw mode
– optional line mode with backspace and line-delete functions
– optional processing of X-on/X-off
– optional RETURN/LINEFEED conversion
– optional echoing of input characters
– optional stripping of the parity bit from 8-bit input
– optional special characters for shell abort and system restart

Most of the routines in this library are called only by device drivers. Functions that normally might be called by an application or interactive user are the routines to set special characters, ***ty...Set*( )**.

USE IN SERIAL DEVICE DRIVERS

Each device that uses **tyLib** is described by a data structure of type **TY_DEV**. This structure begins with an I/O system device header so that it can be added directly to the I/O system's device list. A driver calls ***tyDevInit*( )** to initialize a **TY_DEV** structure for a specific device and then calls ***iosDevAdd*( )** to add the device to the I/O system.

The call to ***tyDevInit*( )** takes three parameters: the pointer to the **TY_DEV** structure to initialize, the desired size of the read and write ring buffers, and the address of a transmitter start-up routine. This routine will be called when characters are added for output and the transmitter is idle. Thereafter, the driver can call the following routines to perform the usual device functions:

***tyRead*( )**
: user read request to get characters that have been input

***tyWrite*( )**
: user write request to put characters to be output

***tyIoctl*( )**
: user I/O control request

***tyIRd*( )**
: interrupt-level routine to get an input character

***tyITx*( )**
: interrupt-level routine to deliver the next output character

Thus, ***tyRead*( )**, ***tyWrite*( )**, and ***tyIoctl*( )** are called from the driver's read, write, and I/O control functions. The routines ***tyIRd*( )** and ***tyITx*( )** are called from the driver's interrupt handler in response to receive and transmit interrupts, respectively.

Examples of using **tyLib** in a driver can be found in the source file(s) included by tyCoDrv. Source files are located in src/drv/serial.

TTY OPTIONS

A full range of options affects the behavior of tty devices. These options are selected by setting bits in the device option word using the **FIOSETOPTIONS** function in the ***ioctl*( )** routine (see "I/O Control Functions" below for more information). The following is a list of available options. The options are defined in the header file **ioLib.h**.

OPT_LINE
: Selects line mode. A tty device operates in one of two modes: raw mode (unbuffered) or line mode. Raw mode is the default. In raw mode, each byte of input from the device is immediately available to readers, and the input is not modified except as directed by other options below. In line mode, input from the device is not available to readers until a NEWLINE character is received, and the input may be modified by backspace, line-delete, and end-of-file special characters.

OPT_ECHO
: Causes all input characters to be echoed to the output of the same channel. This is done simply by putting incoming characters in the output ring as well as the input ring. If the output ring is full, the echoing is lost without affecting the input.

OPT_CRMOD
: C language conventions use the NEWLINE character as the line terminator on both input and output. Most terminals, however, supply a RETURN character when the return key is hit, and require both a RETURN and a LINEFEED character to advance the output line. This option enables the appropriate translation: NEWLINEs are substituted for input RETURN characters, and NEWLINEs in the output file are automatically turned into a RETURN-LINEFEED sequence.

OPT_TANDEM
: Causes the driver to generate and respond to the special flow control characters

CTRL-Q and CTRL-S in what is commonly known as X-on/X-off protocol. Receipt of a CTRL-S input character will suspend output to that channel. Subsequent receipt of a CTRL-Q will resume the output. Also, when the VxWorks input buffer is almost full, a CTRL-S will be output to signal the other side to suspend transmission. When the input buffer is almost empty, a CTRL-Q will be output to signal the other side to resume transmission.

OPT_7_BIT
Strips the most significant bit from all bytes input from the device.

OPT_MON_TRAP
Enables the special monitor trap character, by default CTRL-X. When this character is received and this option is enabled, VxWorks will trap to the ROM resident monitor program. Note that this is quite drastic. All normal VxWorks functioning is suspended, and the computer system is entirely controlled by the monitor. Depending on the particular monitor, it may or may not be possible to restart VxWorks from the point of interruption. The default monitor trap character can be changed by calling ***tyMonitorTrapSet( )***.

OPT_ABORT
Enables the special shell abort character, by default CTRL-C. When this character is received and this option is enabled, the VxWorks shell is restarted. This is useful for freeing a shell stuck in an unfriendly routine, such as one caught in an infinite loop or one that has taken an unavailable semaphore. For more information, see the *VxWorks Programmer's Guide: Shell.*

OPT_TERMINAL
This is not a separate option bit. It is the value of the option word with all the above bits set.

OPT_RAW
This is not a separate option bit. It is the value of the option word with none of the above bits set.

I/O CONTROL FUNCTIONS

The tty devices respond to the following ***ioctl( )*** functions. The functions are defined in the header **ioLib.h**.

FIOGETNAME
Gets the file name of the file descriptor and copies it to the buffer referenced to by *nameBuf*:

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

This function is common to all file descriptors for all devices.

FIOSETOPTIONS, FIOOPTIONS
Sets the device option word to the specified argument. For example, the call:

```
status = ioctl (fd, FIOOPTIONS, OPT_TERMINAL);
status = ioctl (fd, FIOSETOPTIONS, OPT_TERMINAL);
```

enables all the tty options described above, putting the device in a "normal" terminal mode. If the line protocol (**OPT_LINE**) is changed, the input buffer is flushed. The various options are described in **ioLib.h**.

FIOGETOPTIONS
Returns the current device option word:

```
options = ioctl (fd, FIOGETOPTIONS, 0);
```

FIONREAD
Copies to *nBytesUnread* the number of bytes available to be read in the device's input buffer:

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

In line mode (**OPT_LINE** set), the **FIONREAD** function actually returns the number of characters available plus the number of lines in the buffer. Thus, if five lines of just NEWLINEs were in the input buffer, it would return the value 10 (5 characters + 5 lines).

FIONWRITE
Copies to *nBytes* the number of bytes queued to be output in the device's output buffer:

```
status = ioctl (fd, FIONWRITE, &nBytes);
```

FIOFLUSH
Discards all the bytes currently in both the input and the output buffers:

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

FIOWFLUSH
Discards all the bytes currently in the output buffer:

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

FIORFLUSH
Discards all the bytes currently in the input buffers:

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

FIOCANCEL
Cancels a read or write. A task blocked on a read or write may be released by a second task using this ***ioctl*( )** call. For example, a task doing a read can set a watchdog timer before attempting the read; the auxiliary task would wait on a semaphore. The watchdog routine can give the semaphore to the auxiliary task, which would then use the following call on the appropriate file descriptor:

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

FIOBAUDRATE
Sets the baud rate of the device to the specified argument. For example, the call:

```
status = ioctl (fd, FIOBAUDRATE, 9600);
```

Sets the device to operate at 9600 baud. This request has no meaning on a pseudo terminal.

FIOISATTY
Returns TRUE for a tty device:

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

FIOPROTOHOOK
Adds a protocol hook function to be called for each input character. *pfunction* is a pointer to the protocol hook routine which takes two arguments of type *int* and returns values of type STATUS (TRUE or FALSE). The first argument passed is set by the user via the **FIOPROTOARG** function. The second argument is the input character. If no further processing of the character is required by the calling routine (the input routine of the driver), the protocol hook routine *pFunction* should return TRUE. Otherwise, it should return FALSE:

```
status = ioctl (fd, FIOPROTOHOOK, pFunction);
```

FIOPROTOARG
Sets the first argument to be passed to the protocol hook routine set by **FIOPROTOHOOK** function:

```
status = ioctl (fd, FIOPROTOARG, arg);
```

FIORBUFSET
Changes the size of the receive-side buffer to *size*:

```
status = ioctl (fd, FIORBUFSET, size);
```

FIOWBUFSET
Changes the size of the send-side buffer to *size*:

```
status = ioctl (fd, FIOWBUFSET, size);
```

Any other ***ioctl*( )** request will return an error and set the status to **S_ioLib_UNKNOWN_REQUEST**.

INCLUDE FILES **tyLib.h**, **ioLib.h**

SEE ALSO **ioLib**, **iosLib**, **tyCoDrv**, *VxWorks Programmer's Guide: I/O System*

---

udpShow

NAME **udpShow** – UDP information display routines

ROUTINES ***udpShowInit*( )** – initialize UDP show routines
***udpstatShow*( )** – display statistics for the UDP protocol

DESCRIPTION This library provides routines to show UDP related statistics.

Interpreting these statistics requires detailed knowledge of Internet network protocols. Information on these protocols can be found in the following books:

– *TCP/IP Illustrated Volume II, The Implementation,* by Richard Stevens

– *The Design and Implementation of the 4.4 BSD UNIX Operating System,* by Leffler, McKusick, Karels and Quarterman

The ***udpShowInit*( )** routine links the UDP show facility into the VxWorks system. This is performed automatically if **INCLUDE_NET_SHOW** is defined in **configAll.h**.

SEE ALSO **udpShow**, **netLib**, **netShow**, *VxWorks Programmer's Guide: Network*

ultraEnd

NAME **ultraEnd** – SMC Ultra Elite END network interface driver

ROUTINES
***ultraLoad*( )** – initialize the driver and device
***ultraParse*( )** – parse the init string
***ultraMemInit*( )** – initialize memory for the chip
***ultraAddrFilterSet*( )** – set the address filter for multicast addresses

DESCRIPTION This module implements the SMC Elite Ultra Ethernt 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 only masked during configuration or in polled mode.

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 SUPPORT REQUIREMENTS

This driver requires several external support functions, defined as macros:

```
SYS_INT_CONNECT(pDrvCtrl, routine, arg)
SYS_INT_DISCONNECT (pDrvCtrl, routine, arg)
SYS_INT_ENABLE(pDrvCtrl)
SYS_INT_DISABLE(pDrvCtrl)
SYS_IN_BYTE(pDrvCtrl, reg, pData)
SYS_OUT_BYTE(pDrvCtrl, reg, pData)
```

These macros allow the driver to be customized for BSPs that use special versions of these routines.

The macro **SYS_INT_CONNECT** is used to connect the interrupt handler to the appropriate vector. By default it is the routine ***intConnect*( )**.

The macro **SYS_INT_DISCONNECT** is used to disconnect the interrupt handler prior to unloading the module. By default this is a dummy routine that returns OK.

The macro **SYS_INT_ENABLE** is used to enable the interrupt level for the end device. It is called once during initialization. It calls an external board level routine ***sysUltraIntEnable*( )**.

The macro **SYS_INT_DISABLE** is used to disable the interrupt level for the end device. It is called once during shutdown. It calls an external board level routine ***sysUltraIntDisable*( )**.

The macros **SYS_IN_BYTE** and **SYS_OUT_BYTE** are used for accessing the ultra device. The default macros map these operations onto ***sysInByte*( )** and ***sysOutByte*( )**.

INCLUDES **end.h endLib.h etherMultiLib.h**

SEE ALSO **ultraEnd**, **muxLib**, **endLib**, *Writing an Enhanced Network Driver*

---

unixDrv

NAME **unixDrv** – UNIX-file disk driver (VxSim for Solaris and VxSim for HP)

ROUTINES
***unixDrv*( )** – install UNIX disk driver
***unixDiskDevCreate*( )** – create a UNIX disk device
***unixDiskInit*( )** – initialize a dosFs disk on top of UNIX

DESCRIPTION This driver emulates a VxWorks disk driver, but actually uses the UNIX file system to store the data. The VxWorks disk appears under UNIX as a single file. The UNIX file name, and the size of the disk, may be specified during the ***unixDiskDevCreate*( )** call.

USER-CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. The routine ***unixDrv*( )** must be called to initialize the driver and the ***unixDiskDevCreate*( )** routine is used to create devices.

CREATING UNIX DISKS

Before a UNIX disk can be used, it must be created. This is done with the ***unixDiskDevCreate*( )** call. The format of this call is:

```
BLK_DEV *unixDiskDevCreate
    (
    char    *unixFile,      /* name of the UNIX file to use         */
    int     bytesPerBlk,    /* number of bytes per block            */
    int     blksPerTrack,   /* number of blocks per track           */
    int     nBlocks         /* number of blocks on this device      */
    )
```

The UNIX file must be pre-allocated separately. This can be done using the UNIX mkfile(8) command. Note that you have to create an appropriately sized file. For example, to create a UNIX file system that is used as a common floppy dosFs file system, you would issue the 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

unixDrv, ***dosFsDevInit*( )**, ***dosFsMkfs*( )**, ***rt11FsDevInit*( )**, ***rt11FsMkfs*( )**, ***rawFsDevInit*( )**, *VxWorks Programmer's Guide: I/O System, Local File Systems*

---

unixSio

NAME

unixSio – unix serial driver

ROUTINES

***unixDevInit*( )** – initialize a **UNIX_DUSART**
***unixDevInit2*( )** – enable interrupts
***unixIntRcv*( )** – handle a channel's receive-character interrupt.
***dummyCallback*( )** – dummy callback routine.

DESCRIPTION

This is the driver for the UNIX stdin/stdio-base simulated serial port.

USAGE

A **UNIX_CHAN** structure is used to describe each channel available.

The BSP's ***sysHwInit*( )** routine typically calls **sysSerial.c:*sysSerialHwInit*( )**, which opens UNIX tty/pty devices for serial lines and initializes the **UNIX_CHAN u_fd** and **u_pid** fields before calling ***unixDevInit*( )**.

The BSP ***sysSerialHwInit2*( )** calls ***unixDevInit2*( )** to enable interrupts.

```
#include "drv/sio/unixSio.h"
UNIX_CHAN myChan [NUM_TTY];
```

```
SIO_CHAN * sysSioChans[NUM_TTY];
sysSerialHwInit (void)
    {
    ...
    for (ix = 0; ix < NUM_TTY; ix++)
        {
        if (ix > 0)     // dev 0 is unix sdtin/out/err //
            {
            UNIX_CHAN * pChan = &myChan[ix];
            sysSioChans[ix] = (SIO_CHAN *) pChan;
            pChan->u_fd = ptyXtermOpen (ptyName, &pChan->u_pid, 0);
            }
        ...
        unixDevInit (&myChan);
        }
    }
sysSerialHwInit2 (void)
    {
    ...
    for (i = 0; i < NUM_TTY; i++)
        intConnect (FD_TO_IVEC(myChan[i]->u_fd), unixInt, (int)&myChan[i]);
    ...
    }
```

INCLUDE FILES **drv/sio/unixSio.h sioLib.h**

unldLib

NAME **unldLib** – object module unloading library

ROUTINES
***unld*( )** – unload an object module by specifying a file name or module ID
***unldByModuleId*( )** – unload an object module by specifying a module ID
***unldByNameAndPath*( )** – unload an object module by specifying a name and path
***unldByGroup*( )** – unload an object module by specifying a group number
***reld*( )** – reload an object module

DESCRIPTION This library provides a facility for unloading object modules. Once an object module has been loaded into the system (using the facilities provided by **loadLib**), it can be removed from the system by calling one of the ***unld...*( )** routines in this library.

Unloading of an object module does the following:

(1) It frees the space allocated for text, data, and BSS segments, unless ***loadModuleAt*( )**

was called with specific addresses, in which case the user is responsible for freeing the space.

(2) It removes all symbols associated with the object module from the system symbol table.

(3) It removes the module descriptor from the module list.

Once the module is unloaded, any calls to routines in that module from other modules will fail unpredictably. The user is responsible for ensuring that no modules are unloaded that are used by other modules. ***unld*( )** checks the hooks created by the following routines to ensure none of the unloaded code is in use by a hook:

***taskCreateHookAdd*( )**
***taskDeleteHookAdd*( )**
***taskHookAdd*( )**
***taskSwapHookAdd*( )**
***taskSwitchHookAdd*( )**

However, ***unld*( )** *does not* check the hooks created by these routines:

***etherInputHookAdd*( )**
***etherOutputHookAdd*( )**
***excHookAdd*( )**
***rebootHookAdd*( )**
***moduleCreateHookAdd*( )**

INCLUDE FILES **unldLib.h**, **moduleLib.h**

SEE ALSO **loadLib**, **moduleLib**

---

usrAta

NAME **usrAta** – ATA initialization

ROUTINES ***usrAtaConfig*( )** – mount a DOS file system from an ATA hard disk
***usrAtaPartition*( )** – get an offset to the first partition of the drive

usrConfig

NAME

usrConfig – user-defined system configuration library

ROUTINES

***usrInit*( )** – user-defined system initialization routine
***usrRoot*( )** – the root task
***usrClock*( )** – user-defined system clock interrupt routine

DESCRIPTION

This library is the WRS-supplied configuration module for VxWorks. It contains the root task, the primary system initialization routine, the network initialization routine, and the clock interrupt routine.

The include file **config.h** includes a number of system-dependent parameters used in this file.

In an effort to simplify the presentation of the configuration of vxWorks, this file has been split into smaller files. These additional configuration source files are located in ../../src/config/usr[xxx].c and are #included into this file below. This file contains the bulk of the code a customer is likely to customize.

The module **usrDepend.c** contains checks that guard against unsupported configurations 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

VxWorks Programmer's Guide: Configuration & Build

usrFd

NAME

usrFd – floppy disk initialization

ROUTINES

***usrFdConfig*( )** – mount a DOS file system from a floppy disk

usrIde

NAME **usrIde** – IDE initialization

ROUTINES ***usrIdeConfig*( )** – mount a DOS file system from an IDE hard disk

usrLib

NAME **usrLib** – user interface subroutine library

ROUTINES
***help*( )** – print a synopsis of selected routines
***netHelp*( )** – print a synopsis of network routines
***bootChange*( )** – change the boot line
***periodRun*( )** – call a function periodically
***period*( )** – spawn a task to call a function periodically
***repeatRun*( )** – call a function repeatedly
***repeat*( )** – spawn a task to call a function repeatedly
***sp*( )** – spawn a task with default parameters
***checkStack*( )** – print a summary of each task's stack usage
***i*( )** – print a summary of each task's TCB
***ti*( )** – print complete information from a task's TCB
***show*( )** – print information on a specified object
***ts*( )** – suspend a task
***tr*( )** – resume a task
***td*( )** – delete a task
***version*( )** – print VxWorks version information
***m*( )** – modify memory
***d*( )** – display memory
***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

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.

NOTE

This library uses a small number of undocumented VxWorks internal routines.

INCLUDE FILES

usrLib.h

SEE ALSO

spyLib, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

usrScsi

NAME **usrScsi** – SCSI initialization

ROUTINES ***usrScsiConfig*( )** – configure SCSI peripherals

usrSmObj

NAME **usrSmObj** – shared memory object initialization

ROUTINES ***usrSmObjInit*( )** – initialize shared memory objects

vmBaseLib

NAME **vmBaseLib** – base virtual memory support library

ROUTINES

***vmBaseLibInit*( )** – initialize base virtual memory support
***vmBaseGlobalMapInit*( )** – initialize global mapping
***vmBaseStateSet*( )** – change the state of a block of virtual memory
***vmBasePageSizeGet*( )** – return the page size

DESCRIPTION

This library provides the minimal MMU (Memory Management Unit) support needed in a system. Its primary purpose is to create cache-safe buffers for **cacheLib**. Buffers are provided to optimize I/O throughput.

A call to ***vmBaseLibInit*( )** initializes this library, thus permitting ***vmBaseGlobalMapInit*( )** to initialize the MMU and set up MMU translation tables. Additionally, ***vmBaseStateSet*( )** can be called to change the translation tables dynamically.

This library is a release-bundled complement to **vmLib** and **vmShow**, modules that offer full-featured MMU support and virtual memory information display routines. The **vmLib** and **vmShow** libraries are distributed as the unbundled virtual memory support option, VxVMI.

CONFIGURATION

Bundled MMU support is included in VxWorks when the configuration macro **INCLUDE_MMU_BASIC** is defined. If the configuration macro **INCLUDE_MMU_FULL** is also defined, the default is full MMU support (unbundled).

INCLUDE FILES **sysLib.h**, **vmLib.h**

SEE ALSO **vmLib**, **vmShow**, *VxWorks Programmer's Guide: Virtual Memory*

vmLib

NAME **vmLib** – architecture-independent virtual memory support library (VxVMI Opt.)

ROUTINES
***vmLibInit*( )** – initialize the virtual memory support module (VxVMI Opt.)
***vmGlobalMapInit*( )** – initialize global mapping (VxVMI Opt.)
***vmContextCreate*( )** – create a new virtual memory context (VxVMI Opt.)
***vmContextDelete*( )** – delete a virtual memory context (VxVMI Opt.)
***vmStateSet*( )** – change the state of a block of virtual memory (VxVMI Opt.)
***vmStateGet*( )** – get the state of a page of virtual memory (VxVMI Opt.)
***vmMap*( )** – map physical space into virtual space (VxVMI Opt.)
***vmGlobalMap*( )** – map physical pages to virtual space in shared global virtual memory (VxVMI Opt.)
***vmGlobalInfoGet*( )** – get global virtual memory information (VxVMI Opt.)
***vmPageBlockSizeGet*( )** – get the architecture-dependent page block size (VxVMI Opt.)
***vmTranslate*( )** – translate a virtual address to a physical address (VxVMI Opt.)
***vmPageSizeGet*( )** – return the page size (VxVMI Opt.)
***vmCurrentGet*( )** – get the current virtual memory context (VxVMI Opt.)
***vmCurrentSet*( )** – set the current virtual memory context (VxVMI Opt.)
***vmEnable*( )** – enable or disable virtual memory (VxVMI Opt.)
***vmTextProtect*( )** – write-protect a text segment (VxVMI Opt.)

DESCRIPTION This library provides an architecture-independent interface to the CPU's memory management unit (MMU). Although **vmLib** is implemented with architecture-specific libraries, application code need never reference directly the architecture-dependent code in these libraries.

A fundamental goal in the design of **vmLib** was to permit transparent backward compatibility with previous versions of VxWorks that did not use the MMU. System designers may opt to disable the MMU because of timing constraints, and some architectures do not support MMUs; therefore VxWorks functionality must not be dependent on the MMU. The resulting design permits a transparent configuration with no change in the programming environment (but the addition of several protection features, such as text segment protection) and the ability to disable virtual memory in systems that require it.

The **vmLib** library provides a mechanism for creating virtual memory contexts, ***vmContextCreate*( )**. These contexts are not automatically created for individual tasks, but may be created dynamically by tasks, and swapped in and out in an application specific manner.

All virtual memory contexts share a global transparent mapping of virtual to physical memory for all of local memory and the local hardware device space (defined in **sysLib.c** for each board port in the **sysPhysMemDesc**data structure). When the system is initialized, all of local physical memory is accessible at the same address in virtual memory (this is done with calls to ***vmGlobalMap*( )**.) Modifications made to this global mapping in one virtual memory context appear in all virtual memory contexts. For example, if the exception vector table (which resides at address 0 in physical memory) is made read only by calling ***vmStateSet*( )** on virtual address 0, the vector table will be read only in all virtual memory contexts.

Private virtual memory can also be created. When physical pages are mapped to virtual memory that is not in the global transparent region, this memory becomes accessible only in the context in which it was mapped. (The physical pages will also be accessible in the transparent translation at the physical address, unless the virtual pages in the global transparent translation region are explicitly invalidated.) State changes (writability, validity, etc.) to a section of private virtual memory in a virtual memory context do not appear in other contexts. To facilitate the allocation of regions of virtual space, ***vmGlobalInfoGet*( )** returns a pointer to an array of booleans describing which portions of the virtual address space are devoted to global memory. Each successive array element corresponds to contiguous regions of virtual memory the size of which is architecture-dependent and which may be obtained with a call to ***vmPageBlockSizeGet*( )**. If the boolean array element is true, the corresponding region of virtual memory, a "page block", is reserved for global virtual memory and should not be used for private virtual memory. (If ***vmMap*( )** is called to map virtual memory previously defined as global, the routine will return an error.)

All the state information for a block of virtual memory can be set in a single call to ***vmStateSet*( )**. It performs parameter checking and checks the validity of the specified virtual memory context. It may also be used to set architecture-dependent state information. See **vmLib.h** for additional architecture-dependent state information.

The routine ***vmContextShow*( )** in **vmShow** displays the virtual memory context for a specified context. For more information, see the manual entry for this routine.

CONFIGURATION Full MMU support (**vmLib**, and optionally, **vmShow**) is included in VxWorks when the configuration macro **INCLUDE_MMU_FULL** is defined. If the configuration macro **INCLUDE_MMU_BASIC** is also defined, the default is full MMU support (unbundled).

The **sysLib.c** library contains a data structure called **sysPhysMemDesc**, which is an array of **PHYS_MEM_DESC** structures. Each element of the array describes a contiguous section of physical memory. The description of this memory includes its physical address, the virtual address where it should be mapped (typically, this is the same as the physical address, but not necessarily so), an initial state for the memory, and a mask defining which state bits in the state value are to be set. Default configurations are defined for each board support package (BSP), but these mappings may be changed to suit user-specific system configurations. For example, the user may need to map additional VME space where the backplane network interface data structures appear.

AVAILABILITY

This library and **vmShow** are distributed as the unbundled virtual memory support option, VxVMI. A scaled down version, **vmBaseLib**, is provided with VxWorks for systems that do not permit optional use of the MMU, or for architectures that require certain features of the MMU to perform optimally (in particular, architectures that rely heavily on caching, but do not support bus snooping, and thus require the ability to mark 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.)

ROUTINES

***vmShowInit*( )** – include virtual memory show facility (VxVMI Opt.)
***vmContextShow*( )** – display the translation table for a context (VxVMI Opt.)

DESCRIPTION

This library contains virtual memory information display routines.

The routine ***vmShowInit*( )** links this facility into the VxWorks system. It is called automatically when this facility is configured into VxWorks using either of the following methods:

– If you use the configuration header files, define both **INCLUDE_MMU_FULL** and **INCLUDE_SHOW_ROUTINES** in **config.h**.

– If you use the Tornado project facility, select **INCLUDE_MMU_FULL_SHOW**.

AVAILABILITY

This module and **vmLib** are distributed as the unbundled virtual memory support option, VxVMI.

INCLUDE FILES

vmLib.h

SEE ALSO

vmLib, *VxWorks Programmer's Guide: Virtual Memory*

vxLib

NAME **vxLib** – miscellaneous support routines

ROUTINES
***vxTas*( )** – C-callable atomic test-and-set primitive
***vxMemArchProbe*( )** – architecture specific part of vxMemProbe
***vxMemProbe*( )** – probe an address for a bus error
***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)

DESCRIPTION This module contains miscellaneous VxWorks support routines.

INCLUDE FILES **vxLib.h**

VXWList

NAME **VXWList** – simple linked list class (WFC Opt.)

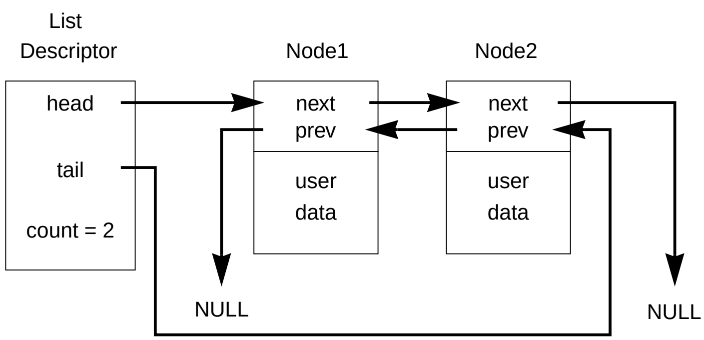
METHODS
***VXWList::VXWList*( )** – initialize a list
***VXWList::VXWList*( )** – initialize a list as a copy of another
***VXWList::~VXWList*( )** – free up a list
***VXWList::add*( )** – add a node to the end of list
***VXWList::concat*( )** – concatenate two lists
***VXWList::count*( )** – report 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

DESCRIPTION

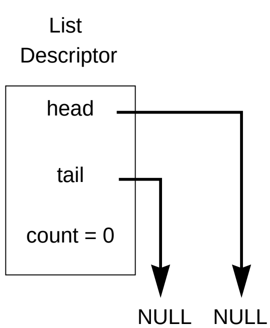
The VXWList class supports the creation and maintenance of a doubly linked list. The class contains pointers to the first and last nodes in the list, and a count of the number of nodes in the list. The nodes in the list are derived from the structure NODE, which provides two pointers: **NODE::next** 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**++ 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**

VXWMemPart

NAME **VXWMemPart** – memory partition classes (WFC Opt.)

METHODS
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

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

VXWModule

NAME

VXWModule – object module class (WFC Opt.)

METHODS

***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

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**, **VXWMemPart**, *VxWorks Programmer's Guide: C++ Development*

VXWMsgQ

NAME **VXWMsgQ** – message queue classes (WFC Opt.)

METHODS
***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

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*

VXWRingBuf

NAME **VXWRingBuf** – ring buffer class (WFC Opt.)

METHODS

***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

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**

VXWSem

NAME **VXWSem** – semaphore classes (WFC Opt.)

METHODS

***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

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*

1

VXWSmName

NAME

VXWSmName – naming behavior common to all shared memory classes (WFC Opt.)

METHODS

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

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.c** 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*

VXWSymTab

NAME

VXWSymTab – symbol table class (WFC Opt.)

METHODS

***VXWSymTab::VXWSymTab*( )** – create a symbol table
***VXWSymTab::VXWSymTab*( )** – create a symbol-table object
***VXWSymTab::~VXWSymTab*( )** – delete a symbol table
***VXWSymTab::add*( )** – create and add symbol to a symbol table, including 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

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**

VXWTask

NAME **VXWTask** – task class (WFC Opt.)

METHODS

***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

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*

VXWWd

NAME
VXWWd – watchdog timer class (WFC Opt.)

METHODS
***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

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*, *C++ Development*

wd33c93Lib

NAME

wd33c93Lib – WD33C93 SCSI-Bus Interface Controller (SBIC) library

ROUTINES

***wd33c93CtrlInit*( )** – initialize the user-specified fields in an SBIC structure
***wd33c93Show*( )** – display the values of all readable WD33C93 chip registers

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 **wd33c93Lib**1 and **wd33c93Lib**2 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 **wd33c93Lib**1, must be invoked. Similarly, ***wd33c93CtrlCreateScsi2*( )**, defined in **wd33c93Lib**2 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)

ROUTINES
***wd33c93CtrlCreate*( )** – create and partially initialize a WD33C93 SBIC structure

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)

ROUTINES
***wd33c93CtrlCreateScsi2*( )** – create and partially initialize an SBIC structure

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*

wdbEndPktDrv

NAME
wdbEndPktDrv – END based packet driver for lightweight UDP/IP

ROUTINES
No Callable Routines

DESCRIPTION
This is an END based driver for the WDB system. It uses the MUX and END based drivers to allow for interaction between the target and target server.

USAGE
The driver is typically only called only from **usrWdb.c**. The only directly callable routine in this module is ***wdbEndPktDevInit*( )**. Your **configAll.h**file will have to be modified so that **WDB_COMM_TYPE** is defined as **WDB_COMM_END**.

DATA BUFFERING
The drivers only need to handle one input packet at a time because the WDB protocol only supports one outstanding host-request at a time. If multiple input packets arrive, the driver can simply drop them. The driver then loans the input buffer to the WDB agent, and the agent invokes a driver callback when it is done with the buffer.

For output, the agent will pass the driver a chain of mbufs, which the driver must send as a packet. When it is done with the mbufs, it calls ***wdbMbufChainFree*( )** to free them. The header file **wdbMbuflib.h** provides the calls for allocating, freeing, and initializing mbufs for use with the lightweight UDP/IP interpreter. It ultimatly makes calls to the routines wdbMbufAlloc and wdbMbufFree, which are provided in source code in **usrWdb.c**.

wdbLib

NAME
wdbLib – WDB agent context management library

ROUTINES
***wdbSystemSuspend*( )** – suspend the system.

DESCRIPTION
This library provides a routine to transfer control from the run time system to the WDB agent running in external mode. This agent in external mode allows a system-wide control, including ISR debugging, from a host tool (e.g.: Crosswind, WindSh ...) through the target server and the WDB communcation link.

INCLUDE FILES
wdb/wdbLib.h

SEE ALSO
API Guide: WTX Protocol , *Tornado User's Guide: Overview*

wdbNetromPktDrv

NAME **wdbNetromPktDrv** – NETROM packet driver for the WDB agent

ROUTINES ***wdbNetromPktDevInit*( )** – initialize a NETROM packet device for the WDB agent

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.

wdbPipePktDrv

NAME **wdbPipePktDrv** – pipe packet driver for lightweight UDP/IP

ROUTINES ***wdbPipePktDevInit*( )** – initialize a pipe packet device.

DESCRIPTION This module is a pipe for drivers interfacing with the WDB agent's lightweight UDP/IP interpreter. It can be used as a starting point when writing new drivers. Such drivers are the lightweight equivalent of a network interface driver.

These drivers, along with the lightweight UDP-IP interpreter, have two benefits over the stand combination of a netif driver + the full VxWorks networking stack; First, they can run in a much smaller amout of target memory because the lightweight UDP-IP interpreter is much smaller than the VxWorks network stack (about 800 bytes total). Second, they provide a communication path which is independant of the OS, and thus can be used to support an external mode (e.g., monitor style) debug agent.

Throughout this file the word "pipe" is used in place of a real driver name. For example, if you were writing a lightweight driver for the lance ethernet chip, you would want to substitute "pipe" with "ln" throughout this file.

PACKET READY CALLBACK

When the driver detects that a packet has arrived (either in its receiver ISR or in its poll input routine), it invokes a callback to pass the data to the debug agent. Right now the callback routine is called "udpRcv", however other callbacks may be added in the future. The driver's ***wdbPipeDevInit*( )** routine should be passed the callback as a parameter and place it in the device data structure. That way the driver will continue to work if new callbacks are added later.

MODES

Ideally the driver should support both polled and interrupt mode, and be capable of switching modes dynamically. However this is not required. When the agent is not running, the driver will be placed in "interrupt mode" so that the agent can be activated as soon as a packet arrives. If your driver does not support an interrupt mode, you can simulate this mode by spawning a VxWorks task to poll the device at periodic intervals and simulate a receiver ISR when a packet arrives.

For dynamically mode switchable drivers, be aware that the driver may be asked to switch modes in the middle of its input ISR. A driver's input ISR will look something like this:

```
doSomeStuff();
pPktDev->wdbDrvIf.stackRcv (pMbuf);   /* invoke the callback */
doMoreStuff();
```

If this channel is used as a communication path to an external mode debug agent, then the agent's callback will lock interrupts, switch the device to polled mode, and use the device in polled mode for awhile. Later on the agent will unlock interrupts, switch the device back to interrupt mode, and return to the ISR. In particular, the callback can cause two mode switches, first to polled mode and then back to interrupt mode, before it returns. This may require careful ordering of the callback within the interrupt handler. For example, you may need to acknowledge the interrupt within the ***doSomeStuff*( )** processing rather than the ***doMoreStuff*( )** processing.

USAGE

The driver is typically only called only from **usrWdb.c**. The only directly callable routine in this module is ***wdbPipePktDevInit*( )**. You will need to modify **usrWdb.c** to allow your driver to be initialized by the debug agent. You will want to modify **usrWdb.c** to include your driver's header file, which should contain a definition of **WDB_PIPE_PKT_MTU**. There is a default user-selectable macro called **WDB_MTU**, which must be no larger than **WDB_PIPE_PKT_MTU**. Modify the begining of **usrWdb.c** to insure that this is the case by copying the way it is done for the other drivers. The routine ***wdbCommIfInit*( )** also needs to be modified so that if your driver is selected as the **WDB_COMM_TYPE**, then your drivers init routine will be called. Search **usrWdb.c** for the macro **WDB_COMM_CUSTOM** and mimic that style of initialization for your driver.

DATA BUFFERING

The drivers only need to handle one input packet at a time because the WDB protocol only supports one outstanding host-request at a time. If multiple input packets arrive, the driver can simply drop them. The driver then loans the input buffer to the WDB agent, and the agent invokes a driver callback when it is done with the buffer.

For output, the agent will pass the driver a chain of mbufs, which the driver must send as a packet. When it is done with the mbufs, it calls ***wdbMbufChainFree*( )** to free them. The header file **wdbMbuflib.h** provides the calls for allocating, freeing, and initializing mbufs for use with the lightweight UDP/IP interpreter. It ultimatly makes calls to the routines ***wdbMbufAlloc*( )** and ***wdbMbufFree*( )**, which are provided in source code in **usrWdb.c**.

wdbSlipPktDrv

NAME **wdbSlipPktDrv** – a serial line packetizer for the WDB agent

ROUTINES ***wdbSlipPktDevInit*( )** – initialize a SLIP packet device for a WDB agent

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.

wdbTsfsDrv

NAME **wdbTsfsDrv** – virtual generic file I/O driver for the WDB agent

ROUTINES ***wdbTsfsDrv*( )** – initialize the TSFS device driver for a WDB agent

DESCRIPTION This library provides a virtual file I/O driver for use with the WDB agent. I/O is performed on this virtual I/O device exactly as it would be on any device referencing a VxWorks file system. File operations, such as ***read*( )** and ***write*( )**, move data over a virtual I/O channel created between the WDB agent and the Tornado target server. The operations are then executed on the host file system. Because file operations are actually performed on the host file system by the target server, the file system presented by this virtual I/O device is known as the target-server file system, or TSFS.

The driver is installed with ***wdbTsfsDrv*( )**, creating a device typically called **/tgtsvr**. See the manual page for ***wdbTsfsDrv*( )** for more information about using this function. The initialization is done automatically, enabling access to TSFS, when **INCLUDE_WDB_TSFS** is defined. The target server also must have TSFS enabled in order to use TSFS. See the *WindView User's Guide: Data Upload* and the target server documentation.

TSFS SOCKETS

TSFS provides all of the functionality of other VxWorks file systems. For details, see the *VxWorks Programmer's Guide: I/O System and Local File Systems.*In addition to normal files, however, TSFS also provides basic access to TCP sockets. This includes opening the client side of a TCP socket, reading, writing, and closing the socket. Basic ***setsockopt*( )** commands are also supported.

To open a TCP socket using TSFS, use a filename of the form:

```
TCP:server_name | server_ip:port_number
```

To open and connect a TCP socket to a server socket located on a server named **mongoose**, listening on port 2010, use the following:

```
fd = open ("/tgtsvr/TCP:mongoose:2010", 0, 0)
```

The open flags and permission arguments to the open call are ignored when opening a socket through TSFS. If the server **mongoose** has an IP number of **144.12.44.12**, you can use the following equivalent form of the command:

```
fd = open ("/tgtsvr/TCP:144.12.44.12:2010", 0, 0)
```

DIRECTORIES

All directory functions, such as ***mkdir*( )**, ***rmdir*( )**, ***opendir*( )**, ***readdir*( )**, ***closedir*( )**, and ***rewinddir*( )** are supported by TSFS, regardless of whether the target server providing TSFS is being run on a UNIX or Windows host.

While it is possible to open and close directories using ***open*( )** and ***close*( )**, it is not possible to read from a directory using ***read*( )**. Instead, ***readdir*( )** must be used. It is also not possible to write to an open directory, and opening a directory for anything other than read-only results in an error, with **errno** set to **EISDIR**. Calling ***read*( )** on a directory returns **ERROR** with **errno** set to **EISDIR**.

OPEN FLAGS

When the target server that is providing the TSFS is running on a Windows host, the default file-translation mode is binary translation. If text translation is required, then **WDB_TSFS_O_TEXT** can be included in the mode argument to ***open*( )**. For example:

```
fd = open ("/tgtsvr/foo", O_CREAT | O_RDWR | WDB_TSFS_O_TEXT, 0777)
```

If the target server providing TSFS services is running on a UNIX host, **WDB_TSFS_O_TEXT** is ignored.

TGTSVR

For general information on the target server, see the reference entry for **tgtsvr**. In order to use this library, the target server must support and be configured with the following options:

-R *root*
Specify the root of the host's file system that is visible to target processes using TSFS. This flag is required to use TSFS. Files under this root are by default read only. To allow read/write access, specify -RW.

-RW
Allow read and write access to host files by target processes using TSFS. When this option is specified, access to the target server is restricted as if **-L** were also specified.

IOCTL SUPPORT

TSFS supports the following ***ioctl*( )** functions for controlling files and sockets. Details about each function can be found in the documentation listed below.

FIOSEEK

FIOWHERE

FIOMKDIR
Create a directory. The path, in this case **/tgtsvr/tmp**, must be an absolute path prefixed with the device name. To create the directory **/tmp** on the root of the TSFS file system use the following:

```
status = ioctl (fd, FIOMKDIR, "/tgtsvr/tmp")
```

FIORMDIR
Remove a directory. The path, in this case **/tgtsvr/foo**, must be an absolute path prefixed with the device name. To remove the directory **/foo** from the root of the TSFS file system, use the following:

```
status = ioctl (fd, FIORMDIR, "/tgtsvr/foo")
```

FIORENAME
Rename the file or directory represented by **fd** to the name in the string pointed to by **arg**. The path indicated by **arg** may be prefixed with the device name or not. Using this ***ioctl*( )** function with the path **/foo/goo**produces the same outcome as the path **/tgtsvr/foo/goo**. The path is not modified to account for the current working directory, and therefore must be an absolute path.

```
char *arg = "/tgtsvr/foo/goo";
status = ioctl (fd, FIORENAME, arg);
```

FIOREADDIR

FIONREAD
Return the number of bytes ready to read on a TSFS socket file descriptor.

FIOFSTATGET

FIOGETFL

The following ***ioctl*( )** functions can be used only on socket file descriptors. Using these functions with ***ioctl*( )** provides similar behavior to the ***setsockopt*( )** and ***getsockopt*( )**

functions usually used with socket descriptors. Each command's name is derived from a ***getsockopt*( )**/***setsockopt*( )** command and works in exactly the same way as the respective ***getsockopt*( )**/***setsockopt*( )** command. The functions ***setsockopt*( )** and ***getsockopt*( )** can not be used with TSFS socket file descriptors.

For example, to enable recording of debugging information on the TSFS socket file descriptor, call:

```
int arg = 1;
status = ioctl (fd, SO_SETDEBUG, arg);
```

To determine whether recording of debugging information for the TSFS-socket file descritptor is enabled or disabled, call:

```
int arg;
status = ioctl (fd, SO_GETDEBUG, & arg);
```

After the call to ***ioctl*( )**, **arg** contains the state of the debugging attribute.

The ***ioctl*( )** functions supported for TSFS sockets are:

SO_SETDEBUG
Equivalent to ***setsockopt*( )** with the **SO_DEBUG** command.

SO_GETDEBUG
Equivalent to ***getsockopt*( )** with the **SO_DEBUG** command.

SO_SETSNDBUF
This command changes the size of the send buffer of the host socket. The configuration of the WDB channel between the host and target also affects the number of bytes that can be written to the TSFS file descriptor in a single attempt.

SO_SETRCVBUF
This command changes the size of the receive buffer of the host socket. The configuration of the WDB channel between the host and target also affects the number of bytes that can be read from the TSFS file descriptor in a single attempt.

SO_SETDONTROUTE
Equivalent to ***setsockopt*( )** with the **SO_DONTROUTE** command.

SO_GETDONTROUTE
Equivalent to ***getsockopt*( )** with the **SO_DONTROUTE** command.

SO_SETOOBINLINE
Equivalent to ***setsockopt*( )** with the **SO_OOBINLINE** command.

SO_GETOOBINLINE
Equivalent to ***getsockopt*( )** with the **SO_OOBINLINE** command.

SO_SNDURGB
The **SO_SNDURGB** command sends one out-of-band byte (pointed to by **arg**) through the socket.

ERROR CODES The routines in this library return the VxWorks error codes that most closely match the errnos generated by the corresponding host function. If an error is encountered that is due to a WDB failure, a WDB error is returned instead of the standard VxWorks **errno**. If an **errno** generated on the host has no reasonable VxWorks counterpart, the host **errno** is passed to the target calling routine unchanged.

SEE ALSO *Tornado User's Guide, VxWorks Programmer's Guide: I/O System, Local File Systems*

wdbUlipPktDrv

NAME **wdbUlipPktDrv** – WDB communication interface for the ULIP driver

ROUTINES ***wdbUlipPktDevInit*( )** – initialize the WDB agent's communication functions for ULIP

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. It provides a communication path which supports both a task mode and an external mode WDB agent.

wdbUserEvtLib

NAME **wdbUserEvtLib** – WDB user event library

ROUTINES ***wdbUserEvtLibInit*( )** – include the WDB user event library
***wdbUserEvtPost*( )** – post a user event string to host tools.

DESCRIPTION This library contains routines for sending WDB User Events. The event is sent through the WDB agent, the WDB communication link and the target server to the host tools that have registered for it. The event received by host tools will be a WTX user event string.

INCLUDE FILES **wdb/wdbLib.h**

SEE ALSO *Tornado API Programmer's Guide: WTX Protocol*

wdbVioDrv

NAME

wdbVioDrv – virtual tty I/O driver for the WDB agent

ROUTINES

***wdbVioDrv*( )** – initialize the tty driver for a WDB agent

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*( )**. 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

ROUTINES

***wdCreate*( )** – create a watchdog timer
***wdDelete*( )** – delete a watchdog timer
***wdStart*( )** – start a watchdog timer
***wdCancel*( )** – cancel a currently counting watchdog

DESCRIPTION

This library provides a general watchdog timer facility. Any task may create a watchdog timer and use it to run a specified routine in the context of the system-clock ISR, after a specified delay.

Once a timer has been created with ***wdCreate*( )**, it can be started with ***wdStart*( )**. The ***wdStart*( )** routine specifies what routine to run, a parameter for that routine, and the amount of time (in ticks) before the routine is to be called. (The timeout value is in ticks as determined by the system clock; see ***sysClkRateSet*( )** for more information.) After the specified delay ticks have elapsed (unless ***wdCancel*( )** is called first to cancel the timer) the timeout routine is invoked with the parameter specified in the ***wdStart*( )** call. The timeout routine is invoked whether the task which started the watchdog is running, suspended, or deleted.

The timeout routine executes only once per ***wdStart*( )** invocation; there is no need to cancel a timer with ***wdCancel*( )** after it has expired, or in the expiration callback itself.

Note that the timeout routine is invoked at interrupt level, rather than in the context of the task. Thus, there are restrictions on what the routine may do. Watchdog routines are constrained to the same rules as interrupt service routines. For example, they may not take semaphores, issue other calls that may block, or use I/O system routines like ***printf*( )**.

EXAMPLE

In the fragment below, if ***maybeSlowRoutine*( )** takes more than 60 ticks, ***logMsg*( )** will be called with the string as a parameter, causing the message to be printed on the console. Normally, of course, more significant corrective action would be taken.

```
WDOG_ID wid = wdCreate ();
wdStart (wid, 60, logMsg, "Help, I've timed out!");
maybeSlowRoutine ();        /* user-supplied routine */
wdCancel (wid);
```

INCLUDE FILES

wdLib.h

SEE ALSO

logLib, *VxWorks Programmer's Guide: Basic OS*

---

wdShow

NAME

wdShow – watchdog show routines

ROUTINES

***wdShowInit*( )** – initialize the watchdog show facility
***wdShow*( )** – show information about a watchdog

DESCRIPTION

This library provides routines to show watchdog statistics, such as watchdog activity, a watchdog routine, etc.

The routine ***wdShowInit*( )** links the watchdog show facility into the VxWorks system. It is called automatically when this show facility is configured into VxWorks using either of the following methods:

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

– If you use the Tornado project facility, select **INCLUDE_WATCHDOGS_SHOW**.

INCLUDE FILES

wdLib.h

SEE ALSO

wdLib, *VxWorks Programmer's Guide: Basic OS, Target Shell*, windsh, *Tornado User's Guide: Shell*

winSio

NAME

winSio – win serial driver

ROUTINES

***winDevInit*( )** – initialize a **WIN_CHAN**
***winDevInit2*( )** – initialize a **WIN_CHAN**, part 2
***winIntRcv*( )** – handle a channel's receive-character interrupt
***winIntTx*( )** – transmit a single character.
***dummyCallback*( )** – dummy callback routine

DESCRIPTION

This is the console serial driver for the Windows simulator. It receives character interrupts from Windows and sends them to VxWorks. Device data structures are defined in the header file **h/drv/sio/winSio.h**. A device data structure, **WIN_CHAN**, is defined for each channel.

USAGE

The driver is typically only called only by the BSP. The directly callable routines in this module are ***winDevInit*( )**, ***winDevInit2*( )**, ***winIntRcv*( )**, and ***winIntTx*( )**.

The BSP calls ***winDevInit*( )** to initialize or reset the device. It connects the driver's interrupt handlers (winIntRcv and winIntTx) using ***intConnect*( )**. After connecting the interrupt handlers, the BSP calls ***winDevInit2*( )** to inform the driver that interrupt mode operation is now possible.

BSP

By convention all the BSP-specific serial initialization is performed in a file called **sysSerial.c**, which is #include'ed by **sysLib.c**. **sysSerial.c** implements at least four functions, ***sysSerialHwInit*( )**, ***sysSerialHwInit2*( )**, ***sysSerialChanGet*( )**, and ***sysSerialReset*( )**, which work as follows:

***sysSerialHwInit*( )** is called by ***sysHwInit*( )** to initialize the serial devices. This routine will initialize all the board specific fields in the **WIN_CHAN** structure (e.g., register I/O addresses, etc.) before calling ***winDevInit*( )**, which resets the device and installs the driver function pointers. ***sysSerialHwInit*( )** should also perform any other processing needed for the serial drivers, such as configuring on-board interrupt controllers as appropriate.

***sysSerialHwInit2*( )** is called by ***sysHwInit2*( )** to connect the serial driver's interrupt handlers using ***intConnect*( )**. After connecting the interrupt handlers, the call to ***winDevInit2*( )** is made to permit interrupt mode operations to begin.

***sysSerialChanGet*( )** is called by ***usrRoot*( )** to get the serial channel descriptor associated with a serial channel number. The routine takes a single parameter which is a channel number ranging between zero and **NUM_TTY**. It returns a pointer to the corresponding channel descriptor, **SIO_CHAN ***, which is just the address of the **WIN_CHAN** structure.

***sysSerialReset*( )** is called from ***sysToMonitor*( )** and should reset the serial devices to an inactive state.

INCLUDE FILES

drv/sio/winSio.h, **sioLib.h**

1

z8530Sio

NAME **z8530Sio** – Z8530 SCC Serial Communications Controller driver

ROUTINES
*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

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. Supported baud rates range from 50 to 38400. The default baud rate is **Z8530_DEFAULT_BAUD** (9600). The BSP may redefine this.

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**).

This driver handles setting of hardware options such as parity (odd, even) and number of data bits (5, 6, 7, 8). Hardware flow control is provided with the signals CTS on transmit and DSR on read. Refer to the target documentation for the RS232 port configuration. The function HUPCL (hang up on last close) is supported. Default hardware options are defined by **Z8530_DEFAULT_OPTIONS**. The BSP may redefine them.

All device registers are accessed via BSP-defined macros so that memory-mapped as well as I/O space accesses can be supported. The BSP may redefine the **REG_8530_READ** and **REG_8530_WRITE** macros as needed. By default, they are defined as simple memory-mapped accesses.

The BSP may define **DATA_REG_8530_DIRECT** to cause direct access to the Z8530 data register, where hardware permits it. By default, it is not defined.

The BSP may redefine the macro for the channel reset delay **Z8530_RESET_DELAY** as well as the channel reset delay counter value **Z8530_RESET_DELAY_COUNT** as required. The delay is defined as the minimum time between successive chip accesses (6 PCLKs + 200 nSec for a Z8530, 4 PCLKs for a Z85C30 or Z85230) plus an additional 4 PCLKs. At a typical PCLK frequency of 10 MHz, each PCLK is 100 nSec, giving a minimum reset delay of:

Z8530

10 PCLKs + 200 nSec = 1200 nSec = 1.2 uSec

Z85x30: 8 PCLKs = 800 nSec = 0.8 uSec

INCLUDE FILES

drv/sio/z8530Sio.h

zbufLib

NAME

zbufLib – zbuf interface library

ROUTINES

***zbufCreate*( )** – create an empty zbuf
***zbufDelete*( )** – delete a zbuf
***zbufInsert*( )** – insert a zbuf into another zbuf
***zbufInsertBuf*( )** – create a zbuf segment from a buffer and insert into a zbuf
***zbufInsertCopy*( )** – copy buffer data into a zbuf
***zbufExtractCopy*( )** – copy data from a zbuf to a buffer
***zbufCut*( )** – delete bytes from a zbuf
***zbufSplit*( )** – split a zbuf into two separate zbufs
***zbufDup*( )** – duplicate a zbuf
***zbufLength*( )** – determine the length in bytes of a zbuf
***zbufSegFind*( )** – find the zbuf segment containing a specified byte location
***zbufSegNext*( )** – get the next segment in a zbuf
***zbufSegPrev*( )** – get the previous segment in a zbuf
***zbufSegData*( )** – determine the location of data in a zbuf segment
***zbufSegLength*( )** – determine the length of a zbuf segment

DESCRIPTION

This library contains routines to create, build, manipulate, and delete zbufs. Zbufs, also known as "zero copy buffers," are a data abstraction designed to allow software modules to share buffers without unnecessarily copying data.

To support the data abstraction, the subroutines in this library hide the implementation details of zbufs. This also maintains the library's independence from any particular implementation mechanism, 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*. The *offset* may be positive or negative, and is simply the number of bytes from the beginning of the segment *zbufSeg*.

A *zbufSeg* can be specified as NULL, to identify the segment at the beginning of a zbuf. If *zbufseg* is NULL, *offset* is the absolute offset to any byte in the zbuf. However, it is more efficient to identify a zbuf byte location relative to the *zbufSeg*that contains it; see ***zbufSegFind*( )** to convert any *zbufSeg*, *offset*pair to the most efficient equivalent.

Negative *offset* values always refer to bytes before the corresponding *zbufSeg*, and are 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

ROUTINES
***zbufSockLibInit*( )** – initialize the zbuf socket interface library
***zbufSockSend*( )** – send zbuf data to a TCP socket
***zbufSockSendto*( )** – send a zbuf message to a UDP socket
***zbufSockBufSend*( )** – create a zbuf from user data and send it to a TCP socket
***zbufSockBufSendto*( )** – create a zbuf from a user message and send it to a UDP socket
***zbufSockRecv*( )** – receive data in a zbuf from a TCP socket
***zbufSockRecvfrom*( )** – receive a message in a zbuf from a UDP socket

DESCRIPTION This library contains routines that communicate over BSD sockets using the *zbuf interface* described in the **zbufLib** manual page. These zbuf socket calls communicate over BSD sockets in a similar manner to the socket routines in **sockLib**, but they avoid copying data unnecessarily between application buffers and network buffers.

SEE ALSO **zbufLib**, **sockLib**, *VxWorks Programmer's Guide: Network*

2
Subroutines

2

*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*

aic7880CtrlCreate( )

NAME

aic7880CtrlCreate( ) – create a control structure for the AIC 7880

SYNOPSIS

```
AIC_7880_SCSI_CTRL * aic7880CtrlCreate
    (
    int busNo,    /* PCI bus Number */
    int devNo,    /* PCI device Number */
    int scsiBusId /* SCSI Host Adapter Bus Id */
    )
```

DESCRIPTION

This routine creates an **AIC_7880_SCSI_CTRL** structure and must be called before using the SCSI Host Adapter chip. It must be called exactly once for a specified Host Adapter.

RETURNS

A pointer to the **AIC_7880_SCSI_CTRL** structure, or NULL if memory is unavailable or there are invalid parameters.

SEE ALSO

aic7880Lib

aic7880dFifoThresholdSet( )

NAME *aic7880dFifoThresholdSet*( ) – set the data FIFO threshold.

SYNOPSIS

```
STATUS aic7880dFifoThresholdSet
    (
    SCSI_CTRL * pScsiCtrl, /* ptr to SCSI controller */
    UBYTE       threshHold /* data FIFO threshold value */
    )
```

DESCRIPTION This routine specifies to the AIC-7880 host adapter how to manage its data FIFO. Below is a description of the threshold values for SCSI reads and writes.

SCSI READS

– 0 Xfer data from FIFO as soon as it is available.
– 1 Xfer data from FIFO as soon as the FIFO is half full.
– 2 Xfer data from FIFO as soon as the FIFO is 75% full.
– 3 Xfer data from FIFO as soon as the FIFO is 100% full.

SCSI WRITES

– 0 Xfer data as soon as there is room in the FIFO.
– 1 Xfer data to FIFO as soon as it is 50% empty.
– 2 Xfer data to FIFO as soon as it is 75% empty.
– 3 Xfer data to FIFO as soon as the FIFO is empty.

RETURNS OK or ERROR if the threshold value is not within the valid range.

SEE ALSO **aic7880Lib**

aic7880EnableFast20( )

NAME *aic7880EnableFast20*( ) – enable double speed SCSI data transfers

SYNOPSIS

```
VOID aic7880EnableFast20
    (
    SCSI_CTRL * pScsiCtrl, /* ptr to SCSI controller */
    BOOL        enable     /* enable = 1 / disable = 0 */
    )
```

DESCRIPTION This routine enables double speed SCSI data transfers for the SCSI host adapter. This allows the host adapter to transfer data upto 20 MB/s for an 8 bit device and upto 40 MB/s for a 16 bit device.

RETURNS N/A

SEE ALSO **aic7880Lib**

aic7880GetNumOfBuses( )

NAME *aic7880GetNumOfBuses*( ) – perform a PCI bus scan

SYNOPSIS

```
DWORD aic7880GetNumOfBuses ()
```

DESCRIPTION This routine provides a callback mechanism from the HIM to the OSM It allows the OSM to scan the PCI bus, before the HIM is allowed to perform the bus scan.

RETURNS 0x55555555 if the OSM is not able to conduct its own bus scan

SEE ALSO **aic7880Lib**

aic7880ReadConfig( )

NAME *aic7880ReadConfig*( ) – read from PCI config space

SYNOPSIS

```
DWORD aic7880ReadConfig
    (
    cfp_struct * configPtr, /* ptr to cf_struct */
    UBYTE        busNo,     /* PCI bus number */
    UBYTE        devNo,     /* PCI device number */
    UBYTE        regNo      /* register */
    )
```

DESCRIPTION This routine provides a callback mechanism from the HIM to the OSM. The purpose of this routine is to allow the OSM to do its own Read access of the PCI configuration space. If the OSM cannot successfully complete the Read access, the OSM returns 0x55555555. If this happens the HIM attempts to conduct the configuration space Read access.

RETURNS value read or 0x55555555, if the OSM is not able to conduct read access to the PCI configuration space.

SEE ALSO **aic7880Lib**

2

aic7880ScbCompleted( )

NAME *aic7880ScbCompleted*( ) – successfully completed execution of a client thread

SYNOPSIS

```
VOID aic7880ScbCompleted
    (
    sp_struct * pScb /* ptr to completed SCSI Command Block */
    )
```

DESCRIPTION This routine is called from within the context of the ISR. The HIM calls this routine passing in the pointer of the of the completed SCB. This routine sets the thread status, handles the completed SCB and returns program control back to the HIM which then returns from the ***PH_IntHandler*( )** routine.

This routine could be called more than once from the same PH_IntHandler call. Each call to this routine indicates the completion of an SCB. For each SCB completed, this routine sets the event type and calls the appropriate AIC-7880 event handler routines which sets the SCSI Controller, SCSI Physical Device and SCSI Thread, state variables appropriately. This routine also handles synchronization with the SCSI Manager so that the next runnable thread can be scheduled for execution.

RETURNS N/A

SEE ALSO **aic7880Lib**

aic7880WriteConfig( )

NAME *aic7880WriteConfig*( ) – read to PCI config space

SYNOPSIS

```
DWORD aic7880WriteConfig
    (
    cfp_struct * config_ptr, /* ptr to cf_struct */
    UBYTE        busNo,      /* PCI bus number */
    UBYTE        devNo,      /* PCI device number */
    UBYTE        regNo,      /* register */
    DWORD        regVal      /* register value */
    )
```

DESCRIPTION This routine provides a callback mechanism from the HIM to the OSM. The purpose of this routine is to allow the OSM to do its own write access of the PCI configuration space.

If the OSM cannot successfully complete the write access, the OSM returns 0x55555555. If this happens the HIM attempts to conduct the configuration space write access.

RETURNS OK or 0x55555555, if the OSM is not able to conduct write access to the PCI configuration space.

SEE ALSO **aic7880Lib**

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

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

RETURNS
OK if queued successfully, otherwise ERROR.

ERRNO
EINVAL, EBADF

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, or ERROR.

ERRNO **EINVAL**, **EINPROGRESS**

INCLUDE FILES **aio.h**

SEE ALSO **aioPxLib**

---

aio_suspend( )

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

SYNOPSIS

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

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

– at least one of the previously submitted asynchronous I/O operations referenced by *list* has completed,

– a signal interrupts the function, or

– the time interval specified by *timeout* has passed (if *timeout* is not NULL).

RETURNS OK if an AIO request completes, otherwise ERROR.

ERRNO **EAGAIN**, **EINTR**

INCLUDE FILES **aio.h**

SEE ALSO **aioPxLib**

aio_write( )

NAME *aio_write*( ) – initiate an asynchronous write (POSIX)

SYNOPSIS

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

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

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

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

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

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

RETURNS OK if write queued successfully, otherwise ERROR.

ERRNO **EBADF**, **EINVAL**

INCLUDE FILES **aio.h**

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

ambaDevInit( )

NAME *ambaDevInit*( ) – initialise an AMBA channel

SYNOPSIS

```
void ambaDevInit
    (
    AMBA_CHAN * pChan /* ptr to AMBA_CHAN describing this channel */
    )
```

DESCRIPTION This routine initialises some **SIO_CHAN** function pointers and then resets the chip to a quiescent state. Before this routine is called, the BSP must already have initialised all the device addresses, etc. in the **AMBA_CHAN** structure.

RETURNS N/A

SEE ALSO **ambaSio**

ambaIntRx( )

NAME *ambaIntRx*( ) – handle a receiver interrupt

SYNOPSIS

```
void ambaIntRx
    (
    AMBA_CHAN * pChan /* ptr to AMBA_CHAN describing this channel */
    )
```

DESCRIPTION This routine handles read interrupts from the UART.

RETURNS N/A

SEE ALSO **ambaSio**

ambaIntTx( )

NAME *ambaIntTx*( ) – handle a transmitter interrupt

SYNOPSIS

```
void ambaIntTx
    (
    AMBA_CHAN * pChan /* ptr to AMBA_CHAN describing this channel */
    )
```

DESCRIPTION This routine handles write interrupts from the UART.

RETURNS N/A

SEE ALSO **ambaSio**

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, **S_arpLib_INVALID_FLAG**

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.

EXAMPLE

```
-> arpShow
LINK LEVEL ARP TABLE
destination      gateway              flags  Refcnt  Use      Interface
-----------------------------------------------------------------------
90.0.0.63        08:00:3e:23:79:e7    405      0       82       lo0
-----------------------------------------------------------------------
```

RETURNS N/A

SEE ALSO **netShow**

arptabShow( )

NAME *arptabShow*( ) – display the known ARP entries

SYNOPSIS `void arptabShow (void)`

DESCRIPTION This routine displays current Internet-to-Ethernet address mappings in the ARP table.

RETURNS N/A

SEE ALSO **netShow**

asctime( )

NAME

asctime( ) – convert broken-down time into a string (ANSI)

SYNOPSIS

```
char * asctime
    (
    const struct tm * timeptr /* broken-down time */
    )
```

DESCRIPTION

This routine converts the broken-down time pointed to by *timeptr* into a string of the form:

```
SUN SEP 16 01:03:52 1973\n\0
```

This routine is not reentrant. For a reentrant version, see ***asctime_r*( )**.

INCLUDE FILES

time.h

RETURNS

A pointer to the created string.

SEE ALSO

ansiTime

asctime_r( )

NAME

asctime_r( ) – convert broken-down time into a string (POSIX)

SYNOPSIS

```
int asctime_r
    (
    const struct tm * timeptr,    /* broken-down time */
    char *            asctimeBuf, /* buffer to contain string */
    size_t *          buflen      /* size of buffer */
    )
```

DESCRIPTION

This routine converts the broken-down time pointed to by *timeptr* into a string of the form:

```
SUN SEP 16 01:03:52 1973\n\0
```

The string is copied to *asctimeBuf*. This call is the POSIX re-entrant version of ***asctime*( )**.

INCLUDE FILES

time.h

RETURNS

The size of the created string.

SEE ALSO

ansiTime

2

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 performs hardware initialization of the ATA/IDE chip.

This routine 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

b( )

NAME

b( ) – set or display breakpoints

SYNOPSIS

```
STATUS b
    (
    INSTR * addr,  /* where to set breakpoint, 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 */
                   /* 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. 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 to delete breakpoint for, 0 = delete for all */
    )
```

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 */
    )
```

DESCRIPTION

This routine removes all breakpoints. To execute, enter:

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

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

RETURNS

OK, always.

SEE ALSO

dbgLib, ***bd*( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

bfill( )

NAME

bfill( ) – fill a buffer with a specified character

SYNOPSIS

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

DESCRIPTION

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

RETURNS

N/A

SEE ALSO

bLib, ***bfillBytes*( )**

2

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 function 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 */
    int     access, /* access type (arch dependant) */
    int     task,   /* task for which to set breakboint, 0 = set all tasks */
    int     count,  /* number of passes before hit */
    BOOL    quiet   /* TRUE = don't print debug info, FALSE = print info */
    )
```

DESCRIPTION

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

NOTE

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

RETURNS

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

SEE ALSO

dbgLib, *b*( ), *VxWorks Programmer's Guide: Target Shell*

bind( )

NAME

bind( ) – bind a name to a socket

SYNOPSIS

```
STATUS bind
    (
    int               s,      /* socket descriptor */
    struct sockaddr * name,   /* name to be bound */
    int               namelen /* length of name */
    )
```

DESCRIPTION

This routine associates a network address (also referred to as its "name") with a specified socket so that other processes can connect or send to it. When a socket is created with *socket*( ), it belongs to an address family but has no assigned name.

RETURNS OK, or ERROR if there is an invalid socket, the address is either unavailable or in use, or the socket is already bound.

SEE ALSO **sockLib**

---

bindresvport( )

NAME *bindresvport*( ) – bind a socket to a privileged IP port

SYNOPSIS

```
STATUS bindresvport
    (
    int                   sd, /* socket to be bound */
    struct sockaddr_in * sin /* socket address -- value/result */
    )
```

DESCRIPTION This routine picks a port number between 600 and 1023 that is not being used by any other programs and binds the socket passed as *sd* to that port. Privileged IP ports (numbers between and including 0 and 1023) are reserved for privileged programs.

RETURNS OK, or ERROR if the address family specified in *sin* is not supported or the call fails.

SEE ALSO **remLib**

---

binvert( )

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

SYNOPSIS

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

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

RETURNS N/A

SEE ALSO **bLib**

bootBpAnchorExtract( )

NAME

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

SYNOPSIS

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

DESCRIPTION

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

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

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

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

RETURNS

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

SEE ALSO

bootLib

bootChange( )

NAME

bootChange( ) – change the boot line

SYNOPSIS

```
void bootChange (void)
```

DESCRIPTION

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

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

RETURNS N/A

SEE ALSO **usrLib**, windsh, *Tornado User's Guide: Shell*

bootLeaseExtract( )

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

SYNOPSIS

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

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

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

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

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

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

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

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

RETURNS

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

SEE ALSO

bootLib

bootNetmaskExtract( )

NAME

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

SYNOPSIS

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

DESCRIPTION

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

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

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

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

RETURNS

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

SEE ALSO

bootLib

bootParamsPrompt( )

NAME

bootParamsPrompt( ) – prompt for boot line parameters

SYNOPSIS

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

DESCRIPTION

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

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

RETURNS

N/A

SEE ALSO

bootLib

bootParamsShow( )

NAME

bootParamsShow( ) – display boot line parameters

SYNOPSIS

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

DESCRIPTION

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

RETURNS

N/A

SEE ALSO

bootLib

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 ticks */
    )
```

DESCRIPTION

This routine sends the BOOTP message indicated by *pBootpMsg* using the network interface specified by *ifName*. The *pIpDest* argument specifies the destination IP address. In most cases, the broadcast address (255.255.255.255) is used. However, this parameter also accepts the IP address of a particular BOOTP server. That server must reside on the same subnet as the specified network interface.

A non-zero value for *port* specifies an alternate BOOTP server port. Otherwise, the default port (67) is used.

This routine always sets the values of the **bp_op**, **bp_xid**, and **bp_secs** members in the BOOTP message structure, but it allows the caller to assign values to any of the other members. However, if the **bp_hlen** member is 0, the routine uses the Ethernet address of the specified network interface for the **bp_chaddr** member and sets **bp_type** to 1 and **bp_hlen** to 6 as required for that address.

The ***bootpMsgSend*( )** routine will retransmit the BOOTP message if it gets no reply. The retransmission time increases exponentially but is bounded by the number of ticks specified in the *timeOut* parameter. 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.

ERRNO

S_bootpLib_INVALID_ARGUMENT
S_bootpLib_NO_BROADCASTS
S_bootpLib_TIME_OUT

SEE ALSO

bootpLib

2

bootpParamsGet( )

NAME

bootpParamsGet( ) – retrieve boot parameters using BOOTP

SYNOPSIS

```
STATUS bootpParamsGet
    (
    char *                ifName,      /* network interface name */
    int                   port,        /* optional port number */
    u_int                 timeOut,     /* timeout in ticks */
    struct bootpParams * pBootpParams /* parameters descriptor */
    )
```

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).

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.

pBootpParams is a structure pointer to a **bootpParams** structure that you can use to indicate the parameters of interest to you. The **bootpParams** structure is defined as follows:

```
struct bootpParams
    {
    struct in_addr *            clientAddr;
    struct in_addr *            bootHostAddr;
    char *                      bootfile;
    char *                      serverName;
    struct in_addr *            netmask;
    unsigned short *            timeOffset;
    struct in_addr_list *       routers;
    struct in_addr_list *       timeServers;
    struct in_addr_list *       nameServers;
    struct in_addr_list *       dnsServers;
    struct in_addr_list *       logServers;
    struct in_addr_list *       cookieServers;
    struct in_addr_list *       lprServers;
    struct in_addr_list *       impressServers;
    struct in_addr_list *       rlpServers;
    char *                      clientName;
    unsigned short *            filesize;
    char *                      dumpfile;
```

```
char *                      domainName;
struct in_addr *            swapServer;
char *                      rootPath;
char *                      extoptPath;
unsigned char *             ipForward;
unsigned char *             nonlocalSourceRoute;
struct in_addr_list *       policyFilter;
unsigned short *            maxDgramSize;
unsigned char *             ipTTL;
unsigned long *             mtuTimeout;
struct ushort_list *        mtuTable;
unsigned short *            intfaceMTU;
unsigned char *             allSubnetsLocal;
struct in_addr *            broadcastAddr;
unsigned char *             maskDiscover;
unsigned char *             maskSupplier;
unsigned char *             routerDiscover;
struct in_addr *            routerDiscAddr;
struct in_addr_list *       staticRoutes;
unsigned char *             arpTrailers;
unsigned long *             arpTimeout;
unsigned char *             etherPacketType;
unsigned char *             tcpTTL;
unsigned long *             tcpInterval;
unsigned char *             tcpGarbage;
char *                      nisDomain;
struct in_addr_list *       nisServers;
struct in_addr_list *       ntpServers;
char *                      vendString;
struct in_addr_list *       nbnServers;
struct in_addr_list *       nbddServers;
unsigned char *             nbNodeType;
char *                      nbScope;
struct in_addr_list *       xFontServers;
struct in_addr_list *       xDisplayManagers;
char *                      nispDomain;
struct in_addr_list *       nispServers;
struct in_addr_list *       ipAgents;
struct in_addr_list *       smtpServers;
struct in_addr_list *       pop3Servers;
struct in_addr_list *       nntpServers;
struct in_addr_list *       wwwServers;
struct in_addr_list *       fingerServers;
struct in_addr_list *       ircServers;
struct in_addr_list *       stServers;
struct in_addr_list *       stdaServers;
};
```

This structure allows the retrieval of any BOOTP option specified in RFC 1533. The list of 2-byte (unsigned short) values is defined as:

```
struct ushort_list
    {
    unsigned char       num;
    unsigned short *    shortlist;
    };
```

The IP address lists use the following similar definition:

```
struct in_addr_list
    {
    unsigned char       num;
    struct in_addr *    addrlist;
    };
```

When these lists are present, the routine stores values retrieved from the BOOTP reply in the location indicated by the **shortlist** or **addrlist**members. The amount of space available is indicated by the **num** member. When the routine returns, the **num** member indicates the actual number of entries retrieved. In the case of **bootpParams.policyFilter.num** and **bootpParams.staticRoutes.num**, the **num** member value should be interpreted as the number of IP address pairs requested and received.

The following members of the **bootpParams** structure are also used for both input and output:

clientAddr
: Contains a pointer that holds the client's Internet address. On input, if it contains a non-NULL value, it is interpreted as a pointer to an Internet address of type **struct in_addr** and passed on to the BOOTP server in the **bp_ciaddr** member of the BOOTP message structure (**BOOTP_MSG**). The server will use it as a lookup field into the BOOTP database. When a reply is received, the client's assigned Internet address is copied to the **clientAddr** member.

bootHostAddr
: Contains a pointer that holds the host's IP address. On input, if it contains a non-NULL value, 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. If NULL, the BOOTP message is sent to the local broadcast address. On return, the host's IP address is copied to the **bootHostAddr** member.

On input, if the **bootpParams.bootfile** member points to a non-empty string, the contents are passed to the BOOTP server in the **bp_file** member of the BOOTP message structure (**BOOTP_MSG**). When a reply is received, the file name retrieved from the BOOTP server is copied to the **bootpParams.bootfile** member as a NULL-terminated string.

The remaining elements in the BOOTP parameters descriptor are used to select options for retrieval from the BOOTP server. The BOOTP library attempts to retrieve the values for any options whose corresponding field pointers are non-NULL values. To obtain these parameters, the BOOTP server must support the vendor-specific options described in RFC 1048 (or its successors) and the corresponding parameters must be specified in the BOOTP server database. Where meaningful, the values are returned in host byte order.

The BOOTP request issued during system startup attempts to retrieve a subnet mask for the boot device, in addition to the host and client addresses, and the boot file name.

RETURNS
OK, or ERROR if unsuccessful.

SEE ALSO
bootpLib, **bootLib**, RFC 1048, RFC 1533

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

---

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 cache (68K, x86)

SYNOPSIS

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

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

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

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

For 386 processors and PENTIUMPRO processors with **SNOOP_ENABLED** data cache mode, this routine does nothing.

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 cache library

SYNOPSIS

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

DESCRIPTION

This routine initializes the cache library for the following processor cache families: Motorola 68K, Intel 960, Intel x86, PowerPC ARM, and the Solaris, HP-UX, and NT simulators. It initializes the function pointers and configures the caches to the specified cache modes.

68K PROCESSORS

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

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

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

X86 PROCESSORS

The caching mode **CACHE_WRITETHROUGH** is available for the x86 processor family.

POWER PC PROCESSORS

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

ARM PROCESSORS

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

ARM7TDMI (In ARM or Thumb state)

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

ARM710A

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

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

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

RETURNS OK

SEE ALSO **cacheArchLib**

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**

2

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**

2

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 )
```

2

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

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 ( )`

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
    (
    )
```

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 ( )
```

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 )
```

SEE ALSO **cacheI960JxALib**

cacheI960JxICLoadNLock( )

NAME *cacheI960JxICLoadNLock*( ) – load and lock the I960Jx instruction cache (i960)

SYNOPSIS

```
void cacheI960JxICLoadNLock ( )
```

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
    (
    )
```

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
    (
    )
```

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 a cache. Depending on cache design, the invalidation may be similar to the flush, or the tags may be invalidated 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**

2

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

2

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**

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**

cacheR333x0LibInit( )

NAME *cacheR333x0LibInit*( ) – initialize the R333x0 cache library

SYNOPSIS

```
STATUS cacheR333x0LibInit
    (
    CACHE_MODE instMode, /* instruction cache mode */
    CACHE_MODE dataMode  /* data cache mode */
    )
```

DESCRIPTION This routine initializes the function pointers for the R333x0 cache library. The board support package can select this cache library by calling this routine.

RETURNS OK.

SEE ALSO **cacheR333x0Lib**

cacheStoreBufDisable( )

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

SYNOPSIS

```
void cacheStoreBufDisable (void)
```

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

RETURNS N/A

SEE ALSO **cacheArchLib**

cacheStoreBufEnable( )

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

SYNOPSIS

```
void cacheStoreBufEnable (void)
```

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

RETURNS N/A

SEE ALSO **cacheArchLib**

cacheSun4ClearContext( )

NAME *cacheSun4ClearContext*( ) – clear a specific context from a Sun-4 cache

SYNOPSIS

```
STATUS cacheSun4ClearContext
    (
    CACHE_TYPE cache,  /* cache to clear */
    void *     address /* virtual address */
    )
```

DESCRIPTION This routine flushes and invalidates a specified context from the specified Sun-4 cache.

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

SEE ALSO **cacheSun4Lib**

cacheSun4ClearLine( )

NAME *cacheSun4ClearLine*( ) – clear a line from a Sun-4 cache

SYNOPSIS

```
STATUS cacheSun4ClearLine
    (
    CACHE_TYPE cache,  /* cache to clear */
    void *     address /* virtual address */
    )
```

DESCRIPTION This routine flushes and invalidates a specified line from the specified Sun-4 cache.

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

SEE ALSO **cacheSun4Lib**

cacheSun4ClearPage( )

NAME *cacheSun4ClearPage*( ) – clear a page from a Sun-4 cache

SYNOPSIS

```
STATUS cacheSun4ClearPage
    (
    CACHE_TYPE cache,  /* cache to clear */
    void *     address /* virtual address */
    )
```

DESCRIPTION This routine flushes and invalidates a specified page from the specified Sun-4 cache.

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

SEE ALSO **cacheSun4Lib**

cacheSun4ClearSegment( )

NAME *cacheSun4ClearSegment*( ) – clear a segment from a Sun-4 cache

SYNOPSIS

```
STATUS cacheSun4ClearSegment
    (
    CACHE_TYPE cache,  /* cache to clear */
    void *     address /* virtual address */
    )
```

DESCRIPTION This routine flushes and invalidates a specified segment from the specified Sun-4 cache.

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

SEE ALSO **cacheSun4Lib**

cacheSun4LibInit( )

NAME

cacheSun4LibInit( ) – initialize the Sun-4 cache library

SYNOPSIS

```
STATUS cacheSun4LibInit
    (
    CACHE_MODE instMode, /* instruction cache mode */
    CACHE_MODE dataMode  /* data cache mode */
    )
```

DESCRIPTION

This routine initializes the function pointers for the Sun Microsystems Sun-4 cache library. The board support package can select this cache library by assigning the function pointer **sysCacheLibInit** to *cacheSun4LibInit*( ).

The only available mode for the Sun-4 cache is **CACHE_WRITETHROUGH**.

RETURNS

OK, or ERROR if cache control is not supported.

SEE ALSO

cacheSun4Lib

cacheTextUpdate( )

NAME

cacheTextUpdate( ) – synchronize the instruction and data caches

SYNOPSIS

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

DESCRIPTION

This routine flushes the data cache, then invalidates the instruction cache. This operation forces the instruction cache to fetch code that may have been created via the data path.

RETURNS

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

SEE ALSO

cacheLib

cacheTiTms390LibInit( )

NAME

cacheTiTms390LibInit( ) – initialize the TI TMS390 cache library

SYNOPSIS

```
STATUS cacheTiTms390LibInit
    (
    CACHE_MODE instMode, /* instruction cache mode */
    CACHE_MODE dataMode  /* data cache mode */
    )
```

DESCRIPTION

This routine initializes the function pointers for the TI TMS390 cache library. The board support package can select this cache library by assigning the function pointer **sysCacheLibInit** to *cacheTiTms390LibInit*( ).

The only available cache mode is **CACHE_COPYBACK**.

RETURNS

OK, or ERROR if cache control is not supported.

SEE ALSO

cacheTiTms390Lib

cacheTiTms390PhysToVirt( )

NAME

cacheTiTms390PhysToVirt( ) – translate a physical address for drivers

SYNOPSIS

```
void * cacheTiTms390PhysToVirt
    (
    void * address /* physical address */
    )
```

DESCRIPTION

This routine performs a 32-bit physical to 32-bit virtual address translation in the current context.

It works for only DRAM addresses of the first EMC.

It guesses likely virtual addresses, and checks its guesses with **VM_TRANSLATE**. A likely virtual address is the same as the physical address, or some multiple of 16M less. If any match, it succeeds. If all guesses are wrong, it fails.

RETURNS

The virtual address that maps to the physical address bits [31:0] argument, or NULL if it fails.

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 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*, **windsh**, *Tornado User's Guide: 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**

2

cdromFsDevCreate( )

NAME *cdromFsDevCreate*( ) – create a **cdromFsLib** device

SYNOPSIS

```
CDROM_VOL_DESC_ID cdromFsDevCreate
    (
    char *    devName, /* device name */
    BLK_DEV * pBlkDev  /* ptr to block device */
    )
```

DESCRIPTION This routine creates an instance of a **cdromFsLib** device in the I/O system. As input, this function requires a pointer to a **BLK_DEV** structure for the CD-ROM drive on which you want to create a **cdromFsLib** device. Thus, you should already have called *scsiBlkDevCreate*( ) prior to calling *cdfromFsDevCreate*( ).

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

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

cdromFsInit( )

NAME *cdromFsInit*( ) – initialize cdromFsLib

SYNOPSIS

```
STATUS cdromFsInit (void)
```

DESCRIPTION This routine initializes **cdromFsLib**. It must be called exactly once before calling any other routine in **cdromFsLib**.

ERRNO **S_cdromFsLib_ALREADY_INIT**

RETURNS OK or ERROR, if **cdromFsLib** has already been initialized.

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

cdromFsVolConfigShow( )

NAME
cdromFsVolConfigShow( ) – show the volume configuration information

SYNOPSIS
```
VOID cdromFsVolConfigShow
    (
    void * arg /* device name or CDROM_VOL_DESC * */
    )
```

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

RETURNS
N/A

SEE ALSO
cdromFsLib

ceil( )

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

SYNOPSIS
```
double ceil
    (
    double v /* value to find the ceiling of */
    )
```

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

INCLUDE FILES
math.h

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

SEE ALSO
ansiMath, **mathALib**

ceilf( )

NAME

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

SYNOPSIS

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

DESCRIPTION

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

INCLUDE FILES

math.h

RETURNS

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

SEE ALSO

mathALib

cfree( )

NAME

cfree( ) – free a block of memory

SYNOPSIS

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

DESCRIPTION

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

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

RETURNS

OK, or ERROR if the the block is invalid.

SEE ALSO

memLib

chdir( )

NAME

chdir( ) – set the current default path

SYNOPSIS

```
STATUS chdir
    (
    char * pathname /* name of the new default path */
    )
```

DESCRIPTION

This routine sets the default I/O path. All relative pathnames specified to the I/O system will be prepended with this pathname. This pathname must be an absolute pathname, i.e., *name* must begin with an existing device name.

RETURNS

OK, or ERROR if the first component of the pathname is not an existing device.

SEE ALSO

ioLib, *ioDefPathSet*( ), *ioDefPathGet*( ), *getcwd*( )

checkStack( )

NAME

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

SYNOPSIS

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

DESCRIPTION

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

```
-> checkStack tShell
    NAME         ENTRY        TID     SIZE   CUR  HIGH  MARGIN
------------ ------------ -------- ----- ----- ----- ------
tShell       _shell       23e1c78   9208   832  3632   5576
```

The maximum stack usage is determined by scanning down from the top of the stack for the first byte whose value is not 0xee. In VxWorks, when a task is spawned, all bytes of a task's stack are initialized to 0xee.

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

RETURNS N/A

SEE ALSO **usrLib**, ***taskSpawn*( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

---

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.

ERRNO

EINVAL

SEE ALSO

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

clock_gettime( )

NAME

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

SYNOPSIS

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

DESCRIPTION

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

RETURNS

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

ERRNO

EINVAL, **EFAULT**

SEE ALSO

clockLib

clock_setres( )

NAME

clock_setres( ) – set the clock resolution

SYNOPSIS

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

DESCRIPTION

This routine sets the clock resolution in the POSIX timers data structures. It does not affect the system clock or auxiliary clocks. This routine should be called to inform the 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.

ERRNO

EINVAL

SEE ALSO

clockLib, *clock_getres*( ), *sysClkRateSet*( )

clock_settime( )

NAME

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

SYNOPSIS

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

DESCRIPTION

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

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

ERRNO EINVAL

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

---

close( )

NAME ***close*( )** – close a file

SYNOPSIS

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

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

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

SEE ALSO **ioLib**

---

closedir( )

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

SYNOPSIS

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

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

RETURNS OK or ERROR.

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

connect( )

NAME

connect( ) – initiate a connection to a socket

SYNOPSIS

```
STATUS connect
    (
    int               s,      /* socket descriptor */
    struct sockaddr * name,   /* addr of socket to connect */
    int               namelen /* length of name, in bytes */
    )
```

DESCRIPTION

If *s* is a socket of type **SOCK_STREAM**, this routine establishes a virtual circuit between *s* and another socket specified by *name*. If *s* is of type **SOCK_DGRAM**, it permanently specifies the peer to which messages are sent. If *s* is of type **SOCK_RAW**, it specifies the raw socket upon which data is to be sent and received. The *name* parameter specifies the address of the other socket.

RETURNS

OK, or ERROR if the call fails.

SEE ALSO

sockLib

connectWithTimeout( )

NAME

connectWithTimeout( ) – try to connect 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 basically the same as ***connect*( )**, except that it lets users specify how long to keep trying to make the new connection.

If the *timeVal* is a NULL pointer, this routine acts exactly like ***connect*( )**. If *timeVal* is not NULL, it tries to establish a new connection for the duration of the time specified in *timeVal*. After that time, this routine reports a time-out error if the connection is not established.

RETURNS OK, or ERROR if a connection cannot be established.

SEE ALSO **sockLib**, ***connect*( )**

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 if there is an error copying from *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 failure handler (C++)

SYNOPSIS
```
extern void cplusCallNewHandler ()
```

DESCRIPTION
This function provides a procedural-interface to the new-handler. It can be used by user-defined new operators to call the current new-handler. This function is specific to VxWorks and may not be available in other C++ environments.

RETURNS
N/A

SEE ALSO
cplusLib

2

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
off	0
terse	1
complete	2

In complete mode, when C++ function names are printed, the class name (if any) is prefixed and the function's parameter type list is appended.

In terse mode, only the function name is printed. The class name and parameter type list are omitted.

In off mode, the function name is not demangled.

EXAMPLES The following example shows how one function name would be printed under each demangling mode:

Mode	Printed symbol
off	**_member__5classFPFl_PvPFPv_v**
terse	**_member**
complete	**foo::_member(void* (*)(long),void (*)(void*))**

RETURNS N/A

SEE ALSO **cplusLib**

2

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 the configuration macro **INCLUDE_CPLUS** is defined, ***cplusLibInit*( )** is called automatically from the root task, ***usrRoot*( )**, in **usrConfig.c**.

RETURNS OK or ERROR.

SEE ALSO **cplusLib**

cplusXtorSet( )

NAME *cplusXtorSet*( ) – change C++ static constructor calling strategy (C++)

SYNOPSIS

```
extern "C" void cplusXtorSet
    (
    int strategy
    )
```

DESCRIPTION This command sets the C++ static constructor calling strategy to *strategy*. The default strategy is 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*( )** and ***cpmInit*( )** to configure the Ethernet controller, and connects the interrupt vector *ivec*.

RETURNS

OK or ERROR.

SEE ALSO

if_cpm, **ifLib**, *Motorola MC68360 User's Manual* , *Motorola MPC821 and MPC860 User's Manual*

---

cpmStartOutput( )

NAME

***cpmStartOutput*( )** – output packet to network interface device

SYNOPSIS

```
#ifdef BSD43_DRIVER LOCAL void cpmStartOutput
    (
    int unit /* unit number */
    )
```

DESCRIPTION

***cpmStartOutput*( )** takes a packet from the network interface output queue, copies the mbuf chain into an interface buffer, and sends the packet over the interface. etherOutputHookRtns are supported.

Collision stats are collected in this routine from previously sent BDs. These BDs will not be examined until after the transmitter has cycled the ring, coming upon the BD after it has been sent. Thus, collision stat collection will be delayed a full cycle through the Tx ring.

This routine is called from several possible threads. Each one will be described below.

The first, and most common thread, is when a user task requests the transmission of data. Under BSD 4.3, this will cause ***cpmOutput*( )** to be called, which calls ***ether_output*( )**, which usually calls this routine. This routine will not be called if ***ether_output*( )** finds that our interface output queue is full. In this very rare case, the outgoing data will be thrown out. BSD 4.4 uses a slightly different model in which the generic ***ether_output*( )** routine is called directly, followed by a call to this routine.

The second thread is when a transmitter error occurs that causes a TXE event interrupt. This happens for the following errors: transmitter underrun, retry limit reached, late collision, and heartbeat error. The ISR sets the txStop flag to stop the transmitter until the errors are serviced. These events require a RESTART command of the transmitter, which occurs in the ***cpmTxRestart*( )** routine. After the transmitter is restarted, ***cpmTxRestart*( )** does a netJobAdd of ***cpmStartOutput*( )** to send any packets left in the interface output queue. Thus, the second thread executes in the context of ***netTask*( )**.

The third, and most unlikely, thread occurs when this routine is executing and it runs out of free Tx BDs. In this case, this routine turns on transmit interrupt and exits. When the next BD is actually sent, an interrupt occurs. The ISR does a netJobAdd of ***cpmStartOutput*( )** to continue sending packets left in the interface output queue. Once again, we find ourselves executing in the context of ***netTask*( )**.

RETURNS
N/A

SEE ALSO
if_cpm

---

cpsr( )

NAME
cpsr( ) – return the contents of the current processor status register (ARM)

SYNOPSIS

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

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

RETURNS
The contents of the current processor status register.

SEE ALSO
dbgArchLib, *VxWorks Programmer's Guide: Debugging*

creat( )

NAME

creat( ) – create a file

SYNOPSIS

```
int creat
    (
    const char * name, /* name of the file to create */
    int          flag  /* O_RDONLY, O_WRONLY, or O_RDWR */
    )
```

DESCRIPTION

This routine creates a file called *name* and opens it with a specified *flag*. This routine determines on which device to create the file; it then calls the create routine of the device driver to do most of the work. Therefore, much of what transpires is device/driver-dependent.

The parameter *flag* is set to **O_RDONLY** (0), **O_WRONLY** (1), or **O_RDWR** (2) for the duration of time the file is open. To create NFS files with a UNIX chmod-type file mode, call ***open*( )** with the file mode specified in the third argument.

NOTE

For more information about situations when there are no file descriptors available, see the manual entry for ***iosInit*( )**.

RETURNS

A file descriptor number, or ERROR if a filename is not specified, the device does not exist, no file descriptors are available, or the driver returns ERROR.

SEE ALSO

ioLib, *open*( )

cret( )

NAME

cret( ) – continue until the current subroutine returns

SYNOPSIS

```
STATUS cret
    (
    int task /* task to continue, 0 = default */
    )
```

DESCRIPTION

This routine places a breakpoint at the return address of the current subroutine of a specified task, then continues execution of that task.

To execute, enter:

```
-> cret [task]
```

If *task* is omitted or zero, the last task referenced is assumed.

When the breakpoint is hit, information about the task will be printed in the same format as in single-stepping. The breakpoint is automatically removed when hit, or if the task hits another breakpoint first.

RETURNS OK, or ERROR if there is no such task or the breakpoint table is full.

SEE ALSO **dbgLib**, ***so*( )**, *VxWorks Programmer's Guide: Shell*, **windsh**, *Tornado User's Guide: Shell*

---

csAttach( )

NAME ***csAttach*( )** – publish the **cs** network interface and initialize the driver.

SYNOPSIS

```
STATUS csAttach
    (
    int    unit,        /* unit number */
    int    ioAddr,      /* base IO address */
    int    intVector,   /* interrupt vector, or zero */
    int    intLevel,    /* interrupt level */
    int    memAddr,     /* base memory address */
    int    mediaType,   /* 0: Autodetect 1: AUI 2: BNC 3: RJ45 */
    int    configFlags, /* configuration flag */
    char * pEnetAddr    /* ethernet address */
    )
```

DESCRIPTION This routine is a major entry point to this network interface driver and is called only once per operating system reboot by the operating system startup code. This routine is called before the ***csInit*( )** routine.

This routine takes passed-in configuration parameters and parameters from the EEPROM and fills in the instance global variables in the **cs_softc** structure; these variables are later used by ***csChipInit*( )**. ***csAttach*( )** connects the interrupt handler ***csIntr*( )** to the specified interrupt vector, initializes the 8259 PIC, and resets the CS8900 chip.

Finally, ***csAttach*( )** calls the ***ether_attach*( )** routine to fill in the ifnet structure and attach this network interface driver to the system. The driver's main entry points (***csInit*( )**, ***csIoctl*( )**, ***csOutput*( )**, ***csReset*( )**) are made visable to the protocol stack.

See the reference page for **if_cs** for a detailed description of the configuration flags.

RETURNS OK or ERROR.

SEE ALSO **if_cs**

csShow( )

NAME

csShow( ) – shows statistics for the **cs** network interface

SYNOPSIS

```
void csShow
    (
    int  unit, /* interface unit */
    BOOL zap   /* zero totals */
    )
```

DESCRIPTION

This routine displays statistics about the **cs** 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_cs

ctime( )

NAME

ctime( ) – convert time in seconds into a string (ANSI)

SYNOPSIS

```
char * ctime
    (
    const time_t * timer /* calendar time in seconds */
    )
```

DESCRIPTION

This routine converts the calendar time pointed to by *timer* into local time in the form of a string. It is equivalent to:

```
asctime (localtime (timer));
```

This routine is not reentrant. For a reentrant version, see ***ctime_r*( )**.

INCLUDE FILES

time.h

RETURNS

The pointer returned by ***asctime*( )** with local broken-down time as the argument.

SEE ALSO

ansiTime, *asctime*( ), *localtime*( )

2

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 debugging help menu

SYNOPSIS

```
void dbgHelp (void)
```

DESCRIPTION This routine displays a summary of **dbgLib** utilities with a short description of each, similar to the following:

```
dbgHelp                         Print this list
dbgInit                         Install debug facilities
b                               Display breakpoints
b        addr[,task[,count]]    Set breakpoint
e        addr[,eventNo[,task[,func[,arg]]]]] Set eventpoint (WindView)
bd       addr[,task]            Delete breakpoint
bdall    [task]                 Delete all breakpoints
c        [task[,addr[,addr1]]] Continue from breakpoint
cret     [task]                 Continue to subroutine return
s        [task[,addr[,addr1]]] Single step
so       [task]                 Single step/step over subroutine
l        [adr[,nInst]]          List disassembled memory
tt       [task]                 Do stack trace on task
bh       addr[,access[,task[,count[,quiet]]]] set hardware breakpoint
                                (if supported by the architecture)
```

RETURNS N/A

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

dbgInit( )

NAME
dbgInit( ) – initialize the local debugging package

SYNOPSIS
```
STATUS dbgInit (void)
```

DESCRIPTION
This routine initializes the local debugging package and enables the basic breakpoint and single-step functions.

This routine also enables the shell abort function, CTRL-C.

NOTE
The debugging package should be initialized before any debugging routines are used. If the configuration macro **INCLUDE_DEBUG** is defined, ***dbgInit*( )** is called by the root task, ***usrRoot*( )**, in **usrConfig.c**.

RETURNS
OK, always.

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

dcattach( )

NAME
dcattach( ) – publish the **dc** network interface.

SYNOPSIS
```
STATUS dcattach
    (
    int    unit,       /* unit number */
    ULONG  devAdrs,    /* device 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.

The *unit* parameter is used to specify the device unit to initialize.

The *devAdrs* is used to specify the I/O address base of the device.

The *ivec* parameter is used to specify the interrupt vector associated with the device interrupt.

The *ilevel* parater is used to specify the level of the interrupt which the device would use.

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.

The *pciMemBase* parameter defines the main memory base as seen from PCI bus.

The *dcOpMode* parameter defines the mode in which the device should be operational.

BUGS

To zero out DEC 21x4x 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

---

dcCsrShow( )

NAME

***dcCsrShow*( )** – display dec 21040/21140 status registers 0 thru 15

SYNOPSIS

```
int dcCsrShow
    (
    int unit
    )
```

DESCRIPTION

Display the 16 registers of the DEC 21140 device on the console. Each register is printed in hexadecimal format.

RETURNS

N/A.

SEE ALSO

if_dc

dcReadAllRom( )

NAME *dcReadAllRom*( ) – read entire serial rom

SYNOPSIS

```
void dcReadAllRom
    (
    ULONG   devAdrs, /* device base I/O address */
    UCHAR * buffer,  /* destination bufferr */
    int     cnt      /* Amount to extract in bytes */
    )
```

DESCRIPTION Function to read all of serial rom and store the data in the data structure passed to the function. The count value will indicate how much of the serial rom to read. The routine with also swap the the bytes as the come in.

RETURNS N/A.

SEE ALSO **if_dc**

dcViewRom( )

NAME *dcViewRom*( ) – display lines of serial ROM for dec21140

SYNOPSIS

```
int dcViewRom
    (
    ULONG devAdrs, /* device base I/O address */
    UCHAR lineCnt, /* Serial ROM line Number */
    int   cnt      /* Amount to display */
    )
```

RETURNS Number of bytes displayed.

SEE ALSO **if_dc**

2

dec21x4xEndLoad( )

NAME *dec21x4xEndLoad*( ) – initialize the driver and device

SYNOPSIS

```
END_OBJ * dec21x4xEndLoad
    (
    char * initStr /* String to be parse by the driver. */
    )
```

DESCRIPTION This routine initializes the driver and the device to the operational state. All of the device specific parameters are passed in the initString.

This routine can be called in two modes. If it is called with an empty, but allocated string then it places the name of this device (i.e. dc) into the initString and returns 0.

If the string is allocated then the routine attempts to perform its load functionality.

RETURNS An END object pointer or NULL on error or 0 and the name of the device if the initString was NULL.

SEE ALSO **dec21x4xEnd**

dec21x40EndLoad( )

NAME *dec21x40EndLoad*( ) – initialize the driver and device

SYNOPSIS

```
END_OBJ* dec21x40EndLoad
    (
    char* initStr /* String to be parse by the driver. */
    )
```

DESCRIPTION This routine initializes the driver and the device to an operational state. All of the device-specific parameters are passed in the *initStr*. If this routine is called with an empty but allocated string, it puts the name of this device (that is, "dc") into the *initStr* and returns 0. If the string is allocated but not empty, this routine tries to load the device.

RETURNS An END object pointer or NULL on error.

SEE ALSO **dec21x40End**

dec21x40PhyLinkPoll( )

NAME *dec21x40PhyLinkPoll*( ) – Poll the PHY for link status

SYNOPSIS

```
UINT dec21x40PhyLinkPoll
    (
    DRV_CTRL * pDrvCtrl,
    UINT       linkTry
    )
```

RETURNS number of poll iterations remaining when link became active

SEE ALSO **dec21x40End**

dec21140SromWordRead( )

NAME *dec21140SromWordRead*( ) – read two bytes from the serial ROM

SYNOPSIS

```
USHORT dec21140SromWordRead
    (
    DRV_CTRL * pDrvCtrl,
    UCHAR      lineCnt   /* Serial ROM line Number */
    )
```

DESCRIPTION This routine returns the two bytes of information that is associated with it the specified ROM line number. This will later be used by the ***dec21140GetEthernetAdr*( )** function. It can also be used to review the ROM contents itself. The function must first send some initial bit patterns to the CSR9 that contains the Serial ROM Control bits. Then the line index into the ROM is evaluated bit-by-bit to program the ROM. The 2 bytes of data are extracted and processed into a normal pair of bytes.

RETURNS Value from ROM or ERROR.

SEE ALSO **dec21x40End**

devs( )

NAME *devs*( ) – list all system-known devices

SYNOPSIS

```
void devs (void)
```

DESCRIPTION This command displays a list of all devices known to the I/O system.

RETURNS N/A

SEE ALSO **usrLib**, ***iosDevShow*( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

dhcpcBind( )

NAME *dhcpcBind*( ) – obtain a set of network configuration parameters with DHCP

SYNOPSIS

```
STATUS dhcpcBind
    (
    void * pCookie, /* identifier returned by dhcpcInit() */
    BOOL   syncFlag /* synchronous or asynchronous execution */
    )
```

DESCRIPTION This routine initiates a DHCP negotiation according to the process described in RFC 1541. The *pCookie* argument contains the return value of an earlier ***dhcpcInit*( )** call and is used to identify a particular lease.

The *syncFlag* parameter specifies whether the DHCP negotiation started by this routine will execute synchronously or asynchronously. An asynchronous execution will return after starting the DHCP negotiation, but a synchronous execution will only return once the negotiation process completes.

When a new lease is established, any event hook provided for the lease will be called to process the configuration parameters. The hook is also called when the lease expires or the negotiation process fails. The results of an asynchronous DHCP negotiation are not available unless an event hook is installed.

If automatic configuration of the underlying network interface was specified during the lease initialization, this routine will prevent all higher-level protocols from accessing the underlying network interface used during the initial lease negotiation until that process is complete. In addition, any addressing information obtained will be applied to that network interface, which will remain disabled if the initial negotiation fails. Finally, the interface will be disabled if the lease expires.

NOTE If the DHCP client is used to obtain the VxWorks boot parameters, this routine is called automatically during system startup using the automatic reconfiguration. Therefore, any calls to this routine which use the network boot device for message transfer when the DHCP client was used at boot time must not request automatic reconfiguration during initialization. Otherwise, the resulting lease settings will conflict with the configuration maintained by the lease established during system startup.

RETURNS OK if routine completes, or ERROR otherwise.

ERRNO **S_dhcpcLib_BAD_COOKIE**, **S_dhcpcLib_NOT_INITIALIZED**, **S_dhcpcLib_BAD_OPTION**

SEE ALSO **dhcpcLib**

dhcpcBootBind( )

NAME *dhcpcBootBind*( ) – initialize the network with DHCP at boot time

SYNOPSIS

```
STATUS dhcpcBootBind (void)
```

DESCRIPTION This routine performs the client side of a DHCP negotiation according to RFC 1541. The negotiation uses the network device specified with the initialization call. The addressing information retrieved is applied to that network device. Because the boot image is replaced by the downloaded target image, the resulting lease cannot be renewed. Therefore, the minimum lease length specified by **DHCPC_MIN_LEASE** must be set so that the target image has sufficient time to download and begin monitoring the lease. This routine is called automatically by the boot program when **INCLUDE_DHCPC** is defined and the automatic configuration option is set in the boot flags.

RETURNS OK if negotiation is successful, or ERROR otherwise.

ERRNO N/A

SEE ALSO **dhcpcBootLib**

dhcpcBootInit( )

NAME

dhcpcBootInit( ) – set up the DHCP client parameters and data structures

SYNOPSIS

```
STATUS dhcpcBootInit
    (
    struct ifnet * pIf /* network device used by client */
    )
```

DESCRIPTION

This routine creates any necessary data structures and sets the client's option request list to retrieve a subnet mask and broadcast address for the network interface indicated by *pIf*. The routine is executed automatically by the boot program when **INCLUDE_DHCPC** is defined and the automatic configuration option is set in the boot flags. The network interface specified by *pIf* is used to transmit and receive all DHCP messages during the lease negotiation. That interface must be capable of sending broadcast messages. Currently, only Ethernet devices and the shared-memory network drivers are supported.

ERRNO

N/A

RETURNS

OK, or ERROR if could not initialize.

SEE ALSO

dhcpcBootLib

dhcpcBootOptionSet( )

NAME

dhcpcBootOptionSet( ) – add an option to the option request list

SYNOPSIS

```
STATUS dhcpcBootOptionSet
    (
    int    option, /* RFC 1533 tag of desired option */
    long   value,  /* numeric value for option */
    long   length, /* length of data (if any) or 0 if unused */
    char * pData   /* option data, or NULL if none */
    )
```

DESCRIPTION

This routine sets most client-to-server transmission options for a lease established by the boot program. The *option* parameter specifies an option tag as defined in RFC 1533 and the updates published in the Internet Draft of November 1996. The boot program automatically sets all necessary options for target configuration. This routine is only provided to support special circumstances in which additional options are necessary.

Any options requested with this routine may be retrieved after the runtime image has started. For a listing of defined aliases for the known option tags, see **dhcp/dhcp.h**. This routine cannot set the options associated with the following tags:

_DHCP_PAD_TAG
_DHCP_OPT_OVERLOAD_TAG
_DHCP_MSGTYPE_TAG
_DHCP_SERVER_ID_TAG
_DHCP_REQ_LIST_TAG
_DHCP_MAXMSGSIZE_TAG
_DHCP_END_TAG

Most options only require specification of the appropriate tag in the *option* parameter. In those cases, the ***dhcpcBootOptionSet( )*** call adds the specified option tag to the option request list, if possible. However, some options require additional information. The tags for these options are:

_DHCP_VENDOR_SPEC_TAG
_DHCP_REQUEST_IPADDR_TAG
_DHCP_LEASE_TIME_TAG
_DHCP_ERR_MSG_TAG
_DHCP_CLASS_ID_TAG
_DHCP_CLIENT_ID_TAG

The **_DHCP_LEASE_TIME_TAG** and **_DHCP_CLIENT_ID_TAG** options each require a *value* parameter. For **_DHCP_LEASE_TIME_TAG**, *value* specifies the desired lease length. For **_DHCP_CLIENT_ID_TAG**, *value* specifies the type for a type/value pair. No other options use this parameter.

The **_DHCP_VENDOR_SPEC_TAG**, **_DHCP_CLASS_ID_TAG**, and **_DHCP_CLIENT_ID_TAG**, tags each require a value for the *length* parameter to specify the number of bytes of data provided. No other options use this parameter.

Use the *data* parameter with the following option tags:

_DHCP_VENDOR_SPEC_TAG
: The *data* parameter points to a list of "length" bytes of options in the format specified by RFC 1533.

_DHCP_REQUEST_IPADDR_TAG
: The *data* parameter points to the string representation of the desired Internet address for the client.

_DHCP_ERRMSG_TAG
: The *data* parameter points to the error message to send to the server when releasing the current IP address.

_DHCP_CLASS_ID_TAG
: The *data* parameter points to *length* bytes used as the value for the vendor class identifier.

_DHCP_CLIENT_ID_TAG
The *data* parameter points to *length* bytes used as the value of a type/value pair.

-
The data parameter should be NULL for all other options.

NOTE With the exception of the **_DHCP_ERR_MSG_TAG** option, the DHCP specification forbids changing options after a lease has been established. Therefore, this routine should not be used after the ***dhcpcBootBind*( )** call. Changing any option other than the error message at that point could have unpredictable results.

RETURNS OK if option set successfully, or ERROR if option is invalid or storage failed.

ERRNO N/A

SEE ALSO **dhcpcBootLib**

---

dhcpcCacheHookAdd( )

NAME *dhcpcCacheHookAdd*( ) – add a routine to store and retrieve lease data

SYNOPSIS

```
STATUS dhcpcCacheHookAdd
    (
    FUNCPTR pCacheHookRtn /* routine to store/retrieve lease data */
    )
```

DESCRIPTION This routine adds a hook routine that is called at the bound state (to store the lease data) and during the **INIT_REBOOT** state (to re-use the parameters if the lease is still active). The calling sequence of the input hook routine is:

```
STATUS dhcpcCacheHookRtn
    (
    int command,                      /* requested cache operation */
    unsigned long *pTimeStamp,        /* lease timestamp data */
    int *pDataLen,                    /* length of data to access */
    char *pBuffer                     /* pointer to data buffer */
    )
```

The hook routine should return OK if the requested operation is completed successfully, or ERROR otherwise. All the supplied pointers reference memory locations that are reused upon return from the hook. The hook routine must copy the data elsewhere.

NOTE The setting of the cache hook routine during a ***dhcpcInit*( )** call is recorded and used by the resulting lease throughout its lifetime. Since the hook routine is intended to store a single lease record, a separate hook routine should be specified before the ***dhcpcInit*( )** call for each lease which will re-use its parameters across reboots.

IMPLEMENTATION The *command* parameter specifies one of the following operations:

DHCP_CACHE_WRITE
Save the indicated data. The write operation must preserve the value referenced by *pTimeStamp* and the contents of *pBuffer*. The *pDataLen* parameter indicates the number of bytes in that buffer.

DHCP_CACHE_READ
Restore the saved data. The read operation must copy the data from the most recent write operation into the location indicated by *pBuffer*, set the contents of *pDataLen* to the amount of data provided, and store the corresponding timestamp value in *pTimeStamp*.

-

The read operation has very specific requirements. On entry, the value referenced by *pDataLen* indicates the maximum buffer size available at *pBuffer*. If the amount of data stored by the previous write exceeds this value, the operation must return ERROR. A read must also return ERROR if the saved timestamp value is 0. Finally, the read operation must return ERROR if it is unable to retrieve all the data stored by the write operation or if the previous write was unsuccessful.

DHCP_CACHE_ERASE
Ignore all stored data. Following this operation, subsequent read operations must return ERROR until new data is written. All parameters except *command* are NULL.

RETURNS OK, always.

ERRNO N/A

SEE ALSO **dhcpcLib**

dhcpcCacheHookDelete( )

NAME *dhcpcCacheHookDelete*( ) – delete a lease data storage routine

SYNOPSIS
```
STATUS dhcpcCacheHookDelete (void)
```

DESCRIPTION This routine deletes the hook used to store lease data, preventing re-use of the configuration parameters across system reboots for all subsequent lease attempts.

Currently active leases will continue to use the routine specified before the lease initialization.

RETURNS OK, always.

ERRNO N/A

SEE ALSO **dhcpcLib**

---

dhcpcEventHookAdd( )

NAME *dhcpcEventHookAdd*( ) – add a routine to handle configuration parameters

SYNOPSIS

```
STATUS dhcpcEventHookAdd
    (
    void *  pCookie,   /* identifier returned by dhcpcInit() */
    FUNCPTR pEventHook /* routine to handle lease parameters */
    )
```

DESCRIPTION This routine installs a hook routine to handle changes in the configuration parameters provided for the lease indicated by *pCookie*. The hook provides an alternate configuration method for DHCP leases and uses the following interface:

```
void dhcpcEventHookRtn
    (
    int       leaseEvent,     /* new or expired parameters */
    void *    pCookie         /* lease identifier from dhcpcInit() */
    )
```

The routine is called with the *leaseEvent* parameter set to **DHCPC_LEASE_NEWwhenever** a lease is successfully established. The **DHCPC_LEASE_NEW** event does not occur when a lease is renewed by the same DHCP server, since the parameters do not change in that case. However, it does occur if the client rebinds to a different DHCP server. The **DHCPC_LEASE_INVALID** event indicates that the configuration parameters for the corresponding lease may no longer be used. That event occurs when a lease expires or a renewal or verification attempt fails, and coincides with re-entry into the initial state of the negotiation process.

If the lease initialization specified automatic configuration of the corresponding network interface, any installed hook routine will be invoked after the new address information is applied.

RETURNS OK if notification hook added, or ERROR otherwise.

ERRNO **S_dhcpcLib_BAD_COOKIE, S_dhcpcLib_NOT_INITIALIZED**

SEE ALSO **dhcpcLib**

dhcpcEventHookDelete( )

NAME *dhcpcEventHookDelete*( ) – remove the configuration parameters handler

SYNOPSIS

```
STATUS dhcpcEventHookDelete
    (
    void * pCookie /* identifier returned by dhcpcInit() */
    )
```

DESCRIPTION This routine removes the hook routine that handled changes in the configuration parameters for the lease indicated by *pCookie*. If the lease initialization specified automatic configuration of the corresponding network interface, the assigned address could change without warning after this routine is executed.

RETURNS OK if notification hook removed, or ERROR otherwise.

ERRNO **S_dhcpcLib_BAD_COOKIE, S_dhcpcLib_NOT_INITIALIZED**

SEE ALSO **dhcpcLib**

dhcpcInit( )

NAME *dhcpcInit*( ) – assign network interface and setup lease request

SYNOPSIS

```
void * dhcpcInit
    (
    struct ifnet * pIf,       /* network device used by client */
    BOOL           autoConfig /* reconfigure network device? */
    )
```

DESCRIPTION This routine creates the data structures used to obtain a set of parameters with DHCP and must be called before each attempt at establishing a DHCP lease, but after the ***dhcpcLibInit*( )** routine has initialized the global data structures. The *pIf* argument indicates the network device which will be used for transmission and reception of DHCP messages during the lifetime of the lease. If the *autoConfig* parameter is set to TRUE, any

2

address information obtained will automatically be applied to that interface. The specified interface must access a device capable of sending broadcast messages. Currently, only Ethernet devices and the shared-memory network drivers are supported.

The routine also uses the *autoConfig* parameter to select the default option request list for a lease. If set to FALSE, no specific lease options are requested since any configuration parameters obtained are not intended for the underlying network device. In that case, any specific options required may be added to the request list at any time before the corresponding ***dhcpcBind*( )** call. If *autoConfig* is TRUE, this routine sets the configuration parameters to request the minimal address information (subnet mask and broadcast address) necessary for reconfiguring the network device specified by *pIf*.

The internal lease identifier returned by this routine must be used in subsequent calls to the DHCP client library.

NOTE

This routine is called automatically during system startup if the DHCP client was used to obtain the VxWorks boot parameters. The resulting lease will always reconfigure the network boot device. Therefore, any further calls to this routine which specify the network boot device for use in obtaining additional DHCP leases must set *autoConfig* to FALSE. Otherwise, that device will be unable to maintain a stable configuration. The global variable pDhcpcBootCookie provides access to the configuration parameters for any DHCP lease created during system startup.

RETURNS

Lease handle for later use, or NULL if lease setup fails.

ERRNO

S_dhcpcLib_NOT_INITIALIZED, **S_dhcpcLib_NO_DEVICE**, **S_dhcpcLib_BAD_OPTION**, **S_dhcpcLib_MAX_LEASES_REACHED**, **S_dhcpcLib_MEM_ERROR**

SEE ALSO

dhcpcLib, *dhcpcOptionSet***( )**, *dhcpcEventHookAdd***( )**

---

dhcpcLibInit( )

NAME

*dhcpcLibInit***( )** – DHCP client library initialization

SYNOPSIS

```
STATUS dhcpcLibInit
    (
    int serverPort,   /* port used by DHCP servers (default 67) */
    int clientPort,   /* port used by DHCP clients (default 68) */
    int maxLeases,    /* max number of simultaneous leases allowed */
    int offerTimeout, /* interval to get additional DHCP offers */
    int defaultLease, /* default value for requested lease length */
    int minLease      /* minimum accepted lease length */
    )
```

DESCRIPTION
This routine creates and initializes the global data structures used by the DHCP client library to maintain multiple leases, up to the limit specified by the *maxLeases* parameter. Every subsequent lease attempt will collect additional DHCP offers until the interval specified by *offerTimeout*expires and will request the lease duration indicated by *defaultLease*. This routine must be called before calling any other library routines. The routine is called automatically if **INCLUDE_DHCPC** is defined at the time the system is built and assigns the global lease settings to the values specified by **DHCPC_SPORT**, **DHCPC_CPORT**, **DHCPC_MAX_LEASES**, **DHCPC_DEFAULT_LEASE**, and **DHCPC_OFFER_TIMEOUT**.

RETURNS
OK, or ERROR if initialization fails.

ERRNO
S_dhcpcLib_MEM_ERROR

SEE ALSO
dhcpcLib

dhcpcOptionGet( )

NAME
dhcpcOptionGet( ) – retrieve an option provided to a client and store in a buffer

SYNOPSIS

```
STATUS dhcpcOptionGet
    (
    void * pCookie, /* identifier returned by dhcpcInit() */
    int    option,  /* RFC 1533 option tag */
    int *  pLength, /* size of provided buffer and data returned */
    char * pBuf     /* location for option data */
    )
```

DESCRIPTION
This routine retrieves the data for the specified *option*, if present for the lease indicated by *pCookie*. The data is stored in the provided buffer, whose length must be specified. If the *option* is found, the amount of data available is stored in the location referenced by the *pLength* parameter. The option is not available if the DHCP client is not in the bound state or if the server did not provide it. After returning, the provided buffer may contain IP addresses stored in network byte order. All other numeric values are stored in host byte order. See RFC 1533 for specific details on the data retrieved.

RETURNS
OK if option available, or ERROR otherwise.

ERRNO
S_dhcpcLib_BAD_COOKIE, **S_dhcpcLib_NOT_INITIALIZED**, **S_dhcpcLib_NOT_BOUND**, **S_dhcpcLib_OPTION_NOT_PRESENT**

SEE ALSO
dhcpcLib, *dhcpcOptionSet*( )

2

dhcpcOptionSet( )

NAME

dhcpcOptionSet( ) – add an option to the option request list

SYNOPSIS

```
STATUS dhcpcOptionSet
    (
    void * pCookie, /* identifier returned by dhcpcInit() */
    int    option,  /* RFC 1533 tag of desired option */
    long   value,   /* numeric value for option */
    long   length,  /* length of data (if any) or 0 if unused */
    char * pData    /* option data, or NULL if none */
    )
```

DESCRIPTION

This routine sets most client-to-server transmission options for the lease indicated by the *pCookie* parameter. The *option* parameter specifies an option tag as defined in RFC 1533 and the updates published in the Internet Draft of November 1996. For a listing of defined aliases for the known option tags, see **dhcp/dhcp.h**. This routine cannot set the options associated with the following tags:

_DHCP_PAD_TAG
_DHCP_OPT_OVERLOAD_TAG
_DHCP_MSGTYPE_TAG
_DHCP_SERVER_ID_TAG
_DHCP_REQ_LIST_TAG
_DHCP_MAXMSGSIZE_TAG
_DHCP_END_TAG

Most options only require specification of the appropriate tag in the *option* parameter. In those cases, the *dhcpcOptionSet*( ) call adds the specified option tag to the option request list, if possible. However, some options require additional information. The tags for these options are:

_DHCP_VENDOR_SPEC_TAG
_DHCP_REQUEST_IPADDR_TAG
_DHCP_LEASE_TIME_TAG
_DHCP_ERRMSG_TAG
_DHCP_CLASS_ID_TAG
_DHCP_CLIENT_ID_TAG

The **_DHCP_LEASE_TIME_TAG** and **_DHCP_CLIENT_ID_TAG** options each use the *value* parameter. For **_DHCP_LEASE_TIME_TAG**, *value* specifies the desired lease length. For **_DHCP_CLIENT_ID_TAG**, *value* specifies the type for a type/value pair. No other options use this parameter.

The **_DHCP_VENDOR_SPEC_TAG**, **_DHCP_CLASS_ID_TAG** and **_DHCP_CLIENT_ID_TAG** tags each require a value for the *length* parameter to specify the number of bytes of data provided. No other options use this parameter.

The *pData* parameter is relevant to the following option tags:

_DHCP_VENDOR_SPEC_TAG
The *pData* parameter references a list of *length* bytes of options in the format specified by RFC 1533.

_DHCP_REQUEST_IPADDR_TAG
The *pData* parameter indicates the string representation of the desired Internet address for the client in dot notation.

_DHCP_ERRMSG_TAG
The *pData* parameter indicates the error message to send to the server when releasing the current IP address. That location must be valid until the release is completed, since the message is not copied or stored in any way.

_DHCP_CLASS_ID_TAG
The *pData* parameter references *length* bytes used as the value for the vendor class identifier.

_DHCP_CLIENT_ID_TAG
The *pData* parameter references *length* bytes used as the value of a type/value pair.

-
The *pData* parameter is not used by any other options.

NOTE
With the exception of the **_DHCP_ERRMSG_TAG** option, the DHCP specification forbids changing options after a lease has been established. Therefore, this routine should not be used after the ***dhcpcBind*( )** call. Changing any option other than the error message at that point could have unpredictable results.

RETURNS
OK if the option was set successfully, or ERROR if the option is invalid or storage failed.

ERRNO
S_dhcpcLib_BAD_OPTION, **S_dhcpcLib_OPTION_NOT_STORED**

SEE ALSO
dhcpcLib

dhcpcParamsGet( )

NAME

dhcpcParamsGet( ) – retrieve current configuration parameters

SYNOPSIS

```
STATUS dhcpcParamsGet
    (
    void *              pCookie,   /* identifier returned by dhcpcInit() */
    struct dhcp_param * pParamList /* requested parameters */
    )
```

DESCRIPTION

This routine copies the current configuration parameters for the lease specified by the *pCookie* argument to the user-supplied structure. That structure, defined in **dhcp/dhcpc.h**, should contain non-NULL pointers to indicate the parameters of interest. All other values within the structure must be set to 0 before calling the routine. The requested information is only retrieved if the specified lease is in the bound state and knows that its parameters are good.

Many of the parameters within the user-supplied structure use one of the following secondary data types: struct in_addrs, struct u_shorts, and struct vendor_list. Each of those structures accepts a length designation and a data pointer. For the first two data types, the **num** member indicates the size of the buffer in terms of the number of underlying elements. For example, the **STATIC_ROUTE** option returns one or more IP address pairs. So, setting the **num** member to 2 in the static_route entry would indicate that the corresponding buffer contained 16 bytes. By contrast, the **len** member in the struct **vendor_list** data type consists of the buffer size, in bytes. See RFC 1533 for specific details on the types of data for each option.

On return, each of the length designators are set to indicate the amount of data returned. For instance, the **num** member in the static_route entry could be set to 1 to indicate that only one IP address pair of 8 bytes was available.

RETURNS

OK if in bound state, or ERROR otherwise.

ERRNO

S_dhcpcLib_BAD_COOKIE, **S_dhcpcLib_NOT_INITIALIZED**, **S_dhcpcLib_NOT_BOUND**

SEE ALSO

dhcpcLib

dhcpcParamsShow( )

NAME *dhcpcParamsShow*( ) – display current lease parameters

SYNOPSIS

```
STATUS dhcpcParamsShow
    (
    void * pCookie /* identifier returned by dhcpcInit() */
    )
```

DESCRIPTION This routine prints all lease parameters for the lease identified by *pCookie*. It has no effect if the indicated lease is not currently active.

RETURNS OK, or ERROR if lease identifier unknown.

ERRNO **S_dhcpcLib_BAD_COOKIE**

SEE ALSO **dhcpcShow**

dhcpcRelease( )

NAME *dhcpcRelease*( ) – relinquish specified lease

SYNOPSIS

```
STATUS dhcpcRelease
    (
    void * pCookie /* identifier returned by dhcpcInit() */
    )
```

DESCRIPTION This routine schedules the lease identified by the *pCookie* parameter for immediate release, regardless of time remaining, and removes all the associated data structures. After the release completes, a new call to ***dhcpcInit*( )** is required before attempting another lease.

NOTE This routine will disable the underlying network interface if automatic configuration was requested. This may occur without warning if no event hook is installed.

RETURNS OK if release scheduled, or ERROR otherwise.

ERRNO **S_dhcpcLib_BAD_COOKIE, S_dhcpcLib_NOT_INITIALIZED**

SEE ALSO **dhcpcLib**

dhcpcServerGet( )

NAME *dhcpcServerGet*( ) – retrieve the current DHCP server

SYNOPSIS

```
STATUS dhcpcServerGet
    (
    void *           pCookie,    /* identifier returned by dhcpcInit() */
    struct in_addr * pServerAddr /* location for address of server */
    )
```

DESCRIPTION This routine returns the DHCP server that supplied the configuration parameters for the lease specified by the *pCookie* argument. This information is available only if the lease is in the bound state.

RETURNS OK if in bound state and server available, or ERROR otherwise.

ERRNO **S_dhcpcLib_BAD_COOKIE**, **S_dhcpcLib_NOT_INITIALIZED**, **S_dhcpcLib_NOT_BOUND**

SEE ALSO **dhcpcLib**

dhcpcServerShow( )

NAME *dhcpcServerShow*( ) – display current DHCP server

SYNOPSIS

```
STATUS dhcpcServerShow
    (
    void * pCookie /* identifier returned by dhcpcInit() */
    )
```

DESCRIPTION This routine prints the IP address of the DHCP server that provided the parameters for the lease identified by *pCookie*. It has no effect if the indicated lease is not currently active.

RETURNS OK, or ERROR if lease identifier unknown.

ERRNO **S_dhcpcLib_BAD_COOKIE**

SEE ALSO **dhcpcShow**

dhcpcShowInit( )

NAME *dhcpcShowInit*( ) – initialize the DHCP show facility

SYNOPSIS

```
void dhcpcShowInit (void)
```

DESCRIPTION This routine links the DHCP show facility into the VxWorks system image. It is called from **usrNetwork.c** automatically if **INCLUDE_DHCP** and **INCLUDE_NET_SHOW** are defined at the time the image is constructed.

SEE ALSO **dhcpcShow**

dhcpcShutdown( )

NAME *dhcpcShutdown*( ) – disable DHCP client library

SYNOPSIS

```
STATUS dhcpcShutdown (void)
```

DESCRIPTION This routine schedules the lease monitor task to clean up memory and exit, after releasing all currently active leases. The network boot device will be disabled if the DHCP client was used to obtain the VxWorks boot parameters and the resulting lease is still active. Any other interfaces using the addressing information from leases set for automatic configuration will also be disabled. Notification of a disabled interface will not occur unless an event hook has been installed. After the processing started by this request completes, the DHCP client library is unavailable until restarted with the ***dhcpcLibInit*( )** routine.

RETURNS OK if shutdown scheduled, or ERROR otherwise.

ERRNO **S_dhcpcLib_NOT_INITIALIZED**

SEE ALSO **dhcpcLib**

2

dhcpcTimerGet( )

NAME

dhcpcTimerGet( ) – retrieve current lease timers

SYNOPSIS

```
STATUS dhcpcTimerGet
    (
    void * pCookie, /* identifier returned by dhcpcInit() */
    int *  pT1,     /* time until lease renewal */
    int *  pT2      /* time until lease rebinding */
    )
```

DESCRIPTION

This routine returns the number of clock ticks remaining on the timers governing the DHCP lease specified by the *pCookie* argument. This information is only available if the lease is in the bound state. Therefore, this routine will return ERROR if a BOOTP reply was accepted.

RETURNS

OK if in bound state and values available, or ERROR otherwise.

ERRNO

S_dhcpcLib_BAD_COOKIE, **S_dhcpcLib_NOT_INITIALIZED**, **S_dhcpcLib_NOT_BOUND**, **S_dhcpcLib_OPTION_NOT_PRESENT**, **S_dhcpcLib_TIMER_ERROR**

SEE ALSO

dhcpcLib

dhcpcTimersShow( )

NAME

dhcpcTimersShow( ) – display current lease timers

SYNOPSIS

```
STATUS dhcpcTimersShow
    (
    void * pCookie /* identifier returned by dhcpcInit() */
    )
```

DESCRIPTION

This routine prints the time remaining with each of the DHCP lease timers for the lease identified by *pCookie*. It has no effect if the indicated lease is not currently active.

RETURNS

OK if show routine completes, or ERROR otherwise.

ERRNO

S_dhcpcLib_BAD_COOKIE

SEE ALSO

dhcpcShow

dhcpcVerify( )

NAME

dhcpcVerify( ) – renew an established lease

SYNOPSIS

```
STATUS dhcpcVerify
    (
    void * pCookie /* identifier returned by dhcpcInit() */
    )
```

DESCRIPTION

This routine schedules the lease identified by the *pCookie* parameter for immediate renewal according to the process described in RFC 1541. If the renewal is unsuccessful, the lease negotiation process restarts. The routine is valid as long as the lease is currently active. The routine is also called automatically in response to a ***dhcpcBind*( )** call for an existing lease.

NOTE

This routine will disable the underlying network interface if the verification fails and automatic configuration was requested. This may occur without warning if no event hook is installed.

RETURNS

OK if verification scheduled, or ERROR otherwise.

ERRNO

S_dhcpcLib_BAD_COOKIE, **S_dhcpcLib_NOT_INITIALIZED**, **S_dhcpcLib_NOT_BOUND**

SEE ALSO

dhcpcLib

dhcpsAddressHookAdd( )

NAME

dhcpsAddressHookAdd( ) – assign a permanent address storage hook for the server

SYNOPSIS

```
STATUS dhcpsAddressHookAdd
    (
    FUNCPTR pCacheHookRtn /* routine to store/retrieve lease entries */
    )
```

DESCRIPTION

This routine allows the server to access some form of permanent storage to preserve additional address entries across restarts. This routine is not required, but leases using unsaved addresses are not renewed. The only argument provided is the name of a function with the following interface:

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

The first parameter of this storage routine specifies one of the following operations:

DHCPS_STORAGE_START
DHCPS_STORAGE_READ
DHCPS_STORAGE_WRITE
DHCPS_STORAGE_STOP

In response to a START, the storage routine should prepare to return data or overwrite data provided by earlier WRITE operations. For a WRITE, the storage routine must save the contents of the four buffers to permanent storage. Those buffers contain the NULL-terminated strings received by the ***dhcpsLeaseEntryAdd*( )** routine. For a READ, the storage routine should copy previously stored data (as NULL-terminated strings) into the provided buffers in the order received by earlier WRITE operations. For a STOP, the storage routine should do any necessary cleanup. After a STOP, the storage routine should return an ERROR for all operations except START.

The storage routine should return OK if successful, ERROR otherwise.

Note that, unlike the lease storage routine, there is no CLEAR operation.

Before the server is initialized, VxWorks calls this routine automatically passing in the function named in **DHCPS_ADDRESS_HOOK**.

RETURNS OK, or ERROR if function pointer is NULL.

ERRNO N/A

SEE ALSO **dhcpsLib**

---

dhcpsInit( )

NAME *dhcpsInit*( ) – set up the DHCP server parameters and data structures

SYNOPSIS

```
STATUS dhcpsInit
    (
    struct ifnet * *   ppIf,       /* network devices used by server */
    int                numDev,     /* number of devices */
    DHCPS_LEASE_DESC * pLeasePool, /* table of lease data */
    int                poolSize,   /* size of data table */
    DHCPS_RELAY_DESC * pRelayTbl,  /* table of relay agent data */
    int                relaySize,  /* size of relay agent table */
    DHCP_TARGET_DESC * pTargetTbl, /* table of receiving DHCP servers */
    int                targetSize
    )
```

DESCRIPTION
This routine creates the necessary data structures, builds the server address pool, retrieves any lease or address information from permanent storage through the user-provided hooks, and initializes the network interfaces for monitoring. It is called at system startup if **INCLUDE_DHCPS** is defined at the time the VxWorks image is built.

RETURNS
OK, or ERROR if could not initialize.

SEE ALSO
dhcpsLib

---

dhcpsLeaseEntryAdd( )

NAME
dhcpsLeaseEntryAdd( ) – add another entry to the address pool

SYNOPSIS

```
STATUS dhcpsLeaseEntryAdd
    (
    char * pName,    /* name of lease entry */
    char * pStartIp, /* first IP address to assign */
    char * pEndIp,   /* last IP address in assignment range */
    char * pParams   /* formatted string of lease parameters */
    )
```

DESCRIPTION
This routine allows the user to add new entries to the address pool without rebuilding the VxWorks image. The routine requires a unique entry name of up to eight characters, starting and ending IP addresses, and a colon-separated list of parameters. Possible values for the parameters are listed in the reference entry for **dhcpsLib**. The parameters also determine the type of lease, which the server uses to determine priority when assigning lease addresses. For examples of the possible lease types, see the reference entry for **dhcpsLib**.

RETURNS
OK if entry read successfully, or ERROR otherwise.

ERRNO
N/A

SEE ALSO
dhcpsLib

dhcpsLeaseHookAdd( )

NAME

dhcpsLeaseHookAdd( ) – assign a permanent lease storage hook for the server

SYNOPSIS

```
STATUS dhcpsLeaseHookAdd
    (
    FUNCPTR pCacheHookRtn /* routine to store/retrieve lease records */
    )
```

DESCRIPTION

This routine allows the server to access some form of permanent storage that it can use to store current lease information across restarts. The only argument to *dhcpsLeaseHookAdd*( ) is a pointer to a storage routine with the following interface:

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

The first parameter of the storage routine specifies one of the following operations:

DHCPS_STORAGE_START
DHCPS_STORAGE_READ
DHCPS_STORAGE_WRITE
DHCPS_STORAGE_STOP
DHCPS_STORAGE_CLEAR

In response to START, the storage routine should prepare to return data or overwrite data provided by earlier WRITEs. For a WRITE the storage routine must save the contents of the buffer to permanent storage. For a READ, the storage routine should copy data previously stored into the provided buffer as a NULL-terminated string in FIFO order. For a CLEAR, the storage routine should discard currently stored data. After a CLEAR, the READ operation must return ERROR until additional data is stored. For a STOP, the storage routine must handle cleanup. After a STOP, calls to the storage routine must return error until a START is received. Each of these operations must return OK if successful, or ERROR otherwise.

Before the server is initialized, VxWorks automatically calls *dhcpsLeaseHookAdd*( ), passing in the routine name defined by **DHCPS_LEASE_HOOK**.

RETURNS

OK, or ERROR if routine is NULL.

ERRNO

N/A

SEE ALSO

dhcpsLib

*difftime***( )**

NAME
*difftime***( )** – compute the difference between two calendar times (ANSI)

SYNOPSIS
```
double difftime
    (
    time_t time1, /* later time, in seconds */
    time_t time0  /* earlier time, in seconds */
    )
```

DESCRIPTION
This routine computes the difference between two calendar times: *time1* - *time0*.

INCLUDE FILES
time.h

RETURNS
The time difference in seconds, expressed as a double.

SEE ALSO
ansiTime

*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*. This routine is not reentrant. For a reentrant version, see ***div_r*( )**.

INCLUDE FILES

stdlib.h

RETURNS

A structure of type **div_t**, containing both the quotient and the remainder.

SEE ALSO

ansiStdlib

div_r( )

NAME

div_r( ) – compute a quotient and remainder (reentrant)

SYNOPSIS

```
void div_r
    (
    int     numer,       /* numerator */
    int     denom,       /* denominator */
    div_t * divStructPtr /* div_t structure */
    )
```

DESCRIPTION

This routine computes the quotient and remainder of *numer*/*denom*. The quotient and remainder are stored in the **div_t** structure pointed to by *divStructPtr*.

This routine is the reentrant version of ***div*( )**.

INCLUDE FILES

stdlib.h

RETURNS

N/A

SEE ALSO

ansiStdlib

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*( )

2

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**.

This initialization is enabled when the configuration macro **INCLUDE_DOSFSis** defined; ***dosFsInit*( )** is then 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.

ERRNO

S_dosFsLib_INVALID_PARAMETER

SEE ALSO

dosFsLib, *dosFsDevInit*( )

2

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 has been called, the next attempt to use the volume will result in an attempted remount.

This routine may also be invoked by calling ***ioctl*( )** with **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*( )

---

dummyCallback( )

NAME
dummyCallback( ) – dummy callback routine

SYNOPSIS
```
STATUS dummyCallback (void)
```

RETURNS
ERROR.

SEE ALSO
winSio

---

dummyCallback( )

NAME
dummyCallback( ) – dummy callback routine.

SYNOPSIS
```
STATUS dummyCallback (void)
```

RETURNS
ERROR.

SEE ALSO
unixSio

---

e( )

NAME
e( ) – set or display eventpoints (WindView)

SYNOPSIS
```
STATUS e
    (
    INSTR * addr,         /* where to set eventpoint; 0 means display all */
    event_t eventId,      /* event ID */
    int     taskNameOrId, /* task affected; 0 means all tasks */
    FUNCPTR evtRtn,       /* function to invoke; NULL means no function */
    int     arg           /* argument to be passed to evtRtn */
    )
```

DESCRIPTION

This routine sets "eventpoints"--that is, breakpoint-like instrumentation markers that can be inserted in code to generate and log an event for use with WindView. Event logging must be enabled with ***wvEvtLogEnable( )*** for the eventpoint to be logged.

eventId selects the evenpoint number that will be logged: it is in the user event ID range (0-25536).

If *addr* is NULL, then all eventpoints and breakpoints are displayed. If *taskNameOrId* is 0, then this event is logged in all tasks. The *evtRtn* routine is called when this eventpoint is hit. If *evtRtn*returns OK, then the eventpoint is logged; otherwise, it is ignored. If *evtRtn* is a NULL pointer, then the eventpoint is always logged.

Eventpoints are exactly like breakpoints (which are set with the ***b( )*** command) except in how the system responds when the eventpoint is hit. An eventpoint typically records an event and continues immediately (if *evtRtn* is supplied, this behavior may be different). Eventpoints cannot be used at interrupt level.

To delete an eventpoint, use ***bd( )***.

RETURNS

OK, or ERROR if *addr* is odd or nonexistent in memory, or if the breakpoint table is full.

SEE ALSO

dbgLib, ***wvEvent( )***

---

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*

2

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**

eexTxStartup( )

NAME *eexTxStartup*( ) – start output on the chip

SYNOPSIS

```
#ifdef BSD43_DRIVER static void eexTxStartup
    (
    int unit
    )
```

DESCRIPTION Looks for any action on the queue, and begins output if there is anything there. This routine is called from several possible threads. Each will be described below.

The first, and most common thread, is when a user task requests the transmission of data. Under BSD 4.3, this will cause ***eexOutput*( )** to be called, which will cause ***ether_output*( )** to be called, which will cause this routine to be called (usually). This routine will not be called if ***ether_output*( )** finds that our interface output queue is full. In this case, the outgoing data will be thrown out. BSD 4.4 uses a slightly different model in which the generic ***ether_output*( )** routine is called directly, followed by a call to this routine.

The second, and most obscure thread, is when the reception of certain packets causes an immediate (attempted) response. For example, ICMP echo packets (ping), and ICMP "no listener on that port" notifications. All functions in this driver that handle the reception side are executed in the context of ***netTask*( )**. Always. So, in the case being discussed, ***netTask*( )** will receive these certain packets, cause IP to be stimulated, and cause the generation of a response to be sent. We then find ourselves following the thread explained in the second example, with the important distinction that the context is that of ***netTask*( )**.

The third thread occurs when this routine runs out of TFDs and returns. If this occurs when our output queue is not empty, this routine would typically not get called again until new output was requested. Even worse, if the output queue was also full, this routine would never get called again and we would have a lock state. It DOES happen. To guard against this, the transmit clean-up handler detects the out-of-TFDs state and calls this function. The clean-up handler also runs from netTask.

Note that this function is ALWAYS called between an ***splnet*( )** and an ***splx*( )**. This is true because ***netTask*( )**, and ***ether_output*( )** take care of this when calling this function. Therefore, no calls to these spl functions are needed anywhere in this output thread.

SEE ALSO **if_eex**

---

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*

2

ei82596EndLoad( )

NAME

ei82596EndLoad( ) – initialize the driver and device

SYNOPSIS

```
END_OBJ *ei82596EndLoad
    (
    char * initString /* parameter string */
    )
```

DESCRIPTION

This routine initializes both driver and device to an operational state using the device-specific values specified by *initString*. The *initString* parameter expects an ordered list of colon-separated values.

The format of the *initString* is: *unit*:*ivec*:*sysbus*:*memBase*:*nTfds*:*nRfds*

unit
: Specifies the unit number for this device.

ivec
: This is the interrupt vector number of the hardware interrupt generated by this Ethernet device. The driver uses ***intConnect*( )** to attach an interrupt handler for this interrupt.

sysbus
: Passes in 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.

memBase
: Informs the driver about the shared memory region. 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. If this parameter is set to the constant "NONE", this routine tries 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, 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, this routine uses ***cacheDmaMalloc*( )** to obtain some non-cacheable memory. The attributes of this memory are checked, and, if the memory is not both read- and write-coherent, this routine aborts.

RETURNS

An END object pointer or NULL.

SEE ALSO

ei82596End, **ifLib**, *Intel 82596 User's Manual*

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 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 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*

2

eihkattach( )

NAME

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

SYNOPSIS

```
STATUS eihkattach
    (
    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 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 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_eihk, **ifLib**, *Intel 82596 User's Manual*

eiInt( )

NAME

eiInt( ) – entry point for handling interrupts from the 82596

SYNOPSIS

```
void eiInt
    (
    DRV_CTRL * pDrvCtrl
    )
```

DESCRIPTION

The interrupting events are acknowledged to the device, so that the device will deassert its interrupt signal. The amount of work done here is kept to a minimum; the bulk of the work is defered to the netTask. Several flags are used here to synchronize with task level code and eliminate races.

SEE ALSO

if_eihk

eiTxStartup( )

NAME

eiTxStartup( ) – start output on the chip

SYNOPSIS

```
#ifdef BSD43_DRIVER static void eiTxStartup
    (
    int unit
    )
```

DESCRIPTION

Looks for any action on the queue, and begins output if there is anything there. This routine is called from several possible threads. Each will be described below.

The first, and most common thread, is when a user task requests the transmission of data. Under BSD 4.3, this will cause ***eiOutput*( )** to be called, which calls ***ether_output*( )**, which will usually call this routine. This routine will not be called if ***ether_output*( )** finds that our interface output queue is full. In this case, the outgoing data will be thrown out. BSD 4.4 uses a slightly different model in which the generic ***ether_output*( )** routine is called directly, followed by a call to this routine.

The second, and most obscure thread, is when the reception of certain packets causes an immediate (attempted) response. For example, ICMP echo packets (ping), and ICMP "no listener on that port" notifications. All functions in this driver that handle the reception side are executed in the context of ***netTask*( )**. Always. So, in the case being discussed, ***netTask*( )** will receive these certain packets, cause IP to be stimulated, and cause the generation of a response to be sent. We then find ourselves following the thread explained in the second example, with the important distinction that the context is that of ***netTask*( )**.

The third thread occurs when this routine runs out of TFDs and returns. If this occurs when our output queue is not empty, this routine would typically not get called again until new output was requested. Even worse, if the output queue was also full, this routine would never get called again and we would have a lock state. It DOES happen. To guard against this, the transmit clean-up handler detects the out-of-TFDs state and calls this function. The clean-up handler also runs from netTask.

Note that this function is ALWAYS called between an ***splnet*( )** and an ***splx*( )**. This is true because ***netTask*( )**, and ***ether_output*( )** take care of this when calling this function. Therefore, no calls to these spl functions are needed anywhere in this output thread.

SEE ALSO **if_eihk**

---

eiTxStartup( )

NAME ***eiTxStartup*( )** – start output on the chip

SYNOPSIS

```
void eiTxStartup
    (
    DRV_CTRL * pDrvCtrl
    )
```

DESCRIPTION Looks for any action on the queue, and begins output if there is anything there. This routine is called from several possible threads. Each will be described below.

The first, and most common thread, is when a user task requests the transmission of data. This will cause ***eiOutput*( )** to be called, which will cause ***ether_output*( )** to be called, which will cause this routine to be called (usually). This routine will not be called if ***ether_output*( )** finds that our interface output queue is full. In this case, the outgoing data will be thrown out.

The second, and most obscure thread, is when the reception of certain packets causes an immediate (attempted) response. For example, ICMP echo packets (ping), and ICMP "no listener on that port" notifications. All functions in this driver that handle the reception side are executed in the context of ***netTask*( )**. Always. So, in the case being discussed, ***netTask*( )** will receive these certain packets, cause IP to be stimulated, and cause the generation of a response to be sent. We then find ourselves following the thread explained in the second example, with the important distinction that the context is that of ***netTask*( )**.

The third thread occurs when this routine runs out of TFDs and returns. If this occurs when our output queue is not empty, this routine would typically not get called again until new output was requested. Even worse, if the output queue was also full, this routine would never get called again and we would have a lock state. It DOES happen. To guard against this, the transmit clean-up handler detects the out-of-TFDs state and calls this function. The clean-up handler also runs from netTask.

Note that this function is ALWAYS called between an ***splnet*( )** and an ***splx*( )**. This is true because ***netTask*( )**, and ***ether_output*( )** take care of this when calling this function. Therefore, no calls to these spl functions are needed anywhere in this output thread.

SEE ALSO **if_ei**

el3c90xEndLoad( )

NAME ***el3c90xEndLoad*( )** – initialize the driver and device

SYNOPSIS

```
END_OBJ * el3c90xEndLoad
    (
    char * initString /* String to be parsed by the driver. */
    )
```

DESCRIPTION This routine initializes the driver and the device to the operational state. All of the device-specific parameters are passed in *initString*, which expects a string of the following format:

unit:*devMemAddr*:*devIoAddr*:*pciMemBase:<vecnum*:*intLvl*:*memAdrs*:*memSize*:*memWidth*:*flags*:*buffMultiplier*

This routine can be called in two modes. If it is called with an empty but allocated string, it places the name of this device (that is, "elPci") into the *initString* and returns 0.

If the string is allocated and not empty, the routine attempts to load the driver using the values specified in the string.

RETURNS An END object pointer, or NULL on error, or 0 and the name of the device if the *initString* was NULL.

SEE ALSO **el3c90xEnd**

el3c90xInitParse( )

NAME

el3c90xInitParse( ) – parse the initialization string

SYNOPSIS

```
STATUS el3c90xInitParse
    (
    EL3C90X_DEVICE * pDrvCtrl,  /* pointer to the control structure */
    char *           initString /* initialization string */
    )
```

DESCRIPTION

Parse the input string. This routine is called from *el3c90xEndLoad*( ) which intializes some values in the driver control structure with the values passed in the intialization string.

The initialization string format is:
unit:*devMemAddr*:*devIoAddr*:*pciMemBase:<vecNum*:*intLvl*:*memAdrs*:*memSize*:*memWidth*:*flags*:
buffMultiplier

unit
: Device unit number, a small integer.

devMemAddr
: Device register base memory address

devIoAddr
: Device register base IO address

pciMemBase
: Base address of PCI memory space

vecNum
: Interrupt vector number.

intLvl
: Interrupt level.

memAdrs
: Memory pool address or NONE.

memSize
: Memory pool size or zero.

memWidth
: Memory system size, 1, 2, or 4 bytes (optional).

flags
: Device specific flags, for future use.

buffMultiplier
: Buffer Multiplier or NONE. If NONE is specified, it defaults to 2

RETURNS OK, or ERROR if any arguments are invalid.

SEE ALSO **el3c90xEnd**

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**

elcPut( )

NAME *elcPut*( ) – copy a packet to the interface.

SYNOPSIS

```
#ifdef BSD43_DRIVER LOCAL void elcPut
    (
    int unit
    )
```

DESCRIPTION Copy from mbuf chain to transmitter buffer in shared memory.

SEE ALSO **if_elc**

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 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_elc**

elt3c509Load( )

NAME *elt3c509Load*( ) – initialize the driver and device

SYNOPSIS

```
END_OBJ * elt3c509Load
    (
    char * initString /* String to be parsed by the driver. */
    )
```

DESCRIPTION This routine initializes the driver and the device to the operational state. All of the device-specific parameters are passed in *initString*, which expects a string of the following format:

unit:*port*:*intVector*:*intLevel*:*attachementType*:*noRxFrames*

This routine can be called in two modes. If it is called with an empty but allocated string, it places the name of this device (that is, "elt") into the *initString* and returns 0.

If the string is allocated and not empty, the routine attempts to load the driver using the values specified in the string.

RETURNS
An END object pointer, or NULL on error, or 0 and the name of the device if the *initString* was NULL.

SEE ALSO
elt3c509End

elt3c509Parse( )

NAME
elt3c509Parse( ) – parse the init string

SYNOPSIS

```
STATUS elt3c509Parse
    (
    ELT3C509_DEVICE * pDrvCtrl,  /* device pointer */
    char *            initString /* initialization info string */
    )
```

DESCRIPTION
Parse the input string. Fill in values in the driver control structure.

The initialization string format is:
unit:*port*:*intVector*:*intLevel*:*attachementType*:*noRxFrames*

unit
Device unit number, a small integer.

port
base I/O address

intVector
Interrupt vector number (used with sysIntConnect)

intLevel
Interrupt level

attachmentType
type of Ethernet connector

nRxFrames
no. of Rx Frames in integer format

RETURNS
OK or ERROR for invalid arguments.

SEE ALSO
elt3c509End

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, filling in a network interface record and adding the record to the system list. It also initializes the driver and 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**

eltTxOutputStart( )

NAME

eltTxOutputStart( ) – start output on the board

SYNOPSIS

```
#ifdef BSD43_DRIVER static void eltTxOutputStart
    (
    int unit
    )
```

DESCRIPTION

This routine is called from ***ether_output*( )** when a new packet is enqueued in the interface mbuf queue.

Note that this function is ALWAYS called between an ***splnet*( )** and an ***splx*( )**. This is true because ***netTask*( )**, and ***ether_output*( )** take care of this when calling this function. Therefore, no calls to these spl functions are needed anywhere in this output thread.

SEE ALSO

if_elt

endEtherAddressForm( )

NAME

endEtherAddressForm( ) – form an Ethernet address into a packet

SYNOPSIS

```
M_BLK_ID endEtherAddressForm
    (
    M_BLK_ID pMblk,    /* pointer to packet mBlk */
    M_BLK_ID pSrcAddr, /* pointer to source address */
    M_BLK_ID pDstAddr  /* pointer to destination address */
    )
```

DESCRIPTION

This routine accepts the source and destination addressing information through *pSrcAddr* and *pDstAddr* and returns an **M_BLK_ID** that points to the assembled link-level header. To do this, this routine prefixes the link-level header into the cluster associated with *pMblk* if there is enough space available in the cluster. It then returns a pointer to the pointer referenced in *pMblk*. However, if there is not enough space in the cluster associated with *pMblk*, this call reserves a new **mBlk-clBlk**-cluster construct for the header information. It then prepends the new **mBlk** to the **mBlk** passed in *pMblk*. As the function value, this routine then returns a pointer to the new **mBlk**, which is the head of a chain of **mBlk** structures. The second element in the chain is the **mBlk** referenced in *pMblk*.

RETURNS

M_BLK_ID or NULL.

SEE ALSO

endLib

endEtherPacketAddrGet( )

NAME

endEtherPacketAddrGet( ) – locate the addresses in a packet

SYNOPSIS

```
STATUS endEtherPacketAddrGet
    (
    M_BLK_ID pMblk, /* pointer to packet */
    M_BLK_ID pSrc,  /* pointer to local source address */
    M_BLK_ID pDst,  /* pointer to local destination address */
    M_BLK_ID pESrc, /* pointer to remote source address (if any) */
    M_BLK_ID pEDst  /* pointer to remote destination address (if any) */
    )
```

DESCRIPTION

This routine takes a **M_BLK_ID**, locates the address information, and adjusts the **M_BLK_ID** structures referenced in *pSrc*, *pDst*, *pESrc*, and *pEDst* so that their pData members point to the addressing information in the packet. The addressing information is not copied. All **mBlk** structures share the same cluster.

RETURNS

OK or ERROR.

SEE ALSO

endLib

endEtherPacketDataGet( )

NAME

endEtherPacketDataGet( ) – return the beginning of the packet data

SYNOPSIS

```
STATUS endEtherPacketDataGet
    (
    M_BLK_ID      pMblk,
    LL_HDR_INFO * pLinkHdrInfo
    )
```

DESCRIPTION

This routine fills the given *pLinkHdrInfo* with the appropriate offsets.

RETURNS

OK or ERROR.

SEE ALSO

endLib

endFindByName( )

NAME *endFindByName*( ) – find a device using its string name

SYNOPSIS

```
END_OBJ* endFindByName
    (
    char* pName, /* device name to search for */
    int   unit
    )
```

DESCRIPTION This routine takes a string name and a unit number and finds the END device that has that name/unit combination.

RETURNS A pointer to an **END_OBJ** or NULL (if the device is not found).

SEE ALSO **muxLib**

endObjFlagSet( )

NAME *endObjFlagSet*( ) – set the **flags** member of an **END_OBJ** structure

SYNOPSIS

```
STATUS endObjFlagSet
    (
    END_OBJ * pEnd,
    UINT      flags
    )
```

DESCRIPTION As input, this routine expects a pointer to an **END_OBJ** structure (the *pEnd* parameter) and a flags value (the *flags* parameter). This routine sets the **flags** member of the **END_OBJ** structure to the value of the *flags* parameter.

Because this routine assumes that the driver interface is now up, this routine also sets the **attached** member of the referenced **END_OBJ** structure to TRUE.

RETURNS OK

SEE ALSO **endLib**

endObjInit( )

NAME *endObjInit*( ) – initialize an **END_OBJ** structure

SYNOPSIS

```
STATUS endObjInit
    (
    END_OBJ *   pEndObj,     /* object to be initialized */
    DEV_OBJ*    pDevice,     /* ptr to device struct */
    char *      pBaseName,   /* device base name, for example, "ln" */
    int         unit,        /* unit number */
    NET_FUNCS * pFuncTable,  /* END device functions */
    char*       pDescription
    )
```

DESCRIPTION This routine initializes an **END_OBJ** structure and fills it with data from the argument list. It also creates and initializes semaphores and protocol list.

RETURNS OK or ERROR.

SEE ALSO **endLib**

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**

enePut( )

NAME *enePut*( ) – copy a packet to the interface.

SYNOPSIS

```
#ifdef BSD43_DRIVER static void enePut
    (
    int unit
    )
```

DESCRIPTION Copy from mbuf chain to transmitter buffer in shared memory.

SEE ALSO **if_ene**

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**

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**

envoy_call_timer( )

NAME *envoy_call_timer*( ) – execute the specified function when the timer expires

SYNOPSIS

```
void envoy_call_timer
    (
    bits32_t when,
    void (*  what)(void)
    )
```

DESCRIPTION This routine executes the *what* function after *when* ticks have elapsed. This function is used internally to respond when the interval between the test and set of a "test and set" exceeds the timeout specified by *when*.

RETURNS N/A

SEE ALSO **saIoLib**

envoy_now( )

NAME *envoy_now*( ) – return the number of clock ticks elapsed since the timer was set

SYNOPSIS

```
bits32_t envoy_now (void)
```

DESCRIPTION Call this function to find out the number of clock ticks elapsed since the timer was set.

RETURNS Elapsed time, in ticks.

SEE ALSO **saIoLib**

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 to new private env */
                  /* taskId = copy the specified 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 *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*( )

esmcattach( )

NAME

esmcattach( ) – publish the **esmc** network interface and initialize the driver.

SYNOPSIS

```
STATUS esmcattach
    (
    int unit,     /* unit number */
    int ioAddr,   /* address of esmc's shared memory */
    int intVec,   /* interrupt vector to connect to */
    int intLevel, /* interrupt level */
    int config,   /* 0: Autodetect 1: AUI 2: BNC 3: RJ45 */
    int mode      /* 0: rx in interrupt 1: rx in task(netTask) */
    )
```

DESCRIPTION

This routine attaches an **esmc** 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_esmc, **ifLib**, **netShow**

esmcPut( )

NAME

esmcPut( ) – copy a packet to the interface.

SYNOPSIS

```
#ifdef BSD43_DRIVER LOCAL void esmcPut
    (
    int unit
    )
```

DESCRIPTION

Copy from mbuf chain to transmitter buffer in shared memory.

RETURNS

N/A

SEE ALSO

if_esmc

esmcShow( )

NAME

esmcShow( ) – display statistics for the **esmc** network interface

SYNOPSIS

```
void esmcShow
    (
    int  unit, /* interface unit */
    BOOL zap   /* zero totals */
    )
```

DESCRIPTION

This routine displays statistics about the **esmc** 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_esmc

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 */
    char*   pName,     /* name of device if MUX/END is being used */
    int     unit       /* unit of device if MUX/END is being used */
    )
```

DESCRIPTION

This routine adds a hook routine that will be called for every Ethernet packet that is 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 (for example, 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
    (
    FUNCPTR inputHook,
    char *  pName,
    int     unit
    )
```

DESCRIPTION This routine deletes a network interface input hook.

RETURNS N/A

SEE ALSO **etherLib**

etherMultiAdd( )

NAME *etherMultiAdd*( ) – add multicast address to a multicast address list

SYNOPSIS

```
int etherMultiAdd
    (
    LIST * pList,   /* pointer to list of multicast addresses */
    char*  pAddress /* address you want to add to list */
    )
```

DESCRIPTION This routine adds an Ethernet multicast address list for a given END. The address is a six-byte value pointed to by *pAddress*.

RETURNS OK or ENETRESET.

SEE ALSO **etherMultiLib**

etherMultiDel( )

NAME *etherMultiDel*( ) – delete an Ethernet multicast address record

SYNOPSIS

```
int etherMultiDel
    (
    LIST * pList,   /* pointer to list of multicast addresses */
    char*  pAddress /* address you want to add to list */
    )
```

DESCRIPTION This routine deletes an Ethernet multicast address from the list. The address is a six-byte value pointed to by *pAddress*.

RETURNS OK or ENETRESET.

SEE ALSO **etherMultiLib**

2

etherMultiGet( )

NAME *etherMultiGet*( ) – retrieve a table of multicast addresses from a driver

SYNOPSIS

```
int etherMultiGet
    (
    LIST*        pList, /* pointer to list of multicast addresses */
    MULTI_TABLE* pTable /* table into which to copy addresses */
    )
```

DESCRIPTION This routine runs down the multicast address list stored in a driver and places all the entries it finds into the multicast table structure passed to it.

RETURNS OK or ERROR.

SEE ALSO **etherMultiLib**

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, if the hook could be added, ERROR otherwise.

SEE ALSO **etherLib**

etherOutputHookDelete( )

NAME *etherOutputHookDelete*( ) – delete a network interface output hook routine

SYNOPSIS

```
void etherOutputHookDelete
    (
    FUNCPTR outputHook
    )
```

DESCRIPTION This routine deletes a network interface output hook, which must be supplied as the only argument.

RETURNS N/A

SEE ALSO **etherLib**

etherTypeGet( )

NAME

etherTypeGet( ) – get the type from an ethernet packet

SYNOPSIS

```
USHORT etherTypeGet
    (
    char * pPacket /* pointer to the beginning of the packet */
    )
```

DESCRIPTION

This routine returns a short that is the ethertype (defined in RFC 1700) from either an 802.3 addressed packet or an RFC 894 packet. Most packets are encoded as described in RFC 894 but we should also be able to understand 802.3 addressing.

RETURNS

A USHORT value that is the ethertype, or 0 on error.

SEE ALSO

etherLib, *RFC 894, TCP/IP Illustrated,* Volume 1, by Richard Stevens.

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

excConnect( )

NAME

excConnect( ) – connect a C routine to an exception vector (PowerPC)

SYNOPSIS

```
STATUS excConnect
    (
    VOIDFUNCPTR * vector, /* exception vector to attach to */
    VOIDFUNCPTR   routine /* routine to be called */
    )
```

DESCRIPTION

This routine connects a specified C routine to a specified exception vector. An exception stub is created and in placed at *vector* in the exception table. The address of *routine* is stored in the exception stub code. When an exception occurs, the processor jumps to the exception stub code, saves the registers, and calls the C routines.

The routine can be any normal C code, except that it must not invoke certain operating system functions that may block or perform I/O operations.

The registers are saved to an Exception Stack Frame (ESF) placed on the stack of the task that has produced the exception. The structure of the ESF used to save the registers is defined in **h/arch/ppc/esfPpc.h**.

The only argument passed by the exception stub to the C routine is a pointer to the ESF containing the registers values. The prototype of this C routine is described below:

```
    void excHandler (ESFPPC *);
```

When the C routine returns, the exception stub restores the registers saved in the ESF and continues execution of the current task.

RETURNS OK, always.

SEE ALSO **excArchLib**, ***excIntConnect*( )**, ***excVecSet*( )**

---

excCrtConnect( )

NAME *excCrtConnect*( ) – connect a C routine to a critical exception vector (PowerPC 403)

SYNOPSIS

```
STATUS excCrtConnect
    (
    VOIDFUNCPTR * vector, /* exception vector to attach to */
    VOIDFUNCPTR   routine /* routine to be called */
    )
```

DESCRIPTION This routine connects a specified C routine to a specified critical exception vector. An exception stub is created and in placed at *vector* in the exception table. The address of *routine* is stored in the exception stub code. When an exception occurs, the processor jumps to the exception stub code, saves the registers, and call the C routines.

The routine can be any normal C code, except that it must not invoke certain operating system functions that may block or perform I/O operations.

The registers are saved to an Exception Stack Frame (ESF) which is placed on the stack of the task that has produced the exception. The ESF structure is defined in **h/arch/ppc/esfPpc.h**.

The only argument passed by the exception stub to the C routine is a pointer to the ESF containing the register values. The prototype of this C routine is as follows:

```
void excHandler (ESFPPC *);
```

When the C routine returns, the exception stub restores the registers saved in the ESF and continues execution of the current task.

RETURNS OK, always.

SEE ALSO **excArchLib**, ***excIntConnect*( )**, excIntCrtConnect, ***excVecSet*( )**

2

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, ARM)

SYNOPSIS

```
STATUS excIntConnect
    (
    VOIDFUNCPTR * vector, /* exception vector to attach to */
    VOIDFUNCPTR   routine /* routine to be called */
    )
```

DESCRIPTION This routine connects a specified C routine to a specified asynchronous exception vector.

When the C routine is invoked, interrupts are still locked. It is the responsibility of the C routine to re-enable the interrupt.

The routine can be any normal C code, except that it must not invoke certain operating system functions that may block or perform I/O operations.

NOTE On PowerPC, the vector is typically the external interrupt vector 0x500 and the decrementer vector 0x900. An interrupt stub is created and placed at *vector* in the exception table. The address of *routine* is stored in the interrupt stub code. When the asynchronous exception occurs the processor jumps to the interrupt stub code, saves only the requested registers, and calls the C routines.

Before saving the requested registers, the interrupt stub switches from the current task stack to the interrupt stack. For nested interrupts, no stack-switching is performed, because the interrupt is already set.

NOTE On the ARM, the address of *routine* is stored in a function pointer to be called by the stub installed on the IRQ exception vector following an asynchronous exception. This routine is responsible for determining the interrupt source and despatching the correct handler for that source.

Before calling the routine, the interrupt stub switches to SVC mode, changes to a separate interrupt stack and saves necessary registers. In the case of a nested interrupt, no SVC stack switch occurs.

RETURNS OK, always.

SEE ALSO **excArchLib**, *excConnect***( )**, *excVecSet***( )**

---

excIntCrtConnect( )

NAME *excIntCrtConnect***( )** – connect a C routine to a critical interrupt vector (PowerPC 403)

SYNOPSIS

```
STATUS excIntCrtConnect
    (
    VOIDFUNCPTR * vector, /* exception vector to attach to */
    VOIDFUNCPTR   routine /* routine to be called */
    )
```

DESCRIPTION This routine connects a specified C routine to a specified asynchronous critical exception vector such as the critical external interrupt vector (0x100), or the watchdog timer vector (0x1020). An interrupt stub is created and placed at *vector* in the exception table. The address of *routine* is stored in the interrupt stub code. When the asynchronous exception occurs, the processor jumps to the interrupt stub code, saves only the requested registers, and calls the C routines.

When the C routine is invoked, interrupts are still locked. It is the C routine's responsibility to re-enable interrupts.

The routine can be any normal C routine, except that it must not invoke certain operating system functions that may block or perform I/O operations.

Before the requested registers are saved, the interrupt stub switches from the current task stack to the interrupt stack. In the case of nested interrupts, no stack switching is performed, because the interrupt stack is already set.

RETURNS OK, always.

SEE ALSO **excArchLib**, *excConnect***( )**, excCrtConnect, *excVecSet***( )**

excTask( )

NAME
excTask( ) – handle task-level exceptions

SYNOPSIS
```
void excTask ()
```

DESCRIPTION
This routine is spawned as a task by ***excInit*( )** to perform functions that cannot be performed at interrupt or trap level. It has a priority of 0. Do not suspend, delete, or change the priority of this task.

RETURNS
N/A

SEE ALSO
excLib, *excInit*( )

excVecGet( )

NAME
excVecGet( ) – get a CPU exception vector (PowerPC, ARM)

SYNOPSIS
```
FUNCPTR excVecGet
    (
    FUNCPTR * vector /* vector offset */
    )
```

DESCRIPTION
This routine returns the address of the C routine currently connected to *vector*.

RETURNS
The address of the C routine.

SEE ALSO
excArchLib, *excVecSet*( )

excVecInit( )

NAME
excVecInit( ) – initialize the exception/interrupt vectors

SYNOPSIS
```
STATUS excVecInit (void)
```

DESCRIPTION

This routine sets all exception vectors to point to the appropriate default exception handlers. These handlers will safely trap and report exceptions caused by program errors or unexpected hardware interrupts.

MC680x0:
All vectors from vector 2 (address 0x0008) to 255 (address 0x03fc) are initialized. Vectors 0 and 1 contain the reset stack pointer and program counter.

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.

ARM
All exception vectors are initialized to default handlers except 0x14 (Address) which is now reserved on the ARM and 0x1C (FIQ), which is not used by VxWorks.

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, ARM)

SYNOPSIS

```
void excVecSet
    (
    FUNCPTR * vector,  /* vector offset */
    FUNCPTR   function /* address to place in vector */
    )
```

DESCRIPTION

This routine specifies the C routine that will be called when the exception corresponding to *vector* occurs. This routine does not create the exception stub; it simply replaces the C routine to be called in the exception stub.

NOTE ARM

On the ARM, there is no ***excConnect( )*** routine, unlike the PowerPC. The C routine is attached to a default stub using ***excVecSet( )***.

RETURNS

N/A

SEE ALSO

excArchLib, ***excVecGet( )***, ***excConnect( )***, ***excIntConnect( )***

---

exit( )

NAME

exit( ) – exit a task (ANSI)

SYNOPSIS

```
void exit
    (
    int code /* code stored in TCB for delete hooks */
    )
```

DESCRIPTION

This routine is called by a task to cease to exist as a task. It is called implicitly when the "main" routine of a spawned task is exited. The *code* parameter will be stored in the **WIND_TCB** for possible use by the delete hooks, or post-mortem debugging.

ERRNO

N/A

SEE ALSO

taskLib, ***taskDelete( )***, *American National Standard for Information Systems – Programming Language – C, ANSI X3.159-1989: Input/Output (**stdlib.h**), VxWorks Programmer's Guide: Basic OS*

---

exp( )

NAME

exp( ) – compute an exponential value (ANSI)

SYNOPSIS

```
double exp
    (
    double x /* exponent */
    )
```

DESCRIPTION This routine returns the exponential value of *x* in double precision (IEEE double, 53 bits).

A range error occurs if *x* is too large.

INCLUDE FILES **math.h**

RETURNS The double-precision exponential value of *x*.

Special cases:
If *x* is +INF or NaN, ***exp*( )** returns *x*.
If *x* is -INF, it returns 0.

SEE ALSO **ansiMath**, **mathALib**

---

expf( )

NAME ***expf*( )** – compute an exponential value (ANSI)

SYNOPSIS

```
float expf
    (
    float x /* exponent */
    )
```

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

INCLUDE FILES **math.h**

RETURNS The single-precision exponential value of *x*.

SEE ALSO **mathALib**

---

fabs( )

NAME ***fabs*( )** – compute an absolute value (ANSI)

SYNOPSIS

```
double fabs
    (
    double v /* number to return the absolute value of */
    )
```

DESCRIPTION This routine returns the absolute value of *v* in double precision.

INCLUDE FILES **math.h**

RETURNS The double-precision absolute value of *v*.

ERRNO **EDOM**, **ERANGE**

SEE ALSO **ansiMath**, **mathALib**

---

fabsf( )

NAME *fabsf*( ) – compute an absolute value (ANSI)

SYNOPSIS

```
float fabsf
    (
    float v /* number to return the absolute value of */
    )
```

DESCRIPTION This routine returns the absolute value of *v* in single precision.

INCLUDE FILES **math.h**

RETURNS The single-precision absolute value of *v*.

SEE ALSO **mathALib**

---

fclose( )

NAME *fclose*( ) – close a stream (ANSI)

SYNOPSIS

```
int fclose
    (
    FILE * fp /* stream to close */
    )
```

DESCRIPTION This routine flushes a specified stream and closes the associated file. Any unwritten buffered data is delivered to the host environment to be written to the file; any unread buffered data is discarded. The stream is disassociated from the file. If the associated buffer was allocated automatically, it is deallocated.

INCLUDE FILES **stdio.h**

RETURNS Zero if the stream is closed successfully, or EOF if errors occur.

ERRNO **EBADF**

SEE ALSO **ansiStdio**, *fflush***( )**

---

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 */
```

T he *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.

ERRNO **EINVAL**

SEE ALSO **ansiStdio**, ***fopen*( )**, ***freopen*( )**, .br *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 */
    )
```

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*( )

fei82557EndLoad( )

NAME

fei82557EndLoad( ) – initialize the driver and device

SYNOPSIS

```
END_OBJ* fei82557EndLoad
    (
    char * initString /* parameter string */
    )
```

DESCRIPTION

This routine initializes both, driver and device to an operational state using device specific parameters specified by *initString*.

The parameter string, *initString*, is an ordered list of parameters each separated by a colon. The format of *initString* is, "*unit*:*memBase*:*memSize*:*nCFDs*:*nRFDs*:*flags*"

The 82557 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.

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 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. The *memSize* parameter is used to check that this region is large enough with respect to the provided values of both transmit/receive frames.

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 write coherent, this routine will abort and return ERROR.

RETURNS an END object pointer, or NULL on error.

SEE ALSO **fei82557End**, **ifLib**, *Intel 82557 User's Manual*

---

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.

ERRNO **EBADF**

SEE ALSO **ansiStdio**, *fclose*( )

fgetc( )

NAME

fgetc( ) – return the next character from a stream (ANSI)

SYNOPSIS

```
int fgetc
    (
    FILE * fp /* stream to read from */
    )
```

DESCRIPTION

This routine returns the next character (converted to an **int**) from the specified stream, and advances the file position indicator for the stream.

If the stream is at end-of-file, the end-of-file indicator for the stream is set; if a read error occurs, the error indicator is set.

INCLUDE FILES

stdio.h

RETURNS

The next character from the stream, or EOF if the stream is at end-of-file or a read error occurs.

SEE ALSO

ansiStdio, *fgets*( ), *getc*( )

fgetpos( )

NAME

fgetpos( ) – store the current value of the file position indicator for a stream (ANSI)

SYNOPSIS

```
int fgetpos
    (
    FILE *   fp, /* stream */
    fpos_t * pos /* where to store position */
    )
```

DESCRIPTION

This routine stores the current value of the file position indicator for a specified stream *fp* in the object pointed to by *pos*. The value stored contains unspecified information usable by *fsetpos*( ) for repositioning the stream to its position at the time *fgetpos*( ) was called.

INCLUDE FILES

stdio.h

RETURNS

Zero, or non-zero if unsuccessful, with **errno** set to indicate the error.

SEE ALSO

ansiStdio, *fsetpos*( )

fgets( )

NAME *fgets*( ) – read a specified number of characters from a stream (ANSI)

SYNOPSIS

```
char * fgets
    (
    char * buf, /* where to store characters */
    size_t n,   /* no. of bytes to read + 1 */
    FILE * fp   /* stream to read from */
    )
```

DESCRIPTION This routine stores in the array *buf* up to *n*-1 characters from a specified stream. No additional characters are read after a new-line or end-of-line. A null character is written immediately after the last character read into the array.

If end-of-file is encountered and no characters have been read, the contents of the array remain unchanged. If a read error occurs, the array contents are indeterminate.

INCLUDE FILES **stdio.h**

RETURNS A pointer to *buf*, or a null pointer if an error occurs or end-of-file is encountered and no characters have been read.

SEE ALSO **ansiStdio**, *fread*( ), *fgetc*( )

fileno( )

NAME *fileno*( ) – return the file descriptor for a stream (POSIX)

SYNOPSIS

```
int fileno
    (
    FILE * fp /* stream */
    )
```

DESCRIPTION This routine returns the file descriptor associated with a specified stream.

INCLUDE FILES **stdio.h**

RETURNS The file descriptor, or -1 if an error occurs, with **errno** set to indicate the error.

SEE ALSO **ansiStdio**, *Information Technology – POSIX – Part 1: System API [C Lang.], 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

2

fioLibInit( )

NAME *fioLibInit*( ) – initialize the formatted I/O support library

SYNOPSIS

```
void fioLibInit (void)
```

DESCRIPTION This routine initializes the formatted I/O support library. It should be called once in ***usrRoot*( )** when formatted I/O functions such as ***printf*( )** and ***scanf*( )** are used.

RETURNS N/A

SEE ALSO **fioLib**

fioRdString( )

NAME *fioRdString*( ) – read a string from a file

SYNOPSIS

```
int fioRdString
    (
    int  fd,       /* fd of device to read */
    char string[], /* buffer to receive input */
    int  maxbytes  /* max no. of chars to read */
    )
```

DESCRIPTION This routine puts a line of input into *string*. The specified input file descriptor is read until *maxbytes*, an EOF, an EOS, or a newline character is reached. A newline character or EOF is replaced with EOS, unless *maxbytes* characters have been read.

RETURNS The length of the string read, including the terminating EOS; or EOF if a read error occurred or end-of-file occurred without reading any other character.

SEE ALSO **fioLib**

fioRead( )

NAME *fioRead*( ) – read a buffer

SYNOPSIS

```
int fioRead
    (
    int    fd,       /* file descriptor of file to read */
    char * buffer,   /* buffer to receive input */
    int    maxbytes  /* maximum number of bytes to read */
    )
```

DESCRIPTION This routine repeatedly calls the routine ***read*( )** until *maxbytes* have been read into *buffer*. If EOF is reached, the number of bytes read will be less than *maxbytes*.

RETURNS The number of bytes read, or ERROR if there is an error during the read operation.

SEE ALSO **fioLib**, *read*( )

floatInit( )

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

SYNOPSIS

```
void floatInit (void)
```

DESCRIPTION This routine must be called if floating-point format specifications are to be supported by the ***printf*( )**/***scanf*( )** family of routines. If the configuration macro **INCLUDE_FLOATING_POINT** is defined, it is called by the root task, ***usrRoot*( )**, in **usrConfig.c**.

RETURNS N/A

SEE ALSO **floatLib**

floor( )

NAME *floor*( ) – compute the largest integer less than or equal to a specified value (ANSI)

SYNOPSIS

```
double floor
    (
    double v /* value to find the floor of */
    )
```

DESCRIPTION This routine returns the largest integer less than or equal to *v*, in double precision.

INCLUDE FILES **math.h**

RETURNS The largest integral value less than or equal to *v*, in double precision.

SEE ALSO **ansiMath**, **mathALib**

floorf( )

NAME *floorf*( ) – compute the largest integer less than or equal to a specified value (ANSI)

SYNOPSIS

```
float floorf
    (
    float v /* value to find the floor of */
    )
```

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

INCLUDE FILES **math.h**

RETURNS The largest integral value less than or equal to *v*, in single precision.

SEE ALSO **mathALib**

fmod( )

NAME
fmod( ) – compute the remainder of x/y (ANSI)

SYNOPSIS
```
double fmod
    (
    double x, /* numerator */
    double y  /* denominator */
    )
```

DESCRIPTION
This routine returns the remainder of *x*/*y* with the sign of *x*, in double precision.

INCLUDE FILES
math.h

RETURNS
The value *x* - *i* * *y*, for some integer *i*. If *y* is non-zero, the result has the same sign as *x* and magnitude less than the magnitude of *y*. If *y* is zero, ***fmod*( )** returns zero.

ERRNO
EDOM

SEE ALSO
ansiMath, **mathALib**

fmodf( )

NAME
fmodf( ) – compute the remainder of x/y (ANSI)

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

DESCRIPTION
This routine returns the remainder of *x*/*y* with the sign of *x*, in single precision.

INCLUDE FILES
math.h

RETURNS
The single-precision modulus of *x*/*y*.

SEE ALSO
mathALib

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*( )**

2

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 the configuration macro **INCLUDE_HW_FP** is defined.

RETURNS N/A

SEE ALSO **fppLib**

fppProbe( )

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

SYNOPSIS

```
STATUS fppProbe (void)
```

DESCRIPTION This routine determines whether there is a floating-point coprocessor in the system.

The implementation of this routine is architecture-dependent:

MC680x0, 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.

ARM

This routine currently returns ERROR to indicate no floating-point coprocessor support.

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
– registers **st0** – **st7**

ARM
– currently, on this architecture, this routine does nothing.

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)

ARM
- – currently, on this architecture, this routine does nothing.

RETURNS N/A

SEE ALSO **fppArchLib**, *fppRestore*( )

fppShowInit( )

NAME *fppShowInit*( ) – initialize the floating-point show facility

SYNOPSIS

```
void fppShowInit (void)
```

DESCRIPTION This routine links the floating-point show facility into the VxWorks system. It is called automatically when the floating-point show facility is configured into VxWorks using either of the following methods:

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

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

RETURNS N/A

SEE ALSO **fppShow**

fppTaskRegsGet( )

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

SYNOPSIS

```
STATUS fppTaskRegsGet
    (
    int         task,     /* task to get info about */
    FPREG_SET * pFpRegSet /* ptr to floating-point register set */
    )
```

DESCRIPTION This routine copies a task's floating-point registers and/or status registers to the locations whose pointers are passed as parameters. The floating-point registers are copied into an array containing all the registers.

NOTE This routine only works well if *task* is not the calling task. If a task tries to discover its own registers, the values will be stale (that is, left over from the last task switch).

RETURNS OK, or ERROR if there is no floating-point support or there is an invalid state.

SEE ALSO **fppArchLib**, *fppTaskRegsSet*( )

fppTaskRegsSet( )

NAME

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

SYNOPSIS

```
STATUS fppTaskRegsSet
    (
    int         task,     /* task to set registers for */
    FPREG_SET * pFpRegSet /* ptr to floating-point register set */
    )
```

DESCRIPTION

This routine loads the specified values into the TCB of a specified task. The register values are copied from the array at *pFpRegSet*.

RETURNS

OK, or ERROR if there is no floating-point support or there is an invalid state.

SEE ALSO

fppArchLib, *fppTaskRegsGet*( )

fppTaskRegsShow( )

NAME

fppTaskRegsShow( ) – print the contents of a task's floating-point registers

SYNOPSIS

```
void fppTaskRegsShow
    (
    int task /* task to display floating point registers for */
    )
```

DESCRIPTION

This routine prints to standard output the contents of a task's floating-point registers.

RETURNS

N/A

SEE ALSO

fppShow

2

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 */
    )
```

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*( )**

2

fputc( )

NAME *fputc*( ) – write a character to a stream (ANSI)

SYNOPSIS

```
int fputc
    (
    int    c, /* character to write */
    FILE * fp /* stream to write to */
    )
```

DESCRIPTION This routine writes a character *c* to a specified stream, at the position indicated by the stream's file position indicator (if defined), and advances the indicator appropriately.

If the file cannot support positioning requests, or if the stream was opened in append mode, the character is appended to the output stream.

INCLUDE FILES **stdio.h**

RETURNS The character written, or EOF if a write error occurs, with the error indicator set for the stream.

SEE ALSO **ansiStdio**, *fputs*( ), *putc*( )

fputs( )

NAME *fputs*( ) – write a string to a stream (ANSI)

SYNOPSIS

```
int fputs
    (
    const char * s, /* string */
    FILE *       fp /* stream to write to */
    )
```

DESCRIPTION This routine writes the string *s*, minus the terminating NULL character, to a specified stream.

INCLUDE FILES **stdio.h**

RETURNS A non-negative value, or EOF if a write error occurs.

SEE ALSO **ansiStdio**, *fputc*( )

fread( )

NAME

fread( ) – read data into an array (ANSI)

SYNOPSIS

```
int fread
    (
    void * buf,    /* where to copy data */
    size_t size,   /* element size */
    size_t count,  /* no. of elements */
    FILE * fp      /* stream to read from */
    )
```

DESCRIPTION

This routine reads, into the array *buf*, up to *count* elements of size *size*, from a specified stream *fp*. The file position indicator for the stream (if defined) is advanced by the number of characters successfully read. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate. If a partial element is read, its value is indeterminate.

INCLUDE FILES

stdio.h

RETURNS

The number of elements successfully read, which may be less than *count*if a read error or end-of-file is encountered; or zero if *size* or *count* is zero, with the contents of the array and the state of the stream remaining unchanged.

SEE ALSO

ansiStdio

free( )

NAME

free( ) – free a block of memory (ANSI)

SYNOPSIS

```
void free
    (
    void * ptr /* pointer to block of memory to free */
    )
```

DESCRIPTION

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

RETURNS

N/A

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

freopen( )

NAME *freopen*( ) – open a file specified by name (ANSI)

SYNOPSIS

```
FILE * freopen
    (
    const char * file, /* name of file */
    const char * mode, /* mode */
    FILE *       fp    /* stream */
    )
```

DESCRIPTION This routine opens a file whose name is the string pointed to by *file*and associates it with a specified stream *fp*. The *mode* argument is used just as in the ***fopen*( )** function.

This routine first attempts to close any file that is associated with the specified stream. Failure to close the file successfully is ignored. The error and end-of-file indicators for the stream are cleared.

Typically, ***freopen*( )** is used to attach the already-open streams **stdin**, **stdout**, and **stderr** to other files.

INCLUDE FILES **stdio.h**

RETURNS The value of *fp*, or a null pointer if the open operation fails.

SEE ALSO **ansiStdio**, ***fopen*( )**

frexp( )

NAME *frexp*( ) – break a floating-point number into a normalized fraction and power of 2 (ANSI)

SYNOPSIS

```
double frexp
    (
    double value, /* number to be normalized */
    int *  pexp   /* pointer to the exponent */
    )
```

DESCRIPTION
This routine breaks a double-precision number *value* into a normalized fraction and integral power of 2. It stores the integer exponent in *pexp*.

INCLUDE FILES
math.h

RETURNS
The double-precision value *x*, such that the magnitude of *x* is in the interval [1/2,1] or zero, and *value* equals *x* times 2 to the power of *pexp*. If *value* is zero, both parts of the result are zero.

ERRNO
EDOM

SEE ALSO
ansiMath

fscanf( )

NAME
fscanf( ) – read and convert characters from a stream (ANSI)

SYNOPSIS
```
int fscanf
    (
    FILE *       fp, /* stream to read from */
    char const * fmt /* format string */
    )
```

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, 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 an **n** specifier is included. An input item is defined as the longest matching sequence of input characters, unless it exceeds a specified field width, in which case it is the initial subsequence of that length in the sequence. The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails: this condition is a matching failure, unless an error prevented input from the stream, in which case it is an input failure.

Except in the case of a **%** specifier, the input item is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a *****, the result of the conversion is placed in the object pointed to by the first argument following the *fmt* argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

The following conversion specifiers are valid:

d
: Matches an optionally signed decimal integer whose format is the same as expected for the subject sequence of the ***strtol*( )** function with the value 10 for the *base* argument. The corresponding argument should be a pointer to **int**.

i
: Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the ***strtol*( )** function with the value 0 for the *base* argument. The corresponding argument should be a pointer to **int**.

o
: Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the ***strtoul*( )** function with the value 8 for the *base* argument. The corresponding argument should be a pointer to **unsigned int**.

u
: Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the ***strtoul*( )** function with the value 10 for the *base* argument. The corresponding argument should be a pointer to **unsigned int**.

x
: Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the ***strtoul*( )** function with the value 16 for the *base* argument. The corresponding argument should be a pointer to **unsigned int**.

e, **f**, **g**
: Match an optionally signed floating-point number, whose format is the same as expected for the subject string of the ***strtod*( )** function. The corresponding argument should be a pointer to **float**.

s
: Matches a sequence of non-white-space characters. The corresponding argument should be a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which will be added automatically.

[
: Matches a non-empty sequence of characters from a set of expected characters (the **scanset**). The corresponding argument should be a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which is added automatically. The conversion specifier includes all subsequent character in the format string, up to and including the matching right bracket (**]**). The characters between the brackets (the **scanlist**) comprise the scanset, unless the character after the left bracket is a circumflex (**^**) in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with "[]" or "[^]", the right bracket character is in the scanlist and the next right bracket character is the matching right bracket that ends the specification; otherwise the first right bracket character is the one that ends the specification.

2

c

Matches a sequence of characters of the number specified by the field width (1 if no field width is present in the directive). The corresponding argument should be a pointer to the initial character of an array large enough to accept the sequence. No null character is added.

p

Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the %p conversion of the ***fprintf*( )** function. The corresponding argument should be a pointer to a pointer to **void**. VxWorks defines its pointer input field to be consistent with pointers written by the ***fprintf*( )** function ("0x" hexadecimal notation). If the input item is a value converted earlier during the same program execution, the pointer that results should compare equal to that value; otherwise the behavior of the %p conversion is undefined.

n

No input is consumed. The corresponding argument should be a pointer to **int** into which the number of characters read from the input stream so far by this call to ***fscanf*( )** is written. Execution of a %n directive does not increment the assignment count returned when ***fscanf*( )** completes execution.

%

Matches a single **%**; no conversion or assignment occurs. The complete conversion specification is %%.

If a conversion specification is invalid, the behavior is undefined.

The conversion specifiers **E**, **G**, and **X** are also valid and behave the same as **e**, **g**, and **x**, respectively.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (if any) is terminated with an input failure.

If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream. Trailing white space (including new-line characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the %n directive.

INCLUDE FILES **stdio.h**

RETURNS The number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure; or EOF if an input failure occurs before any conversion.

SEE ALSO **ansiStdio**, ***scanf*( )**, ***sscanf*( )**

*fseek***( )**

NAME

*fseek***( )** – set the file position indicator for a stream (ANSI)

SYNOPSIS

```
int fseek
    (
    FILE * fp,     /* stream */
    long   offset, /* offset from whence */
    int    whence  /* position to offset from: */
                   /* SEEK_SET = beginning SEEK_CUR */
                   /* current position SEEK_END = end-of-file */
    )
```

DESCRIPTION

This routine sets the file position indicator for a specified stream. For a binary stream, the new position, measured in characters from the beginning of the file, is obtained by adding *offset* to the position specified by *whence*, whose possible values are:

SEEK_SET
the beginning of the file.

SEEK_CUR
the current value of the file position indicator.

SEEK_END
the end of the file.

A binary stream does not meaningfully support *fseek***( )** calls with a *whence* value of **SEEK_END**.

For a text stream, either *offset* is zero, or *offset* is a value returned by an earlier call to *ftell***( )** on the stream, in which case *whence* should be **SEEK_SET**.

A successful call to *fseek***( )** clears the end-of-file indicator for the stream and undoes any effects of *ungetc***( )** on the same stream. After an *fseek***( )** call, the next operation on an update stream can be either input or output.

INCLUDE FILES

stdio.h

RETURNS

Non-zero only for a request that cannot be satisfied.

ERRNO

EINVAL

SEE ALSO

ansiStdio, *ftell***( )**

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.

On return, the **stat** structure fields 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.

Upon return, the fields in the **statfs** structure are updated to reflect the characteristics of the file.

RETURNS

OK or ERROR.

SEE ALSO

dirLib, ***statfs*( )**, ***ls*( )**

ftell( )

NAME

ftell( ) – return the current value of the file position indicator for a stream (ANSI)

SYNOPSIS

```
long ftell
    (
    FILE * fp /* stream */
    )
```

DESCRIPTION

This routine returns the current value of the file position indicator for a specified stream. For a binary stream, the value is the number of characters from the beginning of the file. For a text stream, the file position indicator contains unspecified information, usable by ***fseek*( )** for returning the file position indicator to its position at the time of the ***ftell*( )** call; the difference between two such return values is not necessary a meaningful measure of the number of characters written or read.

INCLUDE FILES

stdio.h

RETURNS

The current value of the file position indicator, or -1L if unsuccessful, with **errno** set to indicate the error.

SEE ALSO

ansiStdio, *fseek***( )**

*ftpCommand***( )**

NAME

*ftpCommand***( )** – send an FTP command and get the reply

SYNOPSIS

```
int ftpCommand
    (
    int    ctrlSock, /* fd of control connection socket */
    char * fmt,      /* format string of command to send */
    int    arg1,     /* first of six args to format string */
    int    arg2,
    int    arg3,
    int    arg4,
    int    arg5,
    int    arg6
    )
```

DESCRIPTION

This 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***( )**

2

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***( )** – terminate the FTP server task

SYNOPSIS

```
STATUS ftpdDelete (void)
```

DESCRIPTION This routine halts the FTP server and closes the control connection. All client sessions are removed after completing any commands in progress. When this routine executes, no further client connections will be accepted until the server is restarted. This routine is not reentrant and must not be called from interrupt level.

NOTE If any file transfer operations are in progress when this routine is executed, the transfers will be aborted, possibly leaving incomplete files on the destination host.

RETURNS OK if shutdown completed, or ERROR otherwise.

ERRNO N/A

SEE ALSO **ftpdLib**

ftpdInit( )

NAME *ftpdInit***( )** – initialize the FTP server task

SYNOPSIS

```
STATUS ftpdInit
    (
    FUNCPTR pLoginRtn, /* user verification routine, or NULL */
    int     stackSize  /* task stack size, or 0 for default */
    )
```

DESCRIPTION

This routine installs the password verification routine indicated by *pLoginRtn* and establishes a control connection for the primary FTP server task, which it then creates. It is called automatically during system startup if **INCLUDE_FTP_SERVER** is defined. The primary server task supports simultaneous client sessions, up to the limit specified by the global variable **ftpsMaxClients**. The default value allows a maximum of four simultaneous connections. The *stackSize* argument specifies the stack size for the primary server task. It is set to the value specified in the **ftpdWorkTaskStackSize** global variable by default.

RETURNS

OK if server started, or ERROR otherwise.

ERRNO

N/A

SEE ALSO

ftpdLib

---

ftpHookup( )

NAME

ftpHookup( ) – get a control connection to the FTP server on a specified host

SYNOPSIS

```
int ftpHookup
    (
    char * host /* server host name or inet address */
    )
```

DESCRIPTION

This routine establishes a control connection to the FTP server on the specified host. This is the first step in interacting with a remote FTP server at the lowest level. (For a higher-level interaction with a remote FTP server, see the manual entry for ***ftpXfer*( )**.)

RETURNS

The file descriptor of the control socket, or ERROR if the Internet address or the host name is invalid, if a socket could not be created, or if a connection could not be made.

SEE ALSO

ftpLib, *ftpLogin*( ), *ftpXfer*( )

ftpLogin( )

NAME

ftpLogin( ) – log in to a remote FTP server

SYNOPSIS

```
STATUS ftpLogin
    (
    int    ctrlSock, /* fd of login control socket */
    char * user,     /* user name for host login */
    char * passwd,   /* password for host login */
    char * account   /* account for host login */
    )
```

DESCRIPTION

This routine logs in to a remote server with the specified user name, password, and account name, as required by the specific remote host. This is typically the next step after calling *ftpHookup*( ) in interacting with a remote FTP server at the lowest level. (For a higher-level interaction with a remote FTP server, see the manual entry for *ftpXfer*( )).

RETURNS

OK, or ERROR if the routine is unable to log in.

SEE ALSO

ftpLib, *ftpHookup*( ), *ftpXfer*( )

ftpLs( )

NAME

ftpLs( ) – list directory contents via FTP

SYNOPSIS

```
STATUS ftpLs
    (
    char * dirName /* name of directory to list */
    )
```

DESCRIPTION

This routine lists the contents of a directory. The content list is obtained via an NLST FTP transaction.

The local device name must be the same as the remote host name with a colon ":" as a suffix. (For example "wrs:" is the device name for the "wrs" host.)

RETURNS

OK, or ERROR if could not open directory.

SEE ALSO

ftpLib

ftpReplyGet( )

NAME *ftpReplyGet*( ) – get an FTP command reply

SYNOPSIS

```
int ftpReplyGet
    (
    int  ctrlSock, /* control socket fd of FTP connection */
    BOOL expecteof /* TRUE = EOF expected, FALSE = EOF is error */
    )
```

DESCRIPTION This routine 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 connection) */
    )
```

DESCRIPTION This routine initiates a transfer via a remote FTP server in the following order:

(1) Establishes a connection to the FTP server on the specified host.

(2) Logs in with the specified user name, password, and account, as necessary for the particular host.

(3) Sets the transfer type to image by sending the command "TYPE I".

(4) Changes to the specified directory by sending the command "CWD *dirname*".

(5) Sends the specified transfer command with the specified filename as an argument, and establishes a data connection. Typical transfer commands are "STOR %s", to write to a remote file, or "RETR %s", to read a remote file.

The resulting control and data connection file descriptors are returned via *pCtrlSock* and *pDataSock*, respectively.

After calling this routine, the data can be read or written to the remote server by reading or writing on the file descriptor returned in *pDataSock*. When all incoming data has been read (as indicated by an EOF when reading the data socket) and/or all outgoing data has been written, the data socket fd should be closed. The routine ***ftpReplyGet*( )** should then be called to receive the final reply on the control socket, after which the control socket should be closed.

If the FTP command does not involve data transfer, *pDataSock* should be NULL, in which case no data connection will be established. The only FTP commands supported for this case are DELE, RMD, and MKD.

EXAMPLE The following code fragment reads the file "/usr/fred/myfile" from the host "server", logged in as user "fred", with password "magic" and no account name.

```
#include "vxWorks.h"
#include "ftpLib.h"
int       ctrlSock;
int       dataSock;
char      buf [512];
int       nBytes;
STATUS    status;
if (ftpXfer ("server", "fred", "magic", "",
```

```
                  "RETR %s", "/usr/fred", "myfile",
                  &ctrlSock, &dataSock) == ERROR)
        return (ERROR);
    while ((nBytes = read (dataSock, buf, sizeof (buf))) > 0)
        {
        ...
        }
    close (dataSock);
    if (nBytes < 0)             /* read error? */
        status = ERROR;
    if (ftpReplyGet (ctrlSock, TRUE) != FTP_COMPLETE)
        status = ERROR;
    if (ftpCommand (ctrlSock, "QUIT", 0, 0, 0, 0, 0, 0) != FTP_COMPLETE)
        status = ERROR;
    close (ctrlSock);
```

RETURNS OK, or ERROR if any socket cannot be created or if a connection cannot be made.

SEE ALSO **ftpLib**, *ftpReplyGet*( )

---

ftruncate( )

NAME *ftruncate*( ) – truncate a file (POSIX)

SYNOPSIS

```
int ftruncate
    (
    int   fildes, /* fd of file to truncate */
    off_t length  /* length to truncate file */
    )
```

DESCRIPTION This routine truncates a file to a specified size.

RETURNS 0 (OK) or -1 (ERROR) if unable to truncate file.

ERRNO
EROFS
– File resides on a read-only file system.
EBADF
– File is open for reading only.
EINVAL
– File descriptor refers to a file on which this operation is impossible.

SEE ALSO **ftruncate**

fwrite( )

NAME

fwrite( ) – write from a specified array (ANSI)

SYNOPSIS

```
int fwrite
    (
    const void * buf,   /* where to copy from */
    size_t       size,  /* element size */
    size_t       count, /* no. of elements */
    FILE *       fp     /* stream to write to */
    )
```

DESCRIPTION

This routine writes, from the array *buf*, up to *count* elements whose size is *size*, to a specified stream. The file position indicator for the stream (if defined) is advanced by the number of characters successfully written. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate.

INCLUDE FILES

stdio.h

RETURNS

The number of elements successfully written, which will be less than *count* only if a write error is encountered.

SEE ALSO

ansiStdio

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

---

gethostname( )

NAME *gethostname*( ) – get the symbolic name of this machine

SYNOPSIS

```
int gethostname
    (
    char * name,   /* machine name */
    int    nameLen /* length of name */
    )
```

DESCRIPTION This routine gets the target machine's symbolic name, which can be used for identification.

RETURNS OK or ERROR.

SEE ALSO **hostLib**

---

getpeername( )

NAME *getpeername*( ) – get the name of a connected peer

SYNOPSIS

```
STATUS getpeername
    (
    int               s,      /* socket descriptor */
    struct sockaddr * name,   /* where to put name */
    int *             namelen /* space available in name, later filled in */
                              /* actual name size */
    )
```

DESCRIPTION This routine gets the name of the peer connected to socket *s*. The parameter *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 size of the socket name is copied to *namelen*.

RETURNS OK, or ERROR if the socket is invalid or not connected.

SEE ALSO **sockLib**

gets( )

NAME

gets( ) – read characters from the standard input stream (ANSI)

SYNOPSIS

```
char * gets
    (
    char * buf /* output array */
    )
```

DESCRIPTION

This routine reads characters from the standard input stream into the array *buf* until end-of-file is encountered or a new-line is read. Any new-line character is discarded, and a null character is written immediately after the last character read into the array.

If end-of-file is encountered and no characters have been read, the contents of the array remain unchanged. If a read error occurs, the array contents are indeterminate.

INCLUDE FILES

stdio.h

RETURNS

A pointer to *buf*, or a null pointer if (1) end-of-file is encountered and no characters have been read, or (2) there is a read error.

SEE ALSO

ansiStdio

getsockname( )

NAME

getsockname( ) – get a socket name

SYNOPSIS

```
STATUS getsockname
    (
    int               s,      /* socket descriptor */
    struct sockaddr * name,   /* where to return name */
    int *             namelen /* space available in name, later filled in */
                              /* actual name size */
    )
```

DESCRIPTION

This routine gets the current name for the specified socket *s*. The parameter *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 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. The parameter *optlen* should be initialized to indicate the amount of space referenced by *optval*. On return, the value of the option is copied to *optval* and the actual size of the option is copied to *optlen*.

Although *optval* is passed as a char *, the actual variable whose address gets passed in should be an integer or a structure, depending on which *optname* is being passed. Refer to ***setsockopt*( )** to determine the correct type of the actual variable (whose address should then be cast to a char *).

RETURNS

OK, or ERROR if there is an invalid socket, an unknown option, or the call is unable to get the specified option.

EXAMPLE

Because **SO_REUSEADDR** has an integer parameter, the variable to be passed to *getsockopt*( ) should be declared as

```
int reuseVal;
```

and passed in as

```
(char *)&reuseVal.
```

Otherwise the user might mistakenly declare **reuseVal** as a character, in which case *getsockopt*( ) will only return the first byte of the integer representing the state of this option. Then whether the return value is correct or always 0 depends on the endian-ness of the machine.

SEE ALSO

sockLib, *setsockopt*( )

getw( )

NAME

getw( ) – read the next word (32-bit integer) from a stream

SYNOPSIS

```
int getw
    (
    FILE * fp /* stream to read from */
    )
```

DESCRIPTION

This routine reads the next 32-bit quantity from a specified stream. It returns EOF on end-of-file or an error; however, this is also a valid integer, thus ***feof*( )** and ***ferror*( )** must be used to check for a true end-of-file.

This routine is provided for compatibility with earlier VxWorks releases.

INCLUDE FILES

stdio.h

RETURN

A 32-bit number from the stream, or EOF on either end-of-file or an error.

SEE ALSO

ansiStdio, ***putw*( )**

getwd( )

NAME

getwd( ) – get the current default path

SYNOPSIS

```
char *getwd
    (
    char * pathname /* where to return the pathname */
    )
```

DESCRIPTION

This routine copies the name of the current default path to *pathname*. It provides the same functionality as ***ioDefPathGet*( )** and ***getcwd*( )**. It is provided for compatibility with some older UNIX systems.

The parameter *pathname* should be **MAX_FILENAME_LENGTH** characters long.

RETURNS

A pointer to the resulting path name.

SEE ALSO

ioLib

2

gmtime( )

NAME *gmtime*( ) – convert calendar time into UTC broken-down time (ANSI)

SYNOPSIS

```
struct tm *gmtime
    (
    const time_t * timer /* calendar time in seconds */
    )
```

DESCRIPTION This routine converts the calendar time pointed to by *timer* into broken-down time, expressed as Coordinated Universal Time (UTC).

This routine is not reentrant. For a reentrant version, see ***gmtime_r*( )**.

INCLUDE FILES **time.h**

RETURNS A pointer to a broken-down time structure (**tm**), or a null pointer if UTC is not available.

SEE ALSO **ansiTime**

gmtime_r( )

NAME *gmtime_r*( ) – convert calendar time into broken-down time (POSIX)

SYNOPSIS

```
int gmtime_r
    (
    const time_t * timer,     /* calendar time in seconds */
    struct tm *    timeBuffer /* buffer for broken down time */
    )
```

DESCRIPTION This routine converts the calendar time pointed to by *timer* into broken-down time, expressed as Coordinated Universal Time (UTC). The broken-down time is stored in *timeBuffer*.

This routine is the POSIX re-entrant version of ***gmtime*( )**.

INCLUDE FILES **time.h**

RETURNS OK.

SEE ALSO **ansiTime**

h( )

NAME

h( ) – display or set the size of shell history

SYNOPSIS

```
void h
    (
    int size /* 0 = display, >0 = set history to new size */
    )
```

DESCRIPTION

This command displays or sets the size of VxWorks shell history. If no argument is specified, shell history is displayed. If *size* is specified, that number of the most recent commands is saved for display. The value of *size* is initially 20.

RETURNS

N/A

SEE ALSO

usrLib, ***shellHistory*( )**, **ledLib**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

hdrBlkBuild( )

NAME

***hdrBlkBuild*( )** – create the header block and the demuxer information

SYNOPSIS

```
void hdrBlkBuild
    (
    SA_HEADER_T * hdr,    /* header block */
    VBL_T *       vblist, /* vblist that was built */
    int           opt,    /* reg_option suggesting reg/dereg */
    int           group,  /* group ID */
    PTR_T         saId    /* ipchandle */
    )
```

DESCRIPTION

This routine is called to start a process that encodes a message and transmits it to the master agent. Internally, this routine first prepares a header block and demuxer information. These are then passed in to a ***saMsgBuild*( )** call, along with a varbind list, and a pointer to the IPC mechanism that the master agent can use to respond to this message. As input, ***hdrBlkBuild*( )** expects:

hdr

Expects a pointer to a previously allocated **SA_HEADER_T** structure. The ***hdrBlkBuild*( )** routine uses this structure as a storage place within which to build the header block for the message to the master agent.

vblist
Expects a pointer to the **VBL_T** structure containing the varbind list that you want to include in the message.

opt
Expects an operation code that indicates the type of this message. Valid operation codes are as follows:

SA_REG_OBJ_REQUEST registers an object with the master agent's MIB tree. The response from the master agent will contain an **SA_REG_OBJ_REPLY** code.

SA_REM_OBJ_REQUEST removes (deregisters) an object from the master agent's MIB tree. The response from the master agent will contain an **SA_REM_OBJ_REPLY** code.

SA_REG_INST_REQUEST registers an instance with the master agent's MIB tree. The response from the master agent will contain an **SA_REG_INST_REPLY** code.

SA_REM_INST_REQUEST removes (deregisters) an instance from the master agent's MIB tree. The response from the master agent will contain an **SA_REG_OBJ_REPLY** code.

SA_QUERY_REQUEST requests SNMP operations. The response from the master agent will contain an **SA_QUERY_REPLY** code.

SA_TRAP_REQUEST tells the master agent that this message should be handled as a trap. The response from the master agent (if any) will contain an **SA_TRAP_REPLY** code.

group
Expects the group ID that the master agent has assigned to the objects or instances referenced in *vblist*. This group ID was returned in an If **SA_REG_OBJ_REPLY** or an **SA_REG_INST_REPLY** from the master agent. this is an object registration request, you can supply a NULL pointer here.

said
Expects a pointer to the IPC mechanism that the master agent can use to respond to the message.

RETURNS N/A

SEE ALSO **saIoLib**

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*, **windsh**, *Tornado User's Guide: Shell*

---

hostAdd( )

NAME *hostAdd*( ) – add a host to the host table

SYNOPSIS

```
STATUS hostAdd
    (
    char * hostName, /* host name */
    char * hostAddr  /* host addr in standard Internet format */
    )
```

DESCRIPTION This routine adds a host name to the local host table. This must be called before sockets on the remote host are opened, or before files on the remote host are accessed via **netDrv** or nfsDrv.

The host table has one entry per Internet address. More than one name may be used for an address. Additional host names are added as aliases.

EXAMPLE

```
-> hostAdd "wrs", "90.2"
-> hostShow
hostname          inet address       aliases
--------          ------------       -------
localhost         127.0.0.1
yuba              90.0.0.3
wrs               90.0.0.2
value = 12288 = 0x3000 = _bzero + 0x18
```

RETURNS OK, or ERROR if the host table is full, the host name/inet address pair is already entered, the Internet address is invalid, or memory is insufficient.

SEE ALSO **hostLib**, **netDrv**, **nfsDrv**

hostDelete( )

NAME *hostDelete*( ) – delete a host from the host table

SYNOPSIS

```
STATUS hostDelete
    (
    char * name, /* host name or alias */
    char * addr  /* host addr in standard Internet format */
    )
```

DESCRIPTION This routine deletes a host name from the local host table. If *name* is a host name, the host entry is deleted. If *name* is a host name alias, the alias is deleted.

RETURNS OK, or ERROR if the parameters are invalid or the host is unknown.

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. If the DNS resolver library **resolvLib** has been configured in the vxWorks image, a query for the host name is sent to the DNS server, if the name was not found in the local host table.

WARNING This routine does not look for aliases. Host names are limited to MAXHOSTNAMELEN (from **hostLib.h**) characters.

RETURNS OK, or ERROR if buffer is invalid or the host is unknown.

SEE ALSO **hostLib**, *hostGetByName*( )

2

hostGetByName( )

NAME *hostGetByName*( ) – look up a host in the host table by its name

SYNOPSIS
```
int hostGetByName
    (
    char * name /* name of host */
    )
```

DESCRIPTION This routine returns the Internet address of a host that has been added to the host table by *hostAdd*( ). If the DNS resolver library **resolvLib** has been configured in the vxWorks image, a query for the host IP address is sent to the DNS server, if the name was not found in the local host table.

RETURNS The Internet address (as an integer in network byte order), 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 the configuration macro **INCLUDE_NET_INIT** is defined.

RETURNS
N/A

SEE ALSO
hostLib, usrConfig

---

i( )

NAME
***i*( )** – print a summary of each task's TCB

SYNOPSIS
```
void i
    (
    int taskNameOrId /* task name or task ID, 0 = summarize all */
    )
```

DESCRIPTION
This command displays a synopsis of all the tasks in the system. The ***ti*( )** routine provides more complete information on a specific task.

Both ***i*( )** and ***ti*( )** use ***taskShow*( )**; see the documentation for ***taskShow*( )** for a description of the output format.

EXAMPLE
```
-> i
   NAME        ENTRY       TID    PRI   STATUS     PC       SP     ERRNO DELAY
---------- ---------- -------- --- --------- ------- -------- ----- -----
tExcTask   _excTask    20fcb00   0 PEND      200c5fc  20fca6c     0     0
tLogTask   _logTask    20fb5b8   0 PEND      200c5fc  20fb520     0     0
tShell     _shell      20efcac   1 READY     201dc90  20ef980     0     0
tRlogind   _rlogind    20f3f90   2 PEND      2038614  20f3db0     0     0
tTelnetd   _telnetd    20f2124   2 PEND      2038614  20f2070     0     0
tNetTask   _netTask    20f7398  50 PEND      2038614  20f7340     0     0
value = 57 = 0x39 = '9'
```

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*

2

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 /* pointer to device */
    )
```

DESCRIPTION This routine is called to reset the chip in a quiescent state.

RETURNS N/A

SEE ALSO **i8250Sio**

i8250Int( )

NAME

i8250Int( ) – handle a receiver/transmitter interrupt

SYNOPSIS

```
void i8250Int
    (
    I8250_CHAN * pChan
    )
```

DESCRIPTION

This routine handles four sources of interrupts from the UART. They are prioritized in the following order by the Interrupt Identification Register: Receiver Line Status, Received Data Ready, Transmit Holding Register Empty and Modem Status.

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. disable the interrupt. When a modem status interrupt occurs, the transmit interrupt is enabled if the CTS signal is TRUE.

RETURNS

N/A

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***( )**

---

icmpShowInit( )

NAME *icmpShowInit***( )** – initialize ICMP show routines

SYNOPSIS

```
void icmpShowInit (void)
```

DESCRIPTION This routine links the ICMP show facility into the VxWorks system. These routines are included automatically if **INCLUDE_NET_SHOW** and **INCLUDE_ICMP** are defined in **configAll.h**.

RETURNS N/A

SEE ALSO **icmpShow**

---

icmpstatShow( )

NAME *icmpstatShow***( )** – display statistics for ICMP

SYNOPSIS

```
void icmpstatShow (void)
```

DESCRIPTION This routine displays statistics for the ICMP (Internet Control Message Protocol) protocol.

RETURNS N/A

SEE ALSO **icmpShow**

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**

ifAddrAdd( )

NAME *ifAddrAdd*( ) – Add an interface address for a network interface

SYNOPSIS

```
STATUS ifAddrAdd
    (
    char * interfaceName,    /* name of interface to configure */
    char * interfaceAddress, /* Internet address to assign to interface */
    char * broadcastAddress, /* broadcast address to assign to interface */
    int    subnetMask        /* subnetMask */
    )
```

DESCRIPTION This routine assigns an Internet address to a specified network interface. The Internet address can be a host name or a standard Internet address format (e.g., 90.0.0.4). If a host name is specified, it should already have been added to the host table with ***hostAdd*( )**. *interfaceName*, *interfaceAddress* must be specified. *broadcastAddress* is optional. If *broadcastAddress* is NULL, ***in_ifinit*( )** will generate a *broadcastAddress* by using the *interfaceAddress* and the netmask. *subnetMask* is optional. If *subnetMask* is 0, ***in_ifinit*( )** will set a *subnetMask* as same as a netmask which is generated by the *interfaceAddress*. *broadcastAddress* is also *destAddress* in case of **IFF_POINTOPOINT**.

RETURNS OK, or ERROR if the interface cannot be set.

SEE ALSO **ifLib**, *ifAddrGet*( ), *ifDstAddrSet*( ), *ifDstAddrGet*( )

ifAddrGet( )

NAME *ifAddrGet*( ) – get the Internet address of a network interface

SYNOPSIS

```
STATUS ifAddrGet
    (
    char * interfaceName,   /* name of interface, i.e. ei0 */
    char * interfaceAddress /* buffer for Internet address */
    )
```

DESCRIPTION This routine gets the Internet address of a specified network interface and copies it to *interfaceAddress*, which should point to a buffer large enough for **INET_ADDR_LEN** bytes.

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, i.e. ei0 */
    char * interfaceAddress /* Internet address to assign to interface */
    )
```

DESCRIPTION

This routine assigns an Internet address to a specified network interface. The Internet address can be a host name or a standard Internet address format (e.g., 90.0.0.4). If a host name is specified, it should already have been added to the host table with ***hostAdd*( )**.

A successful call to ***ifAddrSet*( )** results in the addition of a new route.

The subnet mask used in determining the network portion of the address will be that set by ***ifMaskSet*( )**, or the default class mask if ***ifMaskSet*( )** has not been called. It is standard practice to call ***ifMaskSet*( )** prior to calling ***ifAddrSet*( )**.

RETURNS

OK, or ERROR if the interface cannot be set.

SEE ALSO

ifLib, *ifAddrGet*( ), *ifDstAddrSet*( ), *ifDstAddrGet*( )

ifBroadcastGet( )

NAME

ifBroadcastGet( ) – get the broadcast address for a network interface

SYNOPSIS

```
STATUS ifBroadcastGet
    (
    char * interfaceName,   /* name of interface, i.e. ei0 */
    char * broadcastAddress /* buffer for broadcast address */
    )
```

DESCRIPTION

This routine gets the broadcast address for a specified network interface. The broadcast address is copied to the buffer *broadcastAddress*.

RETURNS

OK or ERROR.

SEE ALSO

ifLib, *ifBroadcastSet*( )

ifBroadcastSet( )

NAME

ifBroadcastSet( ) – set the broadcast address for a network interface

SYNOPSIS

```
STATUS ifBroadcastSet
    (
    char * interfaceName,   /* name of interface to assign, i.e. ei0 */
    char * broadcastAddress /* broadcast address to assign to interface */
    )
```

DESCRIPTION

This routine assigns a broadcast address for the specified network interface. The broadcast address must be a string in standard Internet address format (e.g., 90.0.0.0).

An interface's default broadcast address is its Internet address with a host part of all ones (e.g., 90.255.255.255). This conforms to current ARPA specifications. However, some older systems use an Internet address with a host part of all zeros as a broadcast address.

NOTE

VxWorks automatically accepts a host part of all zeros as a broadcast address, in addition to the default or specified broadcast address. But if VxWorks is to broadcast to older systems using a host part of all zeros as the broadcast address, this routine should be used to change the broadcast address of the interface.

RETURNS

OK or ERROR.

SEE ALSO

ifLib

ifDstAddrGet( )

NAME

ifDstAddrGet( ) – get the Internet address of a point-to-point peer

SYNOPSIS

```
STATUS ifDstAddrGet
    (
    char * interfaceName, /* name of interface, i.e. ei0 */
    char * dstAddress     /* buffer for destination address */
    )
```

DESCRIPTION

This routine gets the Internet address of a machine connected to the opposite end of a point-to-point network connection. The Internet address is copied to *dstAddress*.

RETURNS

OK or ERROR.

SEE ALSO **ifLib**, *ifDstAddrSet***( )**, *ifAddrGet***( )**

*ifDstAddrSet***( )**

NAME *ifDstAddrSet***( )** – define an address for the other end of a point-to-point link

SYNOPSIS
```
STATUS ifDstAddrSet
    (
    char * interfaceName, /* name of interface to configure, i.e. ei0 */
    char * dstAddress     /* Internet address to assign to destination */
    )
```

DESCRIPTION This routine assigns the Internet address of a machine connected to the opposite end of a point-to-point network connection, such as a SLIP connection. Inherently, point-to-point connection-oriented protocols such as SLIP require that addresses for both ends of a connection be specified.

RETURNS OK or ERROR.

SEE ALSO **ifLib**, *ifAddrSet***( )**, *ifDstAddrGet***( )**

*ifFlagChange***( )**

NAME *ifFlagChange***( )** – change the network interface flags

SYNOPSIS
```
STATUS ifFlagChange
    (
    char * interfaceName, /* name of the network interface, i.e. ei0 */
    int    flags,         /* the flag to be changed */
    BOOL   on             /* TRUE=turn on, FALSE=turn off */
    )
```

DESCRIPTION This routine changes the flags for the specified network interfaces. If the parameter *on* is TRUE, the specified flags are turned on; otherwise, they are turned off. The routines *ifFlagGet***( )** and *ifFlagSet***( )** are called to do the actual work.

RETURNS OK or ERROR.

SEE ALSO **ifLib**, *ifAddrSet***( )**, *ifMaskSet***( )**, *ifFlagSet***( )**, *ifFlagGet***( )**

ifFlagGet( )

NAME

ifFlagGet( ) – get the network interface flags

SYNOPSIS

```
STATUS ifFlagGet
    (
    char * interfaceName, /* name of the network interface, i.e. ei0 */
    int *  flags          /* network flags returned here */
    )
```

DESCRIPTION

This routine gets the flags for a specified network interface. The flags are copied to the buffer *flags*.

RETURNS

OK or ERROR.

SEE ALSO

ifLib, *ifFlagSet*( )

ifFlagSet( )

NAME

ifFlagSet( ) – specify the flags for a network interface

SYNOPSIS

```
STATUS ifFlagSet
    (
    char * interfaceName, /* name of the network interface, i.e. ei0 */
    int    flags          /* network flags */
    )
```

DESCRIPTION

This routine changes the flags for a specified network interface. Any combination of the following flags can be specified:

IFF_UP (0x1)
Brings the network up or down.

IFF_DEBUG (0x4)
Turns on debugging for the driver interface if supported.

IFF_LOOPBACK (0x8)
Set for a loopback network.

IFF_NOTRAILERS (0x20)
Always set (VxWorks does not use the trailer protocol).

IFF_PROMISC (0x100)
Tells the driver to accept all packets, not just broadcast packets and packets addressed to itself.

IFF_ALLMULTI (0x200)
Tells the driver to accept all multicast packets.

IFF_NOARP (0x80)
Disables ARP for the interface.

NOTE
The following flags can only be set at interface initialization time. Specifying these flags does not change any settings in the interface data structure.

IFF_POINTOPOINT (0x10)
Identifies a point-to-point interface such as PPP or SLIP.

IFF_RUNNING (0x40)
Set when the device turns on.

IFF_BROADCAST (0x2)
Identifies a broadcast interface.

RETURNS
OK or ERROR.

SEE ALSO
ifLib, *ifFlagChange*( ), *ifFlagGet*( )

ifMaskGet( )

NAME
ifMaskGet( ) – get the subnet mask for a network interface

SYNOPSIS
```
STATUS ifMaskGet
    (
    char * interfaceName, /* name of interface, i.e. ei0 */
    int *  netMask        /* buffer for subnet mask */
    )
```

DESCRIPTION
This routine gets the subnet mask for a specified network interface. The subnet mask is copied to the buffer *netMask*. The subnet mask is returned in host byte order.

RETURNS
OK or ERROR.

SEE ALSO
ifLib, *ifAddrGet*( ), *ifFlagGet*( )

ifMaskSet( )

NAME
ifMaskSet( ) – define a subnet for a network interface

SYNOPSIS

```
STATUS ifMaskSet
    (
    char * interfaceName, /* name of interface to set mask for, i.e. ei0 */
    int    netMask        /* subnet mask (e.g. 0xff000000) */
    )
```

DESCRIPTION
This routine allocates additional bits to the network portion of an Internet address. The network portion is specified with a mask that must contain ones in all positions that are to be interpreted as the network portion. This includes all the bits that are normally interpreted as the network portion for the given class of address, plus the bits to be added. Note that all bits must be contiguous. The mask is specified in host byte order.

In order to correctly interpret the address, a subnet mask should be set for an interface prior to setting the Internet address of the interface with the routine ***ifAddrSet*( )**.

RETURNS
OK or ERROR.

SEE ALSO
ifLib, *ifAddrSet*( )

ifMetricGet( )

NAME
ifMetricGet( ) – get the metric for a network interface

SYNOPSIS

```
STATUS ifMetricGet
    (
    char * interfaceName, /* name of the network interface, i.e. ei0 */
    int *  pMetric        /* returned interface's metric */
    )
```

DESCRIPTION
This routine retrieves the metric for a specified network interface. The metric is copied to the buffer *pMetric*.

RETURNS
OK or ERROR.

SEE ALSO
ifLib, *ifMetricSet*( )

2

ifMetricSet( )

NAME *ifMetricSet*( ) – specify a network interface hop count

SYNOPSIS

```
STATUS ifMetricSet
    (
    char * interfaceName, /* name of the network interface, i.e. ei0 */
    int    metric         /* metric for this interface */
    )
```

DESCRIPTION This routine configures *metric* for a network interface from the host machine to the destination network. This information is used primarily by the IP routing algorithm to compute the relative distance for a collection of hosts connected to each interface. For example, a higher *metric* for SLIP interfaces can be specified to discourage routing a packet to slower serial line connections. Note that when *metric* is zero, the IP routing algorithm allows for the direct sending of a packet having an IP network address that is not necessarily the same as the local network address.

RETURNS OK or ERROR.

SEE ALSO **ifLib**, *ifMetricGet*( )

ifRouteDelete( )

NAME *ifRouteDelete*( ) – delete routes associated with a network interface

SYNOPSIS

```
int ifRouteDelete
    (
    char * ifName, /* name of the interface */
    int    unit    /* unit number for this interface */
    )
```

DESCRIPTION This routine deletes all routes that have been associated with the specified interface. A route is associated with an interface if its destination equals to the assigned address, or network number. This routine does not remove routes to arbitrary destinations which pass through the given interface.

RETURNS The number of routes deleted, or ERROR if an interface is not specified.

SEE ALSO **ifLib**

ifShow( )

NAME

ifShow( ) – display the attached network interfaces

SYNOPSIS

```
void ifShow
    (
    char * ifName /* name of the interface to show */
    )
```

DESCRIPTION

This routine displays the attached network interfaces for debugging and diagnostic purposes. If *ifName* is given, only the interfaces belonging to that group are displayed. If *ifName* is omitted, all attached interfaces are displayed.

For each interface selected, the following are shown: Internet address, point-to-point peer address (if using SLIP), broadcast address, netmask, subnet mask, Ethernet address, route metric, maximum transfer unit, number of packets sent and received on this interface, number of input and output errors, and flags (such as loopback, point-to-point, broadcast, promiscuous, ARP, running, and debug).

EXAMPLE

The following call displays all interfaces whose names begin with "ln", (such as "ln0", "ln1", and "ln2"):

```
-> ifShow "ln"
```

The following call displays just the interface "ln0":

```
-> ifShow "ln0"
```

RETURNS

N/A

SEE ALSO

netShow, *routeShow*( ), **ifLib**

ifunit( )

NAME

ifunit( ) – map an interface name to an interface structure pointer

SYNOPSIS

```
struct ifnet *ifunit
    (
    char * ifname /* name of the interface */
    )
```

DESCRIPTION This routine returns a pointer to a network interface structure for *name* or NULL if no such interface exists. For example:

```
struct ifnet *pIf;
...
pIf = ifunit ("ln0");
```

pIf points to the data structure that describes the first network interface device if ln0 is mapped successfully.

RETURNS A pointer to the interface structure, or NULL if an interface is not found.

SEE ALSO **ifLib**, **etherLib**

igmpShowInit( )

NAME *igmpShowInit*( ) – initialize IGMP show routines

SYNOPSIS `void igmpShowInit (void)`

DESCRIPTION This routine links the IGMP show facility into the VxWorks system. These routines are included automatically if **INCLUDE_NET_SHOW** and **INCLUDE_IGMP** are defined in **configAll.h**.

RETURNS N/A

SEE ALSO **igmpShow**

igmpstatShow( )

NAME *igmpstatShow*( ) – display statistics for IGMP

SYNOPSIS `void igmpstatShow (void)`

DESCRIPTION This routine displays statistics for the IGMP (Internet Group Management Protocol) protocol.

RETURNS N/A

SEE ALSO **igmpShow**

index( )

NAME

index( ) – find the first occurrence of a character in a string

SYNOPSIS

```
char *index
    (
    const char * s, /* string in which to find character */
    int          c  /* character to find in string */
    )
```

DESCRIPTION

This routine finds the first occurrence of character *c* in string *s*.

RETURNS

A pointer to the located character, or NULL if *c* is not found.

SEE ALSO

bLib, *strchr*( ).

inet_addr( )

NAME

inet_addr( ) – convert a dot notation Internet address to a long integer

SYNOPSIS

```
u_long inet_addr
    (
    char * inetString /* string inet address */
    )
```

DESCRIPTION

This routine interprets an Internet address. All the network library routines call this routine to interpret entries in the data bases which are expected to be an address. The value returned is in network order.

EXAMPLE

The following example returns 0x5a000002:

```
    inet_addr ("90.0.0.2");
```

RETURNS

The Internet address, or ERROR.

SEE ALSO

inetLib

inet_aton( )

NAME

inet_aton( ) – convert a network address from dot notation, store in a structure

SYNOPSIS

```
STATUS inet_aton
    (
    char *           pString,   /* string containing address, dot notation */
    struct in_addr * inetAddress /* struct in which to store address */
    )
```

DESCRIPTION

This routine interprets an Internet address. All the network library routines call this routine to interpret entries in the data bases that are expected to be an address. The value returned is stored in network byte order in the structure provided.

EXAMPLE

The following example returns 0x5a000002 in the **s_addr** member of the structure pointed to by *pinetAddr*:

```
inet_addr ("90.0.0.2", pinetAddr);
```

RETURNS

OK, 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 an Internet address from network and host numbers

SYNOPSIS

```
struct in_addr inet_makeaddr
    (
    int netAddr, /* network part of the address */
    int hostAddr /* host part of the address */
    )
```

DESCRIPTION

This routine constructs the Internet address from the network number and local host address.

WARNING

This routine is supplied for UNIX compatibility only. Each time this routine is called, four bytes are allocated from memory. Use ***inet_makeaddr_b*( )** instead.

EXAMPLE

The following example returns the address 0x5a000002 to the structure **in_addr**:

```
    inet_makeaddr (0x5a, 2);
```

RETURNS

The network address in an **in_addr** structure.

SEE ALSO

inetLib, *inet_makeaddr_b*( )

inet_makeaddr_b( )

NAME

inet_makeaddr_b( ) – form an Internet address from network and host numbers

SYNOPSIS

```
void inet_makeaddr_b
    (
    int              netAddr,  /* network part of the inet address */
    int              hostAddr, /* host part of the inet address */
    struct in_addr * pInetAddr /* where to return the inet address */
    )
```

DESCRIPTION

This routine constructs the Internet address from the network number and local host address. This routine is identical to the UNIX ***inet_makeaddr*( )** routine except that you must provide a buffer for the resulting value.

EXAMPLE

The following copies the address 0x5a000002 to the location pointed to by *pInetAddr*:

```
inet_makeaddr_b (0x5a, 2, pInetAddr);
```

RETURNS N/A

SEE ALSO **inetLib**

inet_netof( )

NAME *inet_netof*( ) – return the network number from an Internet address

SYNOPSIS

```
int inet_netof
    (
    struct in_addr inetAddress /* inet address */
    )
```

DESCRIPTION This routine extracts the network portion of an Internet address.

EXAMPLE The following example returns 0x5a:

```
inet_netof (0x5a000002);
```

RETURNS The network portion of *inetAddress*.

SEE ALSO **inetLib**

inet_netof_string( )

NAME *inet_netof_string*( ) – extract the network address in dot notation

SYNOPSIS

```
void inet_netof_string
    (
    char * inetString, /* inet addr to extract local portion from */
    char * netString   /* net inet address to return */
    )
```

DESCRIPTION This routine extracts the network Internet address from a host Internet address (specified in dotted decimal notation). The routine handles class A, B, and C network addresses. The buffer *netString* should be **INET_ADDR_LEN** bytes long.

NOTE This is the only routine in **inetLib** that handles subnet masks correctly.

EXAMPLE The following example copies "90.0.0.0" to *netString*:

```
inet_netof_string ("90.0.0.2", netString);
```

RETURNS N/A

SEE ALSO **inetLib**

inet_network( )

NAME *inet_network*( ) – convert an Internet network number from string to address

SYNOPSIS

```
u_long inet_network
    (
    char * inetString /* string version of inet addr */
    )
```

DESCRIPTION This routine forms a network address from an ASCII string containing an Internet network number.

EXAMPLE The following example returns 0x5a:

```
inet_network ("90");
```

RETURNS The Internet address version of an ASCII string.

SEE ALSO **inetLib**

inet_ntoa( )

NAME *inet_ntoa*( ) – convert a network address to dotted decimal notation

SYNOPSIS

```
char *inet_ntoa
    (
    struct in_addr inetAddress /* inet address */
    )
```

DESCRIPTION This routine converts an Internet address in network format to dotted decimal notation.

WARNING This routine is supplied for UNIX compatibility only. Each time this routine is called, 18 bytes are allocated from memory. Use ***inet_ntoa_b*( )** instead.

EXAMPLE The following example returns a pointer to the string "90.0.0.2":

```
struct in_addr iaddr;
 ...
iaddr.s_addr = 0x5a000002;
 ...
inet_ntoa (iaddr);
```

RETURNS A pointer to the string version of an Internet address.

SEE ALSO **inetLib**, ***inet_ntoa_b*( )**

---

inet_ntoa_b( )

NAME ***inet_ntoa_b*( )** – convert an network address to dot notation, store it in a buffer

SYNOPSIS

```
void inet_ntoa_b
    (
    struct in_addr inetAddress, /* inet address */
    char *         pString      /* where to return ASCII string */
    )
```

DESCRIPTION This routine converts an Internet address in network format to dotted decimal notation.

This routine is identical to the UNIX ***inet_ntoa*( )** routine except that you must provide a buffer of size **INET_ADDR_LEN**.

EXAMPLE The following example copies the string "90.0.0.2" to *pString*:

```
struct in_addr iaddr;
 ...
iaddr.s_addr = 0x5a000002;
 ...
inet_ntoa_b (iaddr, pString);
```

RETURNS N/A

SEE ALSO **inetLib**

inetstatShow( )

NAME *inetstatShow*( ) – display all active connections for Internet protocol sockets

SYNOPSIS

```
void inetstatShow (void)
```

DESCRIPTION This routine displays a list of all active Internet protocol sockets in a format similar to the UNIX **netstat** command.

RETURNS N/A

SEE ALSO **netShow**

infinity( )

NAME *infinity*( ) – return a very large double

SYNOPSIS

```
double infinity (void)
```

DESCRIPTION This routine returns a very large double.

INCLUDE FILES **math.h**

RETURNS The double-precision representation of positive infinity.

SEE ALSO **mathALib**

infinityf( )

NAME *infinityf*( ) – return a very large float

SYNOPSIS

```
float infinityf (void)
```

DESCRIPTION This routine returns a very large float.

INCLUDE FILES **math.h**

RETURNS

The single-precision representation of positive infinity.

SEE ALSO

mathALib

inflate( )

NAME

inflate( ) – inflate compressed code

SYNOPSIS

```
int inflate
    (
    Byte * src,
    Byte * dest,
    int    nBytes
    )
```

DESCRIPTION

This routine inflates *nBytes* of data starting at address *src*. The inflated code is copied starting at address *dest*. Two sanity checks are performed on the data being decompressed. First, we look for a magic number at the start of the data to verify that it is really a compressed stream. Second, the entire data is optionally checksummed to verify its integrity. By default, the checksum is not verified in order to speed up the booting process. To turn on checksum verification, set the global variable **inflateCksum** to TRUE in the BSP.

RETURNS

OK or ERROR.

SEE ALSO

inflateLib

intConnect( )

NAME

intConnect( ) – connect a C routine to a hardware interrupt

SYNOPSIS

```
STATUS intConnect
    (
    VOIDFUNCPTR * vector,   /* interrupt vector to attach to */
    VOIDFUNCPTR   routine,  /* routine to be called */
    int           parameter /* parameter to be passed to routine */
    )
```

DESCRIPTION This routine connects a specified C routine to a specified interrupt vector. The address of *routine* is generally stored at *vector* so that *routine* is called with *parameter* when the interrupt occurs. The routine is invoked in supervisor mode at interrupt level. A proper C environment is established, the necessary registers saved, and the stack set up.

The routine can be any normal C code, except that it must not invoke certain operating system functions that may block or perform I/O operations.

This routine generally simply calls ***intHandlerCreate*( )** and ***intVecSet*( )**. The address of the handler returned by ***intHandlerCreate*( )** is what actually goes in the interrupt vector.

This routine takes an interrupt vector as a parameter, which is the byte offset into the vector table. Macros are provided to convert between interrupt vectors and interrupt numbers, see **intArchLib**.

NOTE ARM ARM processors generally do not have on-chip interrupt controllers. Control of interrupts is a BSP-specific matter. This routine calls a BSP-specific routine to install the handler such that, when the interrupt occurs, *routine* is called with *parameter*.

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, ARM)

SYNOPSIS

```
int intDisable
    (
    int level /* new interrupt bits (0x0 - 0xff00) */
    )
```

DESCRIPTION

On MIPS and PowerPC architectures, this routine disables the corresponding interrupt bits from the present status register.

NOTE ARM

ARM processors generally do not have on-chip interrupt controllers. Control of interrupts is a BSP-specific matter. This routine calls a BSP-specific routine to disable a particular interrupt level, regardless of the current interrupt mask level.

NOTE MIPS

For MIPS, the macros **SR_IBIT1** – **SR_IBIT8** define bits that may be set.

RETURNS

OK or ERROR. (MIPS: The previous contents of the status register).

SEE ALSO

intArchLib

intEnable( )

NAME

intEnable( ) – enable corresponding interrupt bits (MIPS, PowerPC, ARM)

SYNOPSIS

```
int intEnable
    (
    int level /* new interrupt bits (0x00 - 0xff00) */
    )
```

DESCRIPTION

This routine enables the input interrupt bits on the present status register of the MIPS and PowerPC processors.

NOTE ARM

ARM processors generally do not have on-chip interrupt controllers. Control of interrupts is a BSP-specific matter. This routine calls a BSP-specific routine to enable the interrupt. For each interrupt level to be used, there must be a call to this routine before it will be allowed to interrupt.

NOTE MIPS

For MIPS, it is strongly advised that the level be a combination of **SR_IBIT1** – **SR_IBIT8**.

RETURNS OK or ERROR. (MIPS: The previous contents of the status register).

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, ARM)

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
ARM	BSP-specific

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**

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 ARM processors, interrupts (IRQs) are disabled by setting the I bit in the CPSR. This means no IRQs can occur.

For PowerPC processors, there is only one interrupt vector. The external interrupt (vector offset 0x500) is disabled when ***intLock*( )** is called; this means that the processor cannot be interrupted by any external event.

IMPLEMENTATION The lock-out key is implemented differently for different architectures:

MC680x0:	interrupt field mask

SPARC:	interrupt level (0 – 15)
i960:	interrupt level (0 – 31)
MIPS:	status register
i386/i486:	interrupt enable flag (IF) bit from EFLAGS register
PowerPC:	MSR register value
ARM	I bit from the CPSR

WARNINGS

Do not call VxWorks system routines with interrupts locked. Violating this rule may re-enable interrupts unpredictably.

The routine ***intLock*( )** can be called from either interrupt or task level. When called from a task context, the interrupt lock level is part of the task context. Locking out interrupts does not prevent rescheduling. Thus, if a task locks out interrupts and invokes kernel services that cause the task to block (e.g., ***taskSuspend*( )** or ***taskDelay*( )**) or that cause a higher priority task to be ready (e.g., ***semGive*( )** or ***taskResume*( )**), then rescheduling occurs and interrupts are unlocked while other tasks run. Rescheduling may be explicitly disabled with ***taskLock*( )**. Traps must be enabled when calling this routine.

EXAMPLES

```
lockKey = intLock ();
 ... (work with interrupts locked out)
intUnlock (lockKey);
```

To lock out interrupts and task scheduling as well (see WARNING above):

```
if (taskLock() == OK)
    {
    lockKey = intLock ();
    ... (critical section)
    intUnlock (lockKey);
    taskUnlock();
    }
 else
    {
    ... (error message or recovery attempt)
    }
```

RETURNS

An architecture-dependent lock-out key for the interrupt level prior to the call.

SEE ALSO

intArchLib, ***intUnlock*( )**, ***taskLock*( )**, ***intLockLevelSet*( )**

intLockLevelGet( )

NAME
intLockLevelGet( ) – get the current interrupt lock-out level (MC680x0, SPARC, i960, x86, ARM)

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. (ARM = ERROR always)

SEE ALSO
intArchLib, *intLockLevelSet*( )

intLockLevelSet( )

NAME
intLockLevelSet( ) – set the current interrupt lock-out level (MC680x0, SPARC, i960, x86, ARM)

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.

NOTE ARM
On the ARM, this call establishes the interrupt level to be set when ***intLock*( )** is called.

RETURNS
N/A

SEE ALSO
intArchLib, *intLockLevelGet*( ), *intLock*( ), *taskLock*( )

intSRGet( )

NAME *intSRGet*( ) – read the contents of the status register (MIPS)

SYNOPSIS

```
int intSRGet (void)
```

DESCRIPTION This routine reads and returns the contents of the MIPS status register.

RETURNS The previous contents of the status register.

SEE ALSO **intArchLib**

intSRSet( )

NAME *intSRSet*( ) – update the contents of the status register (MIPS)

SYNOPSIS

```
int intSRSet
    (
    int value /* value to write to status register */
    )
```

DESCRIPTION This routine updates and returns the previous contents of the MIPS status register.

RETURNS The previous contents of the status register.

SEE ALSO **intArchLib**

intUninitVecSet( )

NAME *intUninitVecSet*( ) – set the uninitialized vector handler (ARM)

SYNOPSIS

```
void intUninitVecSet
    (
    VOIDFUNCPTR routine /* ptr to user routine */
    )
```

DESCRIPTION This routine installs a handler for the uninitialized vectors to be called when any uninitialised vector is entered.

RETURNS N/A.

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, ARM)

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, ARM = 0 always).

SEE ALSO **intArchLib**, *intVecBaseSet*( )

intVecBaseSet( )

NAME

intVecBaseSet( ) – set the vector (trap) base address (MC680x0, SPARC, i960, x86, MIPS, ARM)

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.

NOTE ARM

The ARM 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).

This routine takes an interrupt vector as a parameter, which is the byte offset into the vector table. Macros are provided to convert between interrupt vectors and interrupt numbers, see **intArchLib**.

NOTE 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*( )**.

This routine takes an interrupt vector as a parameter, which is the byte offset into the vector table. Macros are provided to convert between interrupt vectors and interrupt numbers, see **intArchLib**.

NOTE SPARC

This routine generates code to:

(1) save volatile registers;

(2) fix possible window overflow;

(3) read the processor state register into register %L0; 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, ARM)

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.

ERRNO

S_intLib_VEC_TABLE_WP_UNAVAILABLE

SEE ALSO

intArchLib

ioctl( )

NAME

ioctl( ) – perform an I/O control function

SYNOPSIS

```
int ioctl
    (
    int fd,       /* file descriptor */
    int function, /* function code */
    int arg       /* arbitrary argument */
    )
```

DESCRIPTION

This routine performs an I/O control function on a device. The control functions used by VxWorks device drivers are defined in the header file **ioLib.h**. Most requests are passed on to the driver for handling. Since the availability of *ioctl*( ) functions is driver-specific, these functions are discussed separately in **tyLib**, **pipeDrv**, **nfsDrv**, **dosFsLib**, **rt11FsLib**, and **rawFsLib**.

The following example renames the file or directory to the string "newname":

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

Note that the function **FIOGETNAME** is handled by the I/O interface level and is not passed on to the device driver itself. Thus this function code value should not be used by customer-written drivers.

RETURNS The return value of the driver, or ERROR if the file descriptor does not exist.

SEE ALSO **ioLib**, **tyLib**, **pipeDrv**, **nfsDrv**, **dosFsLib**, **rt11FsLib**, **rawFsLib**, *VxWorks Programmer's Guide: I/O System, Local File Systems*

---

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*( )

---

iOlicomEndLoad( )

NAME *iOlicomEndLoad*( ) – initialize the driver and device

SYNOPSIS

```
END_OBJ * iOlicomEndLoad
    (
    char * initString /* String to be parsed by the driver. */
    )
```

DESCRIPTION This routine initializes the driver and the device to the operational state. All of the device specific parameters are passed in the initString.

This routine can be called in two modes. If it is called with an empty, but allocated string then it places the name of this device (i.e. oli) into the initString and returns 0.

If the string is allocated then the routine attempts to perform its load functionality.

RETURNS An END object pointer or NULL on error or 0 and the name of the device if the initString was NULL.

SEE ALSO **iOlicomEnd**

---

iOlicomIntHandle( )

NAME *iOlicomIntHandle*( ) – interrupt service for card interrupts

SYNOPSIS

```
void iOlicomIntHandle
    (
    END_DEVICE * pDrvCtrl /* pointer to END_DEVICE structure */
    )
```

DESCRIPTION This routine is called when an interrupt has been detected from the Olicom card.

RETURNS N/A.

SEE ALSO **iOlicomEnd**

*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 */
    )
```

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*

2

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*( )

2

ipAttach( )

NAME

ipAttach( ) – a generic attach routine for the TCP/IP network stack

SYNOPSIS

```
int ipAttach
    (
    int    unit,   /* Unit number */
    char * pDevice /* Device name (i.e. ln, ei etc.). */
    )
```

DESCRIPTION

This routine takes the unit number and device name of an END driver (e.g., "ln0", "ei0", etc.) and attaches the TCP/IP stack to the MUX. If completed successfully, the IP protocol will begin receiving packets from that driver.

RETURNS

OK or ERROR

SEE ALSO

ipProto

ipDetach( )

NAME

ipDetach( ) – a generic detach routine for the TCP/IP network stack

SYNOPSIS

```
STATUS ipDetach
    (
    int    unit,   /* Unit number */
    char * pDevice /* Device name (i.e. ln, ei etc.). */
    )
```

DESCRIPTION

This routine removes the TCP/IP stack from the MUX. If completed successfully, the IP protocol will no longer receive packets from the named END driver.

RETURNS

OK or ERROR

SEE ALSO

ipProto

ipFilterHookAdd( )

NAME

ipFilterHookAdd( ) – add a routine to receive all internet protocol packets

SYNOPSIS

```
STATUS ipFilterHookAdd
    (
    FUNCPTR ipFilterHook /* routine to receive raw ip packets */
    )
```

DESCRIPTION

This routine adds a hook routine that will be called for every IP packet that is received.

The calling sequence of the filter hook routine is:

```
    BOOL ipFilterHook
         (
         struct ifnet *pIf,         /* interface packet was received on */
         struct mbuf **pPtrMbuf,    /* pointer to pointer to an mbuf chain */
         struct ip   **pPtrIpHdr,   /* pointer to pointer to ip header */
         int          ipHdrLen,     /* ip packet header length */
         )
```

The hook routine should return TRUE if it has handled the input packet and no further action should be taken with it. If returning TRUE the ipFilterHook is responsible for freeing the mbuf chain by calling m_freem(*pPtrMbuf). It should return FALSE if it has not handled the ipFilterHook and normal processing (e.g., Internet) should take place.

The packet is in a mbuf chain of which a pointer to a pointer is passed as one of the arguments. The pointer to the mbuf should be accessed by dereferencing the pointer to pointer, pPtrMbuf. This mbuf chain will be reused upon return from the hook. If the hook routine needs to retain the input packet, it should copy it elsewhere. by using the macro copy_from_mbufs (buffer, *pPtrMbuf, len). copy_from_mbufs is defined "**net/mbuf.h**"

pPtrIpHdr is a pointer to a pointer to a IP header. The pointer to the ip header is obtained by dereferencing pPtrIpHdr. The ip header is used to examine and process the fields in the ip header. The fields ip_len, ip_id and ip_offset in the ip header are converted to the host byte order from the network byte order before a packet is handed to the filter hook.

The pPtrMbuf and pPtrIpHdr are reused upon return from the hook if it is returning FALSE.

Normally you will not be needing to modify pPtrMbuf or the pPtrIpHdr.

RETURNS

OK, always.

SEE ALSO

ipFilterLib

ipFilterHookDelete( )

NAME *ipFilterHookDelete*( ) – delete a ip filter hook routine

SYNOPSIS

```
void ipFilterHookDelete (void)
```

DESCRIPTION This routine deletes an IP filter hook.

SEE ALSO **ipFilterLib**

ipFilterLibInit( )

NAME *ipFilterLibInit*( ) – initialize ip filter facility

SYNOPSIS

```
void ipFilterLibInit (void)
```

DESCRIPTION This routine links the ip filter facility into the VxWorks system. These routines are included automatically if **INCLUDE_IP_FILTER** is defined in **configAll.h**.

RETURNS N/A

SEE ALSO **ipFilterLib**

ipstatShow( )

NAME *ipstatShow*( ) – display IP statistics

SYNOPSIS

```
void ipstatShow
    (
    BOOL zero /* TRUE = reset statistics to 0 */
    )
```

DESCRIPTION This routine displays detailed statistics for the IP protocol.

RETURNS N/A

SEE ALSO **netShow**

irint( )

NAME
irint( ) – convert a double-precision value to an integer

SYNOPSIS

```
int irint
    (
    double x /* argument */
    )
```

DESCRIPTION
This routine converts a double-precision value *x* to an integer using the selected IEEE rounding direction.

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

2

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, */
                  /* from the last instruction disassembled on the */
                  /* call to l */
    int     count /* number of instruction to disassemble */
                  /* if 0, use the same 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:

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

2

ldiv( )

NAME *ldiv*( ) – compute the quotient and remainder of the division (ANSI)

SYNOPSIS

```
ldiv_t ldiv
    (
    long numer, /* numerator */
    long denom  /* denominator */
    )
```

DESCRIPTION This routine computes the quotient and remainder of *numer*/*denom*. This routine is similar to *div*( ), except that the arguments and the elements of the returned structure are all of type **long**.

This routine is not reentrant. For a reentrant version, see *ldiv_r*( ).

INCLUDE FILES **stdlib.h**

RETURNS A structure of type **ldiv_t**, containing both the quotient and the remainder.

SEE ALSO **ansiStdlib**

ldiv_r( )

NAME *ldiv_r*( ) – compute a quotient and remainder (reentrant)

SYNOPSIS

```
void ldiv_r
    (
    long     numer,       /* numerator */
    long     denom,       /* denominator */
    ldiv_t * divStructPtr /* ldiv_t structure */
    )
```

DESCRIPTION This routine computes the quotient and remainder of *numer*/*denom*. The quotient and remainder are stored in the **ldiv_t** structure **divStructPtr**. This routine is the reentrant version of *ldiv*( ).

INCLUDE FILES **stdlib.h**

RETURNS N/A

SEE ALSO **ansiStdlib**

ledClose( )

NAME

ledClose( ) – discard the line-editor ID

SYNOPSIS

```
STATUS ledClose
    (
    int led_id /* ID returned by ledOpen */
    )
```

DESCRIPTION

This routine frees resources allocated by *ledOpen*( ). The low-level input/output file descriptors are not closed.

RETURNS

OK.

SEE ALSO

ledLib, *ledOpen*( )

ledControl( )

NAME

ledControl( ) – change the line-editor ID parameters

SYNOPSIS

```
void ledControl
    (
    int led_id,  /* ID returned by ledOpen */
    int inFd,    /* new input fd (NONE = no change) */
    int outFd,   /* new output fd (NONE = no change) */
    int histSize /* new history list size (NONE = no change), */
                 /* (0 = display) */
    )
```

DESCRIPTION

This routine changes the input/output file descriptor and the size of the history list.

RETURNS

N/A

SEE ALSO

ledLib

ledOpen( )

NAME

ledOpen( ) – create a new line-editor ID

SYNOPSIS

```
int ledOpen
    (
    int inFd,    /* low-level device input fd */
    int outFd,   /* low-level device output fd */
    int histSize /* size of history list */
    )
```

DESCRIPTION

This routine creates the ID that is used by ***ledRead*( )**, ***ledClose*( )**, and ***ledControl*( )**. Storage is allocated for up to *histSize* previously read lines.

RETURNS

The line-editor ID, or ERROR if the routine runs out of memory.

SEE ALSO

ledLib, *ledRead*( ), *ledClose*( ), *ledControl*( )

ledRead( )

NAME

ledRead( ) – read a line with line-editing

SYNOPSIS

```
int ledRead
    (
    int    led_id,  /* ID returned by ledOpen */
    char * string,  /* where to return line */
    int    maxBytes /* maximum number of chars to read */
    )
```

DESCRIPTION

This routine handles line-editing and history substitutions. If the low-level input file descriptor is not in **OPT_LINE** mode, only an ordinary ***read*( )** routine will be performed.

RETURNS

The number of characters read, or EOF.

SEE ALSO

ledLib

lio_listio( )

NAME

lio_listio( ) – initiate a list of asynchronous I/O requests (POSIX)

SYNOPSIS

```
int lio_listio
    (
    int                 mode,   /* LIO_WAIT or LIO_NOWAIT */
    struct aiocb *      list[], /* list of operations */
    int                 nEnt,   /* size of list */
    struct sigevent * pSig      /* signal on completion */
    )
```

DESCRIPTION

This routine submits a number of I/O operations (up to **AIO_LISTIO_MAX**) to be performed asynchronously. *list* is a pointer to an array of **aiocb** structures that specify the AIO operations to be performed. The array is of size *nEnt*.

The **aio_lio_opcode** field of the **aiocb** structure specifies the AIO operation to be performed. Valid entries include **LIO_READ**, **LIO_WRITE**, and **LIO_NOP**. **LIO_READ** corresponds to a call to *aio_read*( ), **LIO_WRITE** corresponds to a call to *aio_write*( ), and **LIO_NOP** is ignored.

The *mode* argument can be either **LIO_WAIT** or **LIO_NOWAIT**. If *mode* is **LIO_WAIT**, *lio_listio*( ) does not return until all the AIO operations complete and the *pSig* argument is ignored. If *mode* is **LIO_NOWAIT**, the *lio_listio*( ) returns as soon as the operations are queued. In this case, if *pSig* is not NULL and the signal number indicated by **pSig>sigev_signo** is not zero, the signal **pSig>sigev_signo** is delivered when all requests have completed.

RETURNS

OK if requests queued successfully, otherwise ERROR.

ERRNO

EINVAL, **EAGAIN**, **EIO**

INCLUDE FILES

aio.h

SEE ALSO

aioPxLib, *aio_read*( ), *aio_write*( ), *aio_error*( ), *aio_return*( ).

listen( )

NAME

listen( ) – enable connections to a socket

SYNOPSIS

```
STATUS listen
    (
    int s,      /* socket descriptor */
    int backlog /* number of connections to queue */
    )
```

DESCRIPTION

This routine enables connections to a socket. It also specifies the maximum number of unaccepted connections that can be pending at one time (*backlog*). After enabling connections with ***listen*( )**, connections are actually accepted by ***accept*( )**.

RETURNS

OK, or ERROR if the socket is invalid or unable to listen.

SEE ALSO

sockLib

lkAddr( )

NAME

lkAddr( ) – list symbols whose values are near a specified value

SYNOPSIS

```
void lkAddr
    (
    unsigned int addr /* address around which to look */
    )
```

DESCRIPTION

This command lists the symbols in the system symbol table that are near a specified value. The symbols that are displayed include:

- symbols whose values are immediately less than the specified value
- symbols with the specified value
- succeeding symbols, until at least 12 symbols have been displayed

This command also displays symbols that are local, i.e., symbols found in the system symbol table only because their module was loaded by ***ld*( )**.

RETURNS

N/A

SEE ALSO

usrLib, **symLib**, ***symEach*( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

*lkup***( )**

NAME

*lkup***( )** – list symbols

SYNOPSIS

```
void lkup
    (
    char * substr /* substring to match */
    )
```

DESCRIPTION

This command lists all symbols in the system symbol table whose names contain the string *substr*. If *substr* is omitted or is 0, a short summary of symbol table statistics is printed. If *substr* is the empty string (""), all symbols in the table are listed.

This command also displays symbols that are local, i.e., symbols found in the system symbol table only because their module was loaded by ***ld*( )**.

By default, ***lkup*( )** displays 22 symbols at a time. This can be changed by modifying the global variable **symLkupPgSz**. If this variable is set to 0, ***lkup*( )** displays all the symbols without interruption.

RETURNS

N/A

SEE ALSO

usrLib, **symLib**, ***symEach*( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

*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*

ln97xEndLoad( )

NAME ***ln97xEndLoad*( )** – initialize the driver and device

SYNOPSIS

```
END_OBJ * ln97xEndLoad
    (
    char * initString /* string to be parse by the driver */
    )
```

DESCRIPTION This routine initializes the driver and the device to the operational state. All of the device-specific parameters are passed in *initString*, which expects a string of the following format:

unit:*devMemAddr*:*devIoAddr*:*pciMemBase:<vecnum*:*intLvl*:*memAdrs*:*memSize*:*memWidth*:*csr3b*:*offset*:*flags*

This routine can be called in two modes. If it is called with an empty but allocated string, it places the name of this device (that is, "lnPci") into the *initString* and returns 0.

If the string is allocated and not empty, the routine attempts to load the driver using the values specified in the string.

RETURNS An END object pointer, or NULL on error, or 0 and the name of the device if the *initString* was NULL.

SEE ALSO **ln97xEnd**

ln97xInitParse( )

NAME ***ln97xInitParse*( )** – parse the initialization string

SYNOPSIS

```
STATUS ln97xInitParse
    (
    LN_97X_DRV_CTRL * pDrvCtrl,  /* pointer to the control structure */
    char *            initString /* initialization string */
    )
```

DESCRIPTION

Parse the input string. This routine is called from ***ln97xEndLoad*( )** which intializes some values in the driver control structure with the values passed in the intialization string.

The initialization string format is:
unit:*devMemAddr*:*devIoAddr*:*pciMemBase:<vecNum*:*intLvl*:*memAdrs*:*memSize*:*memWidth*:*csr3b*:*offset*:*flags*

unit
: Device unit number, a small integer.

devMemAddr
: Device register base memory address

devIoAddr
: Device register base IO address

pciMemBase
: Base address of PCI memory space

vecNum
: Interrupt vector number.

intLvl
: Interrupt level.

memAdrs
: Memory pool address or NONE.

memSize
: Memory pool size or zero.

memWidth
: Memory system size, 1, 2, or 4 bytes (optional).

CSR3
: Value of CSR3 (for endian-ness mainly)

offset
: Offset of starting of data in the device buffers.

flags
: Device specific flags, for future use.

RETURNS

OK, or ERROR if any arguments are invalid.

SEE ALSO

ln97xEnd

ln7990EndLoad( )

NAME
ln7990EndLoad( ) – initialize the driver and device

SYNOPSIS
```
END_OBJ* ln7990EndLoad
    (
    char* initString /* string to be parse by the driver */
    )
```

DESCRIPTION
This routine initializes the driver and the device to the operational state. All of the device-specific parameters are passed in *initString*, which expects a string of the following format:

unit:*CSR_reg_addr*:*RAP_reg_addr*:*int_vector*:*int_level*:*shmem_addr*: *shmem_size*:*shmem_width*

This routine can be called in two modes. If it is called with an empty but allocated string, it places the name of this device (that is, "ln") into the *initString* and returns 0.

If the string is allocated and not empty, the routine attempts to load the driver using the values specified in the string.

RETURNS
An END object pointer, or NULL on error, or 0 and the name of the device if the *initString* was NULL.

SEE ALSO
ln7990End

ln7990InitMem( )

NAME
ln7990InitMem( ) – initialize memory for Lance chip

SYNOPSIS
```
STATUS ln7990InitMem
    (
    LN7990END_DEVICE * pDrvCtrl /* device to be initialized */
    )
```

DESCRIPTION
Using data in the control structure, setup and initialize the memory areas needed. If the memory address is not already specified, then allocate cache safe memory.

RETURNS
OK or ERROR.

SEE ALSO
ln7990End

ln7990InitParse( )

NAME

***ln7990InitParse*( )** – parse the initialization string

SYNOPSIS

```
STATUS ln7990InitParse
    (
    LN7990END_DEVICE * pDrvCtrl,
    char *             initString
    )
```

DESCRIPTION

Parse the input string. Fill in values in the driver control structure. The initialization string format is: *unit*:*csrAdr*:*rapAdr*:*vecnum*:*intLvl*:*memAdrs*:*memSize*:*memWidth*: *offset*:*csr3B*

unit
: Device unit number, a small integer.

csrAdr
: Address of CSR0 register.

rapAdr
: Address of RAP register.

vecNum
: Interrupt vector number (used with ***sysIntConnect*( )**).

intLvl
: Interrupt level.

memAdrs
: Memory pool address or NONE.

memSize
: Memory pool size or zero.

memWidth
: Memory system size, 1, 2, or 4 bytes (optional).

offset
: Memory offset for alignment.

csr3B
: CSR register 3B control value, normally 0x4 or 0x7.

RETURNS

OK, or ERROR if any arguments are invalid.

SEE ALSO

ln7990End

lnattach( )

NAME

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

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

*lnPciattach***( )**

NAME

*lnPciattach***( )** – publish the **lnPci** network interface and initialize the driver and device

SYNOPSIS

```
STATUS lnPciattach
    (
    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,    /* used if memory pool is NOT malloc()'d */
    int    memWidth,   /* byte-width of data (-1 = any width) */
    ULONG  pciMemBase, /* memory base as seen from PCI */
    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_lnPci

2

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, */
                    /* load addr via this ptr */
    char * *ppData, /* load data segment at addr pointed to by this */
                    /* return load addr via this ptr */
    char * *ppBss   /* load BSS segment at addr pointed to by this ptr, */
                    /* 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.

This routine is not reentrant. For a reentrant version, see ***localtime_r*( )**.

INCLUDE FILES **time.h**

RETURNS A pointer to a **tm** structure containing the local broken-down time.

SEE ALSO **ansiTime**

localtime_r( )

NAME *localtime_r*( ) – convert calendar time into broken-down time (POSIX)

SYNOPSIS
```
int localtime_r
    (
    const time_t * timer,     /* calendar time in seconds */
    struct tm *    timeBuffer /* buffer for the broken-down time */
    )
```

DESCRIPTION This routine converts the calendar time pointed to by *timer* into broken-down time, expressed as local time. The broken-down time is stored in *timeBuffer*.

This routine is the POSIX re-entrant version of ***localtime*( )**.

INCLUDE FILES **time.h**

RETURNS OK.

SEE ALSO **ansiTime**

log( )

NAME

log( ) – compute a natural logarithm (ANSI)

SYNOPSIS

```
double log
    (
    double x /* value to compute the natural logarithm of */
    )
```

DESCRIPTION

This routine returns the natural logarithm of *x*in double precision (IEEE double, 53 bits).

A domain error occurs if the argument is negative. A range error may occur if the argument is zero.

INCLUDE FILES

math.h

RETURNS

The double-precision natural logarithm of *x*.

Special cases:
  If $x < 0$ (including -INF), it returns NaN with signal.
  If *x* is +INF, it returns *x* with no signal.
  If *x* is 0, it returns -INF with signal.
  If *x* is NaN it returns *x* with no signal.

SEE ALSO

ansiMath, **mathALib**

log2( )

NAME

log2( ) – compute a base-2 logarithm

SYNOPSIS

```
double log2
    (
    double x /* value to compute the base-two logarithm of */
    )
```

DESCRIPTION

This routine returns the base-2 logarithm of *x* in double precision.

INCLUDE FILES

math.h

RETURNS

The double-precision base-2 logarithm of *x*.

SEE ALSO

mathALib

log2f( )

NAME

log2f( ) – compute a base-2 logarithm

SYNOPSIS

```
float log2f
    (
    float x /* value to compute the base-2 logarithm of */
    )
```

DESCRIPTION

This routine returns the base-2 logarithm of *x* in single precision.

INCLUDE FILES

math.h

RETURNS

The single-precision base-2 logarithm of *x*.

SEE ALSO

mathALib

log10( )

NAME

log10( ) – compute a base-10 logarithm (ANSI)

SYNOPSIS

```
double log10
    (
    double x /* value to compute the base-10 logarithm of */
    )
```

DESCRIPTION

This routine returns the base 10 logarithm of *x* in double precision (IEEE double, 53 bits).

A domain error occurs if the argument is negative. A range error may if the argument is zero.

INCLUDE FILES

math.h

RETURNS

The double-precision base-10 logarithm of *x*.

Special cases:
If $x < 0$, ***log10*( )** returns NaN with signal.
if *x* is +INF, it returns *x* with no signal.
if *x* is 0, it returns -INF with signal.
if *x* is NaN it returns *x* with no signal.

SEE ALSO

ansiMath, **mathALib**

log10f( )

NAME *log10f***( )** – compute a base-10 logarithm (ANSI)

SYNOPSIS

```
float log10f
    (
    float x /* value to compute the base-10 logarithm of */
    )
```

DESCRIPTION This routine returns the base-10 logarithm of *x* in single precision.

INCLUDE FILES **math.h**

RETURNS The single-precision base-10 logarithm of *x*.

SEE ALSO **mathALib**

logf( )

NAME *logf***( )** – compute a natural logarithm (ANSI)

SYNOPSIS

```
float logf
    (
    float x /* value to compute the natural logarithm of */
    )
```

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

INCLUDE FILES **math.h**

RETURNS The single-precision natural logarithm of *x*.

SEE ALSO **mathALib**

logFdAdd( )

NAME

logFdAdd( ) – add a logging file descriptor

SYNOPSIS

```
STATUS logFdAdd
    (
    int fd /* file descriptor for additional logging device */
    )
```

DESCRIPTION

This routine adds to the log file descriptor list another file descriptor *fd* to which messages will be logged. The file descriptor must be a valid open file descriptor.

RETURNS

OK, or ERROR if the allowable number of additional logging file descriptors (5) is exceeded.

SEE ALSO

logLib, *logFdDelete*( )

logFdDelete( )

NAME

logFdDelete( ) – delete a logging file descriptor

SYNOPSIS

```
STATUS logFdDelete
    (
    int fd /* file descriptor to stop using as logging device */
    )
```

DESCRIPTION

This routine removes from the log file descriptor list a logging file descriptor added by ***logFdAdd*( )**. The file descriptor is not closed; but is no longer used by the logging facilities.

RETURNS

OK, or ERROR if the file descriptor was not added with ***logFdAdd*( )**.

SEE ALSO

logLib, *logFdAdd*( )

logFdSet( )

NAME

logFdSet( ) – set the primary logging file descriptor

SYNOPSIS

```
void logFdSet
    (
    int fd /* file descriptor to use as logging device */
    )
```

DESCRIPTION

This routine changes the file descriptor where messages from ***logMsg*( )** are written, allowing the log device to be changed from the default specified by ***logInit*( )**. It first removes the old file descriptor (if one had been previously set) from the log file descriptor list, then adds the new *fd*.

The old logging file descriptor is not closed or affected by this call; it is simply no longer used by the logging facilities.

RETURNS

N/A

SEE ALSO

logLib, *logFdAdd*( ), *logFdDelete*( )

loginDefaultEncrypt( )

NAME

loginDefaultEncrypt( ) – default password encryption routine

SYNOPSIS

```
STATUS loginDefaultEncrypt
    (
    char * in, /* input string */
    char * out /* encrypted string */
    )
```

DESCRIPTION

This routine provides default encryption for login passwords. It employs a simple encryption algorithm. It takes as arguments a string *in* and a pointer to a buffer *out*. The encrypted string is then stored in the buffer.

The input strings must be at least 8 characters and no more than 40 characters.

If a more sophisticated encryption algorithm is needed, this routine can be replaced, as long as the new encryption routine retains the same declarations as the default routine. The 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 the configuration macro **INCLUDE_SECURITY** is defined, it is called by ***usrRoot*( )** in **usrConfig.c**, before any other routines in this module.

RETURNS N/A

SEE ALSO **loginLib**

logInit( )

NAME *logInit*( ) – initialize message logging library

SYNOPSIS

```
STATUS logInit
    (
    int fd,     /* file descriptor to use as logging device */
    int maxMsgs /* max. number of messages allowed in log queue */
    )
```

DESCRIPTION This routine specifies the file descriptor to be used as the logging device and the number of messages that can be in the logging queue. If more than *maxMsgs* are in the queue, they will be discarded. A message is printed to indicate lost messages.

This routine spawns ***logTask*( )**, the task-level portion of error logging.

This routine must be called before any other routine in **logLib**. This is done by the root task, ***usrRoot*( )**, in **usrConfig.c**.

RETURNS OK, or ERROR if a message queue could not be created or ***logTask*( )** could not be spawned.

SEE ALSO **logLib**

loginPrompt( )

NAME *loginPrompt*( ) – display a login prompt and validate a user entry

SYNOPSIS

```
STATUS loginPrompt
    (
    char * userName /* user name, ask if NULL or not provided */
    )
```

DESCRIPTION This routine displays a login prompt and validates a user entry. If both user name and password match with an entry in the login table, the user is then given access to the VxWorks system. Otherwise, it prompts the user again.

All control characters are disabled during authentication except CTRL-D, which will terminate the remote login session.

RETURNS OK if the name and password are valid, or ERROR if there is an EOF or the routine times out.

SEE ALSO **loginLib**

loginStringSet( )

NAME *loginStringSet*( ) – change the login string

SYNOPSIS

```
void loginStringSet
    (
    char * newString /* string to become new login prompt */
    )
```

DESCRIPTION This routine changes the login prompt string to *newString*. The maximum string length is 80 characters.

RETURNS N/A

SEE ALSO **loginLib**

loginUserAdd( )

NAME

loginUserAdd( ) – add a user to the login table

SYNOPSIS

```
STATUS loginUserAdd
    (
    char name[MAX_LOGIN_NAME_LEN+1], /* user name */
    char passwd[80]                  /* user password */
    )
```

DESCRIPTION

This routine adds a user name and password entry to the login table. Note that what is saved in the login table is the user name and the address of *passwd*, not the actual password.

The length of user names should not exceed **MAX_LOGIN_NAME_LEN**, while the length of passwords depends on the encryption routine used. For the default encryption routine, passwords should be at least 8 characters long and no more than 40 characters.

The procedure for adding a new user to login table is as follows:

(1) Generate the encrypted password by invoking vxencrypt in **host/*hostOs*/bin**.

(2) Add a user by invoking ***loginUserAdd*( )** in the VxWorks shell with the user name and the encrypted password.

The password of a user can be changed by first deleting the user entry, then adding the user entry again with the new encrypted password.

EXAMPLE

```
-> loginUserAdd "peter", "RRdRd9Qbyz"
value = 0 = 0x0
-> loginUserAdd "robin", "bSzyydqbSb"
value = 0 = 0x0
-> loginUserShow
  User Name
  =========
  peter
  robin
value = 0 = 0x0
->
```

RETURNS

OK, or ERROR if the user name has already been entered.

SEE ALSO

loginLib, **vxencrypt**

loginUserDelete( )

NAME *loginUserDelete*( ) – delete a user entry from the login table

SYNOPSIS

```
STATUS loginUserDelete
    (
    char * name,  /* user name */
    char * passwd /* user password */
    )
```

DESCRIPTION This routine deletes an entry in the login table. Both the user name and password must be specified to remove an entry from the login table.

RETURNS OK, or ERROR if the specified user or password is incorrect.

SEE ALSO **loginLib**

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( )

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*, **windsh**, *Tornado User's Guide: Shell*

2

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 *pMib2SysDescrpMib2SysContact*, *pMib2SysLocation*, and *pMib2SysObjectId* are passed directly to ***m2SysInit*( )**; *pTrapRtn* and *pTrapArg* are passed directly to ***m2IfInit*( )**; and *maxRouteTableSize* is passed to ***m2IpInit*( )**.

RETURNS

OK if successful, otherwise ERROR.

SEE ALSO

m2Lib, *m2SysInit*( ), *m2TcpInit*( ), *m2UdpInit*( ), *m2IcmpInit*( ), *m2IfInit*( ), *m2IpInit*( )

m2IpAddrTblEntryGet( )

NAME

m2IpAddrTblEntryGet( ) – get an IP MIB-II address entry

SYNOPSIS

```
STATUS m2IpAddrTblEntryGet
    (
    int             search,         /* M2_EXACT_VALUE or M2_NEXT_VALUE */
    M2_IPADDRTBL * pIpAddrTblEntry /* ptr to requested IP address entry */
    )
```

DESCRIPTION

This routine traverses the IP address table and does an **M2_EXACT_VALUE** or a **M2_NEXT_VALUE** search based on the *search* parameter. The calling routine is responsible for supplying a valid MIB-II entry index in the input structure *pIpAddrTblEntry*. The index is the local IP address. The first entry in the table is retrieved by doing a NEXT search with the index field set to zero.

RETURNS
OK, ERROR if the input parameter is not specified, or a match is not found.

ERRNO
S_m2Lib_INVALID_PARAMETER
S_m2Lib_ENTRY_NOT_FOUND

SEE ALSO
m2IpLib, **m2Lib**, *m2IpInit***( )**, *m2IpGroupInfoGet***( )**, *m2IpGroupInfoSet***( )**, *m2IpAtransTblEntrySet***( )**, *m2IpRouteTblEntryGet***( )**, *m2IpRouteTblEntrySet***( )**, *m2IpDelete***( )**

m2IpAtransTblEntryGet( )

NAME
*m2IpAtransTblEntryGet***( )** – get a MIB-II ARP table entry

SYNOPSIS
```
STATUS m2IpAtransTblEntryGet
    (
    int               search,       /* M2_EXACT_VALUE or M2_NEXT_VALUE */
    M2_IPATRANSTBL * pReqIpAtEntry /* ptr to the requested ARP entry */
    )
```

DESCRIPTION
This routine traverses the ARP table and does an **M2_EXACT_VALUE** or a **M2_NEXT_VALUE** search based on the *search* parameter. The calling routine is responsible for supplying a valid MIB-II entry index in the input structure *pReqIpatEntry*. The index is made up of the network interface index and the IP address corresponding to the physical address. The first entry in the table is retrieved by doing a NEXT search with the index fields set to zero.

RETURNS
OK, ERROR if the input parameter is not specified, or a match is not found.

ERRNO
S_m2Lib_INVALID_PARAMETER
S_m2Lib_ENTRY_NOT_FOUND

SEE ALSO
m2IpLib, **m2Lib**, *m2IpInit***( )**, *m2IpGroupInfoGet***( )**, *m2IpGroupInfoSet***( )**, *m2IpAtransTblEntrySet***( )**, *m2IpRouteTblEntryGet***( )**, *m2IpRouteTblEntrySet***( )**, *m2IpDelete***( )**

2

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.

2

ERRNO S_m2Lib_INVALID_PARAMETER
S_m2Lib_INVALID_VAR_TO_SET

SEE ALSO **m2IpLib**, *m2IpInit*( ), *m2IpGroupInfoGet*( ), *m2IpAddrTblEntryGet*( ), *m2IpAtransTblEntrySet*( ), *m2IpRouteTblEntryGet*( ), *m2IpRouteTblEntrySet*( ), *m2IpDelete*( )

m2IpInit( )

NAME *m2IpInit*( ) – initialize MIB-II IP-group access

SYNOPSIS

```
STATUS m2IpInit
    (
    int maxRouteTableSize /* max size of routing table */
    )
```

DESCRIPTION This routine allocates the resources needed to allow access to the MIB-II IP variables. This routine must be called before any IP variables can be accessed. The parameter *maxRouteTableSize* is used to increase the default size of the MIB-II route table cache.

RETURNS OK, or ERROR if the route table or the route semaphore cannot be allocated.

ERRNO S_m2Lib_CANT_CREATE_ROUTE_SEM

SEE ALSO **m2IpLib**, *m2IpGroupInfoGet*( ), *m2IpGroupInfoSet*( ), *m2IpAddrTblEntryGet*( ), *m2IpAtransTblEntrySet*( ), *m2IpRouteTblEntryGet*( ), *m2IpRouteTblEntrySet*( ), *m2IpDelete*( )

m2IpRouteTblEntryGet( )

NAME *m2IpRouteTblEntryGet*( ) – get a MIB-2 routing table entry

SYNOPSIS

```
STATUS m2IpRouteTblEntryGet
    (
    int             search,          /* M2_EXACT_VALUE or M2_NEXT_VALUE */
    M2_IPROUTETBL * pIpRouteTblEntry /* route table entry */
    )
```

DESCRIPTION

This routine retrieves MIB-II information about an entry in the network routing table and returns it in the caller-supplied structure *pIpRouteTblEntry*.

The routine compares routing table entries to the address specified by the **ipRouteDest** member of the *pIpRouteTblEntry* structure, and retrieves an entry chosen by the *search* type (**M2_EXACT_VALUE** or **M2_NEXT_VALUE**, as described in the manual entry for **m2Lib**).

RETURNS

OK if successful, otherwise ERROR.

ERRNO

S_m2Lib_INVALID_PARAMETER
S_m2Lib_ENTRY_NOT_FOUND

SEE ALSO

m2IpLib, **m2Lib**, ***m2IpInit*( )**, ***m2IpGroupInfoGet*( )**, ***m2IpGroupInfoSet*( )**, ***m2IpAddrTblEntryGet*( )**, ***m2IpRouteTblEntryGet*( )**, ***m2IpRouteTblEntrySet*( )**, ***m2IpDelete*( )**

m2IpRouteTblEntrySet( )

NAME

***m2IpRouteTblEntrySet*( )** – set a MIB-II routing table entry

SYNOPSIS

```
STATUS m2IpRouteTblEntrySet
    (
    int              varToSet,        /* variable to set */
    M2_IPROUTETBL * pIpRouteTblEntry /* route table entry */
    )
```

DESCRIPTION

This routine adds, changes, or deletes a network routing table entry. The table entry to be modified is specified by the **ipRouteDest** and **ipRouteNextHop** members of the *pIpRouteTblEntry* structure.

The *varToSet* parameter is a bit-field mask that specifies which values in the route table entry are to be set.

If *varToSet* has the **M2_IP_ROUTE_TYPE** bit set and **ipRouteType** has the value of **M2_ROUTE_TYPE_INVALID**, then the 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*( )**, ***m2IpRouteTblEntrySet*( )**, ***m2IpDelete*( )**

2

m2OspfAreaEntryGet( )

NAME *m2OspfAreaEntryGet*( ) – get an entry from the OSPF area table (OSPF Opt.)

SYNOPSIS

```
STATUS m2OspfAreaEntryGet
    (
    int                  searchType, /* M2_EXACT_VALUE or M2_NEXT_VALUE */
    M2_OSPF_AREA_ENTRY * pInfo       /* ptr to area entry */
    )
```

DESCRIPTION The structure pointed to by *pInfo* is filled with the contents of the area entry specified by **pInfo>ospfAreaId** and *searchType*.

RETURNS OK or ERROR.

SEE ALSO **ospfLib**

m2OspfAreaEntrySet( )

NAME *m2OspfAreaEntrySet*( ) – set values in an OSPF area entry (OSPF Opt.)

SYNOPSIS

```
STATUS m2OspfAreaEntrySet
    (
    int                  varsToSet, /* flags specifying vars to set */
    M2_OSPF_AREA_ENTRY * pInfo      /* ptr to area entry */
    )
```

DESCRIPTION The area entry specified by pInfo>ospfAreaId will be updated with the values provided by *pInfo*. The *varsToSet* parameter indicates the fields to set and is a bitwise or of one or more of **M2_OSPF_AREA_ID**, **M2_OSPF_AUTH_TYPE**, and **M2_OSPF_IMPORT_AS_EXTERN**.

Note that the backbone area (0.0.0.0) is always present and does not need to be created explicitly. It is an error to use the **M2_OSPF_AREA_ID** or **M2_OSPF_IMPORT_AS_EXTERN** flags with an area ID of 0.0.0.0.

RETURNS OK or ERROR.

SEE ALSO **ospfLib**

m2OspfAreaRangeEntryGet( )

NAME *m2OspfAreaRangeEntryGet*( ) – get an OSPF area range entry (OSPF Opt.)

SYNOPSIS

```
STATUS m2OspfAreaRangeEntryGet
    (
    int                          searchType, /* M2_EXACT_VALUE */
                                             /* or M2_NEXT_VALUE */
    M2_OSPF_AREA_RANGE_ENTRY * pInfo       /* ptr to area arange entry */
    )
```

DESCRIPTION The structure pointed to by *pInfo* is filled in with the OSPF area range entry specified by **pInfo>ospfAreaRangeAreaID**, **pInfo>ospfAreaRangeNet**, and *searchType*.

RETURNS OK or ERROR.

SEE ALSO **ospfLib**

m2OspfAreaRangeEntrySet( )

NAME *m2OspfAreaRangeEntrySet*( ) – set values in an OSPF area range entry (OSPF Opt.)

SYNOPSIS

```
STATUS m2OspfAreaRangeEntrySet
    (
    int                          varsToSet, /* flags specifying vars to set */
    M2_OSPF_AREA_RANGE_ENTRY * pInfo      /* ptr to area range entry */
    )
```

DESCRIPTION The OSPF area range entry specified by **pInfo>ospfAreaRangeAreaID** and **pInfo>ospfAreaRangeNet** is updated with the values provided in *pInfo*. The *varsToSet* parameter specifies the fields to set and is a bitwise or of one or more of **M2_OSPF_AREA_RANGE_AREA_ID**, **M2_OSPF_AREA_RANGE_NET**, **M2_OSPF_AREA_RANGE_MASK**, and **M2_OSPF_AREA_RANGE_STATUS**.

RETURNS OK or ERROR.

SEE ALSO **ospfLib**

2

m2OspfGeneralGroupGet( )

NAME
m2OspfGeneralGroupGet( ) – get values of OSPF general group objects (OSPF Opt.)

SYNOPSIS
```
STATUS m2OspfGeneralGroupGet
    (
    M2_OSPF_GENERAL_GROUP * pInfo /* pointer to general group struct */
    )
```

DESCRIPTION
This routine fills in the structure pointed to by *pInfo* with the MIB-II values for the OSPF general group.

RETURNS
OK, or ERROR if the get request fails.

SEE ALSO
ospfLib

m2OspfGeneralGroupSet( )

NAME
m2OspfGeneralGroupSet( ) – set values of OSPF general group objects (OSPF Opt.)

SYNOPSIS
```
STATUS m2OspfGeneralGroupSet
    (
    int                     varsToSet, /* flags specifying vars to set */
    M2_OSPF_GENERAL_GROUP * pInfo      /* ptr to general group structure */
    )
```

DESCRIPTION
This routine sets the values of the OSPF general group objects. The variables to set are specified by a bitwise or of one or more of the flags **M2_OSPF_ROUTER_ID**, **M2_OSPF_ADMIN_STAT**, **M2_OSPF_AS_BDR_RTR_STATUS**, and **M2_OSPF_TOS_SUPPORT**, in the *varsToSet* parameter.

RETURNS
OK or ERROR.

SEE ALSO
ospfLib

m2OspfHostEntryGet( )

NAME

m2OspfHostEntryGet( ) – get an OSPF host entry (OSPF Opt.)

SYNOPSIS

```
STATUS m2OspfHostEntryGet
    (
    int                    searchType, /* M2_EXACT_VALUE or M2_NEXT_VALUE */
    M2_OSPF_HOST_ENTRY * pInfo       /* ptr to host entry */
    )
```

DESCRIPTION

The structure pointed to by *pInfo* is filled in with the entry specified by **pInfo>ospfHostIpAddress**, **pInfo>ospfHostTOS**, and *searchType*.

RETURNS

OK or ERROR.

SEE ALSO

ospfLib

m2OspfHostEntrySet( )

NAME

m2OspfHostEntrySet( ) – set values in an OSPF host entry (OSPF Opt.)

SYNOPSIS

```
STATUS m2OspfHostEntrySet
    (
    int                    varsToSet, /* flags specifying vars to set */
    M2_OSPF_HOST_ENTRY * pInfo      /* ptr to host entry */
    )
```

DESCRIPTION

The OSPF host entry specified by **pInfo>ospfHostIpAddress** and **pInfo>ospfHostTOS** is updated with the values provided in *pInfo*. The *varsToSet* parameter indicates the fields to be set and is a bitwise or of one or more of **M2_OSPF_HOST_IP_ADDRESS**, **M2_OSPF_HOST_TOS**, **M2_OSPF_HOST_METRIC**, and **M2_OSPF_HOST_STATUS**.

RETURNS

OK or ERROR.

SEE ALSO

ospfLib

2

m2OspfIfEntryGet( )

NAME *m2OspfIfEntryGet*( ) – get an OSPF interface entry (OSPF Opt.)

SYNOPSIS

```
STATUS m2OspfIfEntryGet
    (
    int                searchType, /* M2_EXACT_VALUE or M2_NEXT_VALUE */
    M2_OSPF_IF_ENTRY * pInfo       /* ptr ot interface entry */
    )
```

DESCRIPTION The structure pointed to by *pInfo* is filled in with the entry specified by **pInfo>ospfIfIpAddress**, **pInfo>ospfAddressLessIf**, and *searchType*.

RETURNS OK or ERROR.

SEE ALSO **ospfLib**

m2OspfIfEntrySet( )

NAME *m2OspfIfEntrySet*( ) – set values in an OSPF interface entry (OSPF Opt.)

SYNOPSIS

```
STATUS m2OspfIfEntrySet
    (
    int                varsToSet, /* flags specifying vars to set */
    M2_OSPF_IF_ENTRY * pInfo      /* ptr to interface entry */
    )
```

DESCRIPTION This routine updates **pInfo>ospfAddressLessIf** with the contents of *pInfo*. The *varsToSet* parameter indicates the fields to set and is a bitwise or of one or more of:

M2_OSPF_IF_AREA_ID
M2_OSPF_IF_ADMIN_STAT
M2_OSPF_IF_RTR_PRIORITY
M2_OSPF_IF_TRANSIT_DELAY
M2_OSPF_IF_RETRANS_INTERVAL
M2_OSPF_IF_HELLO_INTERVAL
M2_OSPF_IF_RTR_DEAD_INTERVAL
M2_OSPF_IF_POLL_INTERVAL
M2_OSPF_IF_AUTH_KEY

RETURNS OK or ERROR.

SEE ALSO **ospfLib**

m2OspfIfMetricEntryGet( )

NAME *m2OspfIfMetricEntryGet*( ) – get an OSPF interface metric entry (OSPF Opt.)

SYNOPSIS

```
STATUS m2OspfIfMetricEntryGet
    (
    int                         searchType, /* M2_EXACT_VALUE */
                                            /* or M2_NEXT_VALUE */
    M2_OSPF_IF_METRIC_ENTRY * pInfo         /* ptr to interface metric entry */
    )
```

DESCRIPTION The structure pointed to by *pInfo* is filled in with the entry specified by **pInfo>ospfIfMetricIpAddress**, **pInfo>ospfIfMetricAddressLessIf**, **pInfo>ospfIfMetricTOS**, and *searchType*.

RETURNS OK or ERROR.

SEE ALSO **ospfLib**

m2OspfIfMetricEntrySet( )

NAME *m2OspfIfMetricEntrySet*( ) – set OSPF interface metric entry values (OSPF Opt.)

SYNOPSIS

```
STATUS m2OspfIfMetricEntrySet
    (
    int                         varsToSet, /* flags specifying vars to set */
    M2_OSPF_IF_METRIC_ENTRY * pInfo        /* ptr to interface metric entry */
    )
```

DESCRIPTION The fields of the OSPF interface metric entry specified by **pInfo>ospfIfMetricIpAddress**, **pInfo>ospfIfMetricAddress**, and **pInfo>ospfIfMetricTOS** is updated with the contents of *pInfo*. The *varsToSet* parameter indicates the fields to set and is a bitwise or of one or more of **M2_OSPF_IF_METRIC_METRIC** or **M2_OSPF_IF_METRIC_STATUS**.

RETURNS OK or ERROR.

SEE ALSO **ospfLib**

m2OspfLsdbEntryGet( )

NAME *m2OspfLsdbEntryGet*( ) – get an OSPF link state database entry (OSPF Opt.)

SYNOPSIS

```
STATUS m2OspfLsdbEntryGet
    (
    int                  searchType, /* M2_EXACT_VALUE or M2_NEXT_VALUE */
    M2_OSPF_LSDB_ENTRY * pInfo       /* link state database entry */
    )
```

DESCRIPTION The structure pointed to by *pInfo* is filled in with the entry specified by **pInfo>ospfLsdbAreaId**, **pInfo>ospfLsdbType**, and *searchType*.

RETURNS OK or ERROR.

SEE ALSO **ospfLib**

m2OspfNbrEntryGet( )

NAME *m2OspfNbrEntryGet*( ) – get an OSPF neighbor entry (OSPF Opt.)

SYNOPSIS

```
STATUS m2OspfNbrEntryGet
    (
    int                 searchType, /* M2_EXACT_VALUE or M2_NEXT_VALUE */
    M2_OSPF_NBR_ENTRY * pInfo       /* ptr to neighbor entry */
    )
```

DESCRIPTION The structure pointed to by pInfo is filled in with the contents of the OSPF neighbor entry specified by **pInfo>ospfNbrIpAddr**, **pInfo>ospfNbrAddressLessIndex** and *searchType*.

RETURNS OK or ERROR.

SEE ALSO **ospfLib**

m2OspfNbrEntrySet( )

NAME

m2OspfNbrEntrySet( ) – set values in an OSPF neighbor entry (OSPF Opt.)

SYNOPSIS

```
STATUS m2OspfNbrEntrySet
    (
    int                 varsToSet, /* flags specifying vars to set */
    M2_OSPF_NBR_ENTRY * pInfo      /* ptr to neighbor entry */
    )
```

DESCRIPTION

The OSPF neighbor entry specified by **pInfo>ospfNbrIpAddr** and **pInfo>ospfNbrAddressLessIndex** is updated with the contents of *pInfo*. The *varsTosSet* parameter indicates the fields to set, which can be **M2_OSPF_NBMA_NBR_STATUS**.

RETURNS

OK or ERROR.

SEE ALSO

ospfLib

m2OspfStubAreaEntryGet( )

NAME

m2OspfStubAreaEntryGet( ) – get an OSPF stub area entry (OSPF Opt.)

SYNOPSIS

```
STATUS m2OspfStubAreaEntryGet
    (
    int                       searchType, /* M2_EXACT_VALUE */
                                          /* or M2_NEXT_VALUE */
    M2_OSPF_STUB_AREA_ENTRY * pInfo       /* ptr to stub area entry */
    )
```

DESCRIPTION

The structure pointed to by *pInfo* is filled with the contents of the stub area entry specified by **pInfo>ospfStubAreaID**, **pInfo>ospfStubTOS**and *searchType*.

RETURNS

OK or ERROR.

SEE ALSO

ospfLib

m2OspfStubAreaEntrySet( )

NAME *m2OspfStubAreaEntrySet*( ) – set values in an OSPF stub area entry (OSPF Opt.)

SYNOPSIS

```
STATUS m2OspfStubAreaEntrySet
    (
    int                       varsToSet, /* flags specifying vars to set */
    M2_OSPF_STUB_AREA_ENTRY * pInfo      /* ptr to stub area entry */
    )
```

DESCRIPTION The stub area entry specified by **pInfo>ospfStubAreaID** and **pInfo>ospfStubTOS** is updated with the values provided in *pInfo*. The *varsToSet* parameter indicates the fields to be modified and is a bitwise or of one or more of **M2_OSPF_STUB_AREA_ID**, **M2_OSPF_STUB_TOS**, **M2_OSPF_STUB_METRIC**, and **M2_OSPF_STUB_STATUS**.

RETURNS OK or ERROR.

SEE ALSO **ospfLib**

m2OspfVirtIfEntryGet( )

NAME *m2OspfVirtIfEntryGet*( ) – get an OSPF virtual interface entry (OSPF Opt.)

SYNOPSIS

```
STATUS m2OspfVirtIfEntryGet
    (
    int                     searchType, /* M2_EXACT_VALUE or M2_NEXT_VALUE */
    M2_OSPF_VIRT_IF_ENTRY * pInfo       /* ptr to virtual interface entry */
    )
```

DESCRIPTION The structure pointed to by *pInfo* is filled in with the contents of the OSPF virtual interface entry specified by **pInfo>ospfVirtIfAreaID**, **pInfo>ospfVirtIfNeighbor** and *searchType*.

RETURNS OK or ERROR.

SEE ALSO **ospfLib**

m2OspfVirtIfEntrySet( )

NAME *m2OspfVirtIfEntrySet*( ) – set OSPF virtual interface entry values (OSPF Opt.)

SYNOPSIS

```
STATUS m2OspfVirtIfEntrySet
    (
    int                       varsToSet, /* flags specifying vars to set */
    M2_OSPF_VIRT_IF_ENTRY * pInfo      /* ptr to virtual interface entry */
    )
```

DESCRIPTION The OSPF virtual interface entry specified by **pInfo>ospfVirtIfAreaID**and **pInfo>ospfVirtIfNeighbor** is updated with the contents of *pInfo*. The *varsToSet* parameter indicates the fields to be modified and is a bitwise or of one or more of:

M2_OSPF_VIRT_IF_AREA_ID
M2_OSPF_VIRT_IF_NEIGHBOR
M2_OSPF_VIRT_IF_TRANSIT_DELAY
M2_OSPF_VIRT_IF_HELLO_INTERVAL
M2_OSPF_VIRT_IF_RTR_DEAD_INTERVAL
M2_OSPF_VIRT_IF_STATUS
M2_OSPF_VIRT_IF_AUTH_KEY

RETURNS OK or ERROR.

SEE ALSO **ospfLib**

m2OspfVirtNbrEntryGet( )

NAME *m2OspfVirtNbrEntryGet*( ) – get an OSPF virtual neighbor entry (OSPF Opt.)

SYNOPSIS

```
STATUS m2OspfVirtNbrEntryGet
    (
    int                        searchType, /* M2_EXACT_VALUE */
                                           /* or M2_NEXT_VALUE */
    M2_OSPF_VIRT_NBR_ENTRY * pInfo       /* ptr to virtual neighbor entry */
    )
```

DESCRIPTION The structure pointed to by pInfo is filled in with the contents of the OSPF virtual neighbor entry specified by **pInfo>ospfVirtNbrArea**, **pInfo>ospfVirtNbrRtrId**, and *searchType*.

RETURNS OK or ERROR.

SEE ALSO **ospfLib**

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 a **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

2

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**)*

masterIoInit( )

NAME

masterIoInit( ) – create the IPC mechanism at the SNMP master agent

SYNOPSIS

```
STATUS masterIoInit ( void )
```

DESCRIPTION

This routine, called from ***snmpIoInit*( )**, creates the SNMP master agent side of the inter-process communication (IPC) mechanism used to carry messages between subagents and the master agent. In this implementation, ***masterIoInit*( )** creates a single message queue. The identity of this message queue is hard coded into every subagent. The subagent puts a message on this queue when it needs to send a message to the master agent.

The message queue created by ***masterIoInit*( )** is monitored by **tMonQue**. The **tMonQue** task is one of the two tasks used to implement the SNMP master agent. The purpose of **tMonQue** is to note which messages in its queue are registration requests and which are responses to queries. If the message is a subagent registration request, **tMonQue** handles the request and sends a message back to the subagent telling it whether the registration was successful or not.

If the message is a response to a query, **tMonQue** transfers the message to the message queue monitored by **tSnmpd**. The **tSnmpd** task then encodes the response in an SNMP packet and transmits the packet over a socket to the SNMP manager.

Although the shipped version of this function uses message queues as the IPC between the master agent and its subagents, the IPC mechanism is isolated to the relatively small number functions defined in **masterIoLib**. Thus, if necessary, you should have little trouble porting the code to use an IPC more suitable to your transport needs.

For example, you could use sockets instead of message queues. However, if you decide to change the IPC mechanism, you must do so both in the master agent and in its subagents. This means that you must also modify the functions defined in **saIoLib**, the library that defines the agent side of the IPC mechanism.

RETURNS OK or ERROR.

SEE ALSO **masterIoLib**

masterIoWrite( )

NAME *masterIoWrite*( ) – send the encoded buffer to the subagent

SYNOPSIS

```
STATUS masterIoWrite
    (
    EBUFFER_T * pBuf, /* reply message to be sent */
    PTR_T       saId, /* subagent address */
    INT_32_T    flg   /* type of message */
    )
```

DESCRIPTION This routine transmits the byte array at *pBuf* to the subagent at *saId*. This routine is called from a wide variety of functions in the master agent. For example, ***masterIpcSend*( )** calls this routine when it needs to query the subagent about one of the MIB variables it manages. Likewise, the ***masterIpcAyt*( )** function calls this routine when needs to check the IPC link status. Similarly, ***snmpQueMonitor*( )** calls this routine to tell the agent the results of a registration or deregistration request.

The master agent uses the value *flg* to specify the general nature of the message it is writing to the subagent, which partially determines how the subagent responds. For example, when the master agent is responding to the subagent after successfully handling its registration request, the master agent uses a *flg* value of **REG_COMPLETE**. When the master agent does an "are you there" check, it specifies a *flg* value of **IPC_AYT**. **REG_COMPLETE** and **IPC_AYT** are the only currently valid *flg* values.

RETURNS OK or ERROR.

SEE ALSO **masterIoLib**

masterIpcAyt( )

NAME

masterIpcAyt( ) – check the status of the IPC link

SYNOPSIS

```
INT_32_T masterIpcAyt
    (
    PTR_T ipchandle /* pointer to IPC handle */
    )
```

DESCRIPTION

This is an "are you there" routine. The SNMP master agent calls this routine whenever it needs to do a status check on the IPC link to the address *ipchandle*. This routine puts a null-buffer message of type **IPC_AYT** on the subagent's message queue. If the subagent replies with a message of the same type, the link is considered active.

RETURNS

0, if the link is inactive; 1, if the link is inactive

SEE ALSO

masterIoLib

masterIpcComp( )

NAME

masterIpcComp( ) – transmit a completion of transmission message

SYNOPSIS

```
void masterIpcComp
    (
    OCTET_T     opcode,   /* this specifies what needs to be done */
    EBUFFER_T * ebuf,     /* reply message to be sent */
    VBL_T *     vblist,   /* list of varbinds that the message contained */
    PTR_T       ipchandle /* subagent address */
    )
```

DESCRIPTION

If the SNMP master agent uses ***snmpMasterHandlerAsync*( )** to process a subagent's unsolicited control message (such as a registration request), it uses ***masterIpcComp*( )** to complete processing for the message. In the current implementation, this means telling the subagent the completion status of a registration or deregistration request. However, you can rewrite this function to implement a broader range of responses (such as forwarding traps to the SNMP manager).

When the master agent calls this routine, it uses *opcode* to indicate the processing status of the message. If the status indicates an error, ***masterIpcComp*( )** drops the packet. If the status indicates success, the master agent uses the *ebuf* parameter to pass in a message for the subagent at *ipchandle*. Internally, ***masterIpcComp*( )** calls ***masterIoWrite*( )** to forward the message to the specified subagent. If this message is the response to a successful registration request, it contains the group ID for the MIB variables just added to the master agent's MIB tree. The subagent needs this group ID for any deregistration request it might send later. It also uses this ID to register instances of the object just registered.

If the *opcode* is a value of 1 or greater (up to and including 127), the master agent uses the *vblist* parameter to pass in a varbind list that it extracted from the control message. In the current implementation, the ***masterIpcComp*( )** routine does nothing with the message and returns. However, you could modify ***masterIpcComp*( )** to process the message according to the value specified by *opcode*. For example, if *opcode* indicates a trap, you could forward the information at *vblist* to the SNMP manager.

Currently, **subagent.h** defines symbolic constants for opcodes 1 through 12 (with opcode 11, **SA_TRAP_REQUEST**, reserved for trap requests). You are free to use the remaining opcodes for message types specific to your implementation.

RETURNS
N/A

SEE ALSO
masterIoLib

---

masterIpcFree( )

NAME
masterIpcFree( ) – free the IPC resources allocated by the SNMP master agent

SYNOPSIS

```
void masterIpcFree
    (
    PTR_T ipchandle /* pointer to IPC handle */
    )
```

DESCRIPTION
The SNMP master agent calls this routine to free a pointer to an IPC handle. This is part of the deregistration process for an SNMP agent.

RETURNS
N/A

SEE ALSO
masterIoLib

masterIpcRcv( )

NAME

masterIpcRcv( ) – wait for a reply from the subagent

SYNOPSIS

```
INT_32_T masterIpcRcv
    (
    EBUFFER_T * pBuf,     /* buffer to be filled */
    PTR_T       ipchandle /* pointer to the IPC handle */
    )
```

DESCRIPTION

This routine waits for a response after query has been sent to the subagent. In the shipped implementation of the WindNet SNMP master agent, this function waits on a message queue that is local to the master agent. This message queue is used to facilitate communication between **tSnmpd**, the task that manages communication with the SNMP manager, and **tMonQue**, the task that manages communication between the SNMP master agent and its subagents.

In the shipped master agent code, subagents communicate with the master agent by putting messages on the message queue monitored by **tMonQue**. If the message is a control message, it is processed by ***snmpMasterHandlerWR*( )**. If the message is a query response, it is transferred to the local message queue on which ***masterIpcRcv*( )** is waiting. All of this is handled synchronously. Thus, while the master agent is waiting for a response from the subagent, it is blocked. Normally, the amount of time spent blocked is quite short and is not a problem.

However, it is an imperfect world, so it is possible that a response for a query never makes it back to the subagent. To handle this possibility, the shipped version of the WindNet SNMP master agent puts a timeout on its wait for a query response. If you should rewrite the SNMP master agent for any reason, make sure that you preserve this timeout.

RETURNS

0, if the packet was received successfully; 1, if an error or a timeout has caused the objects to be marked inactive and subsequently removed; 2, if the master agent will allow the current packet to be processed without freeing objects.

SEE ALSO

masterIoLib

masterIpcSend( )

NAME *masterIpcSend*( ) – send a message to a subagent

SYNOPSIS

```
INT_32_T masterIpcSend
    (
    EBUFFER_T * pBuf,     /* message to be sent */
    PTR_T       ipchandle /* address of subagent */
    )
```

DESCRIPTION The SNMP master agent calls when it needs to send a query in *buf* to the subagent at the *ipchandle* address. If this routine is used with ***snmpMasterHandlerAsync*( )**, you must rewrite the function according to the prototype of **IPCSEND_AS_T** (see **subagent.h**). The additional parameter *reqid* in this prototype is the request ID of the message being sent. Use *reqid* to call ***snmpMasterCleanup*( )** if the IPC layer times out.

Internally, this function calls ***masterIoWrite*( )** to put a message on the subagent's message queue. If you have rewritten ***masterIoWrite*( )** to use different IPC mechanism, such as sockets, you should take care that your rewrite of ***masterIoWrite*( )** is compatible with its use in ***masterIpcSend*( )**.

RETURNS 0, if the packet has been sent successfully; 1, if and error has been detected that caused the objects to be marked inactive and possibly removed; 2, if the processing of the current packet is allowed to continue without freeing up objects.

SEE ALSO **masterIoLib**

masterQueCleanup( )

NAME *masterQueCleanup*( ) – free resources allocated for SNMP master agent

SYNOPSIS

```
void masterQueCleanup (void)
```

DESCRIPTION This routine is called from the cleanup routine in **snmpIoLib** if the agent fails to allocate resources. This routine deletes the message queue and all other resources that have been allocated for the master agent.

RETURNS N/A

SEE ALSO **masterIoLib**

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**

mb86960EndLoad( )

NAME *mb86960EndLoad*( ) – initialize the driver and device

SYNOPSIS

```
END_OBJ * mb86960EndLoad
    (
    char * pInitString /* String to be parsed by the driver. */
    )
```

DESCRIPTION This routine initializes the driver and puts the device to an operational state. All of the device specific parameters are passed in via the initString, which expects a string of the following format:

unit:*base_addr*:*int_vector*:*int_level*

This routine can be called in two modes. If it is called with an empty but allocated string, it places the name of this device (that is, "fn") into the *initString* and returns 0.

If the string is allocated and not empty, the routine attempts to load the driver using the values specified in the string.

RETURNS An END object pointer, or NULL on error, or 0 and the name of the device if the *initString* was NULL.

SEE ALSO **mb86960End**

mb86960InitParse( )

NAME *mb86960InitParse*( ) – parse the initialization string

SYNOPSIS

```
STATUS mb86960InitParse
    (
    MB86960_END_CTRL * pDrvCtrl,   /* device pointer */
    char *             pInitString /* information string */
    )
```

DESCRIPTION Parse the input string. Fill in values in the driver control structure.

The initialization string format is:
unit:*baseAddr*:*ivec*

unit
Device unit number, a small integer. MUST always be 0.

devBaseAddr
Base address of the device register set

ivec
Interrupt vector number (used with sysIntConnect)

RETURNS OK or ERROR for invalid arguments.

SEE ALSO **mb86960End**

mb86960MemInit( )

NAME

mb86960MemInit( ) – initialize memory for the chip

SYNOPSIS

```
STATUS mb86960MemInit
    (
    MB86960_END_CTRL * pDrvCtrl /* device to be initialized */
    )
```

DESCRIPTION

This routine is highly specific to the device.

RETURNS

OK or ERROR.

SEE ALSO

mb86960End

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 */
    int                  scsiPriority       /* priority of task doing SCSI */
    )
```

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**

mbcAddrFilterSet( )

NAME *mbcAddrFilterSet*( ) – set the address filter for multicast addresses

SYNOPSIS

```
void mbcAddrFilterSet
    (
    MBC_DEVICE * pDrvCtrl /* device to be updated */
    )
```

DESCRIPTION This routine goes through all of the multicast addresses on the list of addresses (added with the *endAddrAdd*( ) routine) and sets the device's filter correctly.

RETURNS N/A.

SEE ALSO **mbcEnd**

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 rang> or non-cacheable memory cannot be allocated; otherwise TRUE.

SEE ALSO

if_mbc, **ifLib**, *Motorola MC68EN302 User's Manual*

mbcEndLoad( )

NAME

mbcEndLoad( ) – initialize the driver and device

SYNOPSIS

```
END_OBJ* mbcEndLoad
    (
    char * initString /* String to be parsed by the driver. */
    )
```

DESCRIPTION
This routine initializes the driver and the device to the operational state. All of the device specific parameters are passed in the initString.

The string contains the target specific parameters like this:

"unit:memAddr:ivec:txBdNum:rxBdNum:dmaParms:bufBase:offset"

RETURNS
An END object pointer or NULL on error.

SEE ALSO
mbcEnd

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

mbcMemInit( )

NAME
mbcMemInit( ) – initialize memory for the chip

SYNOPSIS
```
STATUS mbcMemInit
    (
    MBC_DEVICE * pDrvCtrl /* device to be initialized */
    )
```

DESCRIPTION
Allocates and initializes the memory pools for the mbc device.

RETURNS OK or ERROR.

SEE ALSO **mbcEnd**

mbcParse( )

NAME *mbcParse*( ) – parse the init string

SYNOPSIS

```
STATUS mbcParse
    (
    MBC_DEVICE * pDrvCtrl,  /* device pointer */
    char *       initString /* information string */
    )
```

DESCRIPTION Parse the input string. Fill in values in the driver control structure.

The initialization string format is:
unit:*memAddr*:*ivec*:*txBdNum*:*rxBdNum*:*dmaParms*:*bufBase*:*offset*

unit
: Device unit number, a small integer.

memAddr
: ethernet module base address.

ivec
: Interrupt vector number (used with sysIntConnect)

txBdNum
: transmit buffer descriptor

rxBdNum
: receive buffer descriptor

dmaParms
: dma parameters

bufBase
: address of memory pool

offset
: packet data offset

RETURNS OK or ERROR for invalid arguments.

SEE ALSO **mbcEnd**

mbcStartOutput( )

NAME

mbcStartOutput( ) – output packet to network interface device

SYNOPSIS

```
#ifdef BSD43_DRIVER LOCAL void mbcStartOutput
    (
    int unit /* unit number */
    )
```

DESCRIPTION

mbcStartOutput( ) takes a packet from the network interface output queue, copies the mbuf chain into an interface buffer, and sends the packet over the interface. etherOutputHookRtns are supported.

Collision stats are collected in this routine from previously sent BDs. These BDs will not be examined until after the transmitter has cycled the ring, coming upon the BD after it has been sent. Thus, collision stat collection will be delayed a full cycle through the Tx ring.

This routine is called under several possible scenarios. Each one will be described below.

The first, and most common, is when a user task requests the transmission of data. Under BSD 4.3, this results in a call to ***mbcOutput*( )**, which in turn calls ***ether_output*( )**. The routine, ***ether_output*( )**, will make a call to ***mbcStartOutput*( )** if our interface output queue is not full, otherwise, the outgoing data is discarded. BSD 4.4 uses a slightly different model, in which the generic ***ether_output*( )** routine is called directly, followed by a call to this routine.

The second scenario is when this routine, while executing runs out of free Tx BDs, turns on transmit interrupts and exits. When the next BD is transmitted, an interrupt occurs and the ISR does a netJobAdd of the routine which executes in the context of ***netTask*( )** and continues sending packets from the interface output queue.

The third scenario is when the device is reset, typically when the promiscuous mode is altered; which results in a call to ***mbcInit*( )**. This resets the device, does a ***netJobAdd*( )** of this routine to enable transmitting queued packets.

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, a null pointer.

SEE ALSO
ansiString

memcmp( )

NAME
memcmp( ) – compare two blocks of memory (ANSI)

SYNOPSIS

```
int memcmp
    (
    const void * s1, /* array 1 */
    const void * s2, /* array 2 */
    size_t       n   /* size of memory to compare */
    )
```

DESCRIPTION
This routine compares successive elements from two arrays of **unsigned char**, beginning at the addresses *s1* and *s2* (both of size *n*), until it finds elements that are not equal.

INCLUDE FILES
string.h

RETURNS
If all elements are equal, zero. If elements differ and the differing element from *s1* is greater than the element from *s2*, the routine returns a positive number; otherwise, it returns a negative number.

SEE ALSO
ansiString

memcpy( )

NAME *memcpy*( ) – copy memory from one location to another (ANSI)

SYNOPSIS

```
void * memcpy
    (
    void *       destination, /* destination of copy */
    const void * source,      /* source of copy */
    size_t       size         /* size of memory to copy */
    )
```

DESCRIPTION This routine copies *size* characters from the object pointed to by *source* into the object pointed to by *destination*. If copying takes place between objects that overlap, the behavior is undefined.

INCLUDE FILES **string.h**

RETURNS A pointer to *destination*.

SEE ALSO **ansiString**

memDevCreate( )

NAME *memDevCreate*( ) – create a memory device

SYNOPSIS

```
STATUS memDevCreate
    (
    char * name,  /* device name */
    char * base,  /* where to start in memory */
    int    length /* number of bytes */
    )
```

DESCRIPTION This routine creates a memory device containing a single file. Memory for the device is simply an absolute memory location beginning at *base*. The *length* parameter indicates the size of memory.

For example, to create the device "/mem/cpu0/", a device for accessing the entire memory of the local processor, the proper call would be:

```
memDevCreate ("/mem/cpu0/", 0, sysMemTop())
```

The device is created with the specified name, start location, and size.

To open a file descriptor to the memory, use ***open*( )**. Specify a pseudo-file name of the byte offset desired, or open the "raw" file at the beginning and specify a position to seek to. For example, the following call to ***open*( )** allows memory to be read starting at decimal offset 1000.

```
-> fd = open ("/mem/cpu0/1000", O_RDONLY, 0)
```

Pseudo-file name offsets are scanned with "%d".

CAVEAT

The **FIOSEEK** operation overrides the offset given via the pseudo-file name at open time.

EXAMPLE

Consider a system configured with two CPUs in the backplane and a separate dual-ported memory board, each with 1 megabyte of memory. The first CPU is mapped at VMEbus address 0x00400000 (4 Meg.), the second at bus address 0x00800000 (8 Meg.), the dual-ported memory board at 0x00c00000 (12 Meg.). Three devices can be created on each CPU as follows. On processor 0:

```
-> memDevCreate ("/mem/local/", 0, sysMemTop())
...
-> memDevCreate ("/mem/cpu1/", 0x00800000, 0x00100000)
...
-> memDevCreate ("/mem/share/", 0x00c00000, 0x00100000)
```

On processor 1:

```
-> memDevCreate ("/mem/local/", 0, sysMemTop())
...
-> memDevCreate ("/mem/cpu0/", 0x00400000, 0x00100000)
...
-> memDevCreate ("/mem/share/", 0x00c00000, 0x00100000)
```

Processor 0 has a local disk. Data or an object module needs to be passed from processor 0 to processor 1. To accomplish this, processor 0 first calls:

```
-> copy </disk1/module.o >/mem/share/0
```

Processor 1 can then be given the load command:

```
-> ld </mem/share/0
```

RETURNS

OK, or ERROR if memory is insufficient or the I/O system cannot add the device.

ERRNO

S_ioLib_NO_DRIVER

SEE ALSO

memDrv

memDevCreateDir( )

NAME *memDevCreateDir*( ) – create a memory device for multiple files

SYNOPSIS

```
STATUS memDevCreateDir
    (
    char *             name,    /* device name */
    MEM_DRV_DIRENTRY * files,   /* array of dir. entries - not copied */
    int                numFiles /* number of entries */
    )
```

DESCRIPTION This routine creates a memory device for a collection of files organised into directories. The given array of directory entry records describes a number of files, some of which may be directories, represented by their own directory entry arrays. The structure may be arbitrarily deep. This effectively allows a filesystem to be created and installed in VxWorks, for essentially read-only use. The filesystem structure can be created on the host using the memdrvbuild utility.

Note that the array supplied is not copied; a reference to it is kept. This array should not be modified after being passed to memDevCreateDir.

RETURNS OK, or ERROR if memory is insufficient or the I/O system cannot add the device.

ERRNO **S_ioLib_NO_DRIVER**

SEE ALSO **memDrv**

memDevDelete( )

NAME *memDevDelete*( ) – delete a memory device

SYNOPSIS

```
STATUS memDevDelete
    (
    char * name /* device name */
    )
```

DESCRIPTION This routine deletes a memory device containing a single file or a collection of files. The device is deleted with it own name.

For example, to delete the device created by memDevCreate ("/mem/cpu0/", 0, *sysMemTop*( )), the proper call would be:

```
memDevDelete ("/mem/cpu0/");
```

RETURNS OK, or ERROR if the device doesn't exist.

SEE ALSO **memDrv**

memDrv( )

NAME *memDrv*( ) – install a memory driver

SYNOPSIS `STATUS memDrv (void)`

DESCRIPTION This routine initializes the memory driver. It must be called first, before any other routine in the driver.

RETURNS OK, or ERROR if the I/O system cannot install the driver.

SEE ALSO **memDrv**

memFindMax( )

NAME *memFindMax*( ) – find the largest free block in the system memory partition

SYNOPSIS `int memFindMax (void)`

DESCRIPTION This routine searches for the largest block in the system memory partition free list and returns its size.

RETURNS The size, in bytes, of the largest available block.

SEE ALSO **memLib**, *memPartFindMax*( )

memmove( )

NAME *memmove*( ) – copy memory from one location to another (ANSI)

SYNOPSIS

```
void * memmove
    (
    void *       destination, /* destination of copy */
    const void * source,      /* source of copy */
    size_t       size         /* size of memory to copy */
    )
```

DESCRIPTION This routine copies *size* characters from the memory location *source* to the location *destination*. It ensures that the memory is not corrupted even if *source* and *destination* overlap.

INCLUDE FILES **string.h**

RETURNS A pointer to *destination*.

SEE ALSO **ansiString**

memOptionsSet( )

NAME *memOptionsSet*( ) – set the debug options for the system memory partition

SYNOPSIS

```
void memOptionsSet
    (
    unsigned options /* options for system partition */
    )
```

DESCRIPTION This routine sets the debug options for the system memory partition. Two kinds of errors are detected: attempts to allocate more memory than is available, and bad blocks found when memory is freed. In both cases, the following options can be selected for actions to be taken when the error is detected: (1) return the error status, (2) log an error message and return the error status, or (3) log an error message and suspend the calling task.

These options are discussed in detail in the library manual entry for **memLib**.

RETURNS N/A

SEE ALSO **memLib**, *memPartOptionsSet*( )

memPartAddToPool( )

NAME

memPartAddToPool( ) – add memory to a memory partition

SYNOPSIS

```
STATUS memPartAddToPool
    (
    PART_ID  partId,  /* partition to initialize */
    char *   pPool,   /* pointer to memory block */
    unsigned poolSize /* block size in bytes */
    )
```

DESCRIPTION

This routine adds memory to a specified memory partition already created with *memPartCreate*( ). The memory added need not be contiguous with memory previously assigned to the partition.

RETURNS

OK or ERROR.

ERRNO

S_smObjLib_NOT_INITIALIZED, **S_memLib_INVALID_NBYTES**

SEE ALSO

memPartLib, **smMemLib**, *memPartCreate*( )

memPartAlignedAlloc( )

NAME

memPartAlignedAlloc( ) – allocate aligned memory from a partition

SYNOPSIS

```
void *memPartAlignedAlloc
    (
    PART_ID  partId,    /* memory partition to allocate from */
    unsigned nBytes,    /* number of bytes to allocate */
    unsigned alignment  /* boundary to align to */
    )
```

DESCRIPTION

This routine allocates a buffer of size *nBytes* from a specified partition. Additionally, it insures that the allocated buffer begins on a memory address evenly divisible by *alignment*. The *alignment* parameter must be a power of 2.

RETURNS

A pointer to the newly allocated block, or NULL if the buffer could not be allocated.

SEE ALSO

memPartLib

2

memPartAlloc( )

NAME

memPartAlloc( ) – allocate a block of memory from a partition

SYNOPSIS

```
void *memPartAlloc
    (
    PART_ID  partId, /* memory partition to allocate from */
    unsigned nBytes  /* number of bytes to allocate */
    )
```

DESCRIPTION

This routine allocates a block of memory from a specified partition. The size of the block will be equal to or greater than *nBytes*. The partition must already be created with *memPartCreate*( ).

RETURNS

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

ERRNO

S_smObjLib_NOT_INITIALIZED

SEE ALSO

memPartLib, **smMemLib**, *memPartCreate*( )

memPartCreate( )

NAME

memPartCreate( ) – create a memory partition

SYNOPSIS

```
PART_ID memPartCreate
    (
    char *   pPool,   /* pointer to memory area */
    unsigned poolSize /* size in bytes */
    )
```

DESCRIPTION

This routine creates a new memory partition containing a specified memory pool. It returns a partition ID, which can then be passed to other routines to manage the partition (i.e., to allocate and free memory blocks in the partition). Partitions can be created to manage any number of separate memory pools.

NOTE

The descriptor for the new partition is allocated out of the system memory partition (i.e., with *malloc*( )).

RETURNS

The partition ID, or NULL if there is insufficient memory in the system memory partition for a new partition descriptor.

SEE ALSO **memPartLib**, **smMemLib**

memPartFindMax( )

NAME *memPartFindMax*( ) – find the size of the largest available free block

SYNOPSIS

```
int memPartFindMax
    (
    PART_ID partId /* partition ID */
    )
```

DESCRIPTION This routine searches for the largest block in the memory partition free list and returns its size.

RETURNS The size, in bytes, of the largest available block.

ERRNO **S_smObjLib_NOT_INITIALIZED**

SEE ALSO **memLib**, **smMemLib**

memPartFree( )

NAME *memPartFree*( ) – free a block of memory in a partition

SYNOPSIS

```
STATUS memPartFree
    (
    PART_ID partId, /* memory partition to add block to */
    char *  pBlock  /* pointer to block of memory to free */
    )
```

DESCRIPTION This routine returns to a partition's free memory list a block of memory previously allocated with *memPartAlloc*( ).

RETURNS OK, or ERROR if the block is invalid.

ERRNO **S_smObjLib_NOT_INITIALIZED**

SEE ALSO **memPartLib**, **smMemLib**, *memPartAlloc*( )

memPartInfoGet( )

NAME

memPartInfoGet( ) – get partition information

SYNOPSIS

```
STATUS memPartInfoGet
    (
    PART_ID          partId,    /* partition ID */
    MEM_PART_STATS * ppartStats /* partition stats structure */
    )
```

DESCRIPTION

This routine takes a partition ID and a pointer to a **MEM_PART_STATS** structure. All the parameters of the structure are filled in with the current partition information.

RETURNS

OK if the structure has valid data, otherwise ERROR.

SEE ALSO

memShow( )

memPartOptionsSet( )

NAME

memPartOptionsSet( ) – set the debug options for a memory partition

SYNOPSIS

```
STATUS memPartOptionsSet
    (
    PART_ID  partId, /* partition to set option for */
    unsigned options /* memory management options */
    )
```

DESCRIPTION

This routine sets the debug options for a specified memory partition. Two kinds of errors are detected: attempts to allocate more memory than is available, and bad blocks found when memory is freed. In both cases, the error status is returned. There are four error-handling options that can be individually selected:

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

MEM_ALLOC_ERROR_SUSPEND_FLAG
Suspend the task when there is an error in allocating memory (unless the task was spawned with the **VX_UNBREAKABLE** option, in which case it cannot be suspended).

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

MEM_BLOCK_ERROR_SUSPEND_FLAG
Suspend the task when there is an error in freeing memory (unless the task was spawned with the **VX_UNBREAKABLE** option, in which case it cannot be suspended).

These options are discussed in detail in the library manual entry for **memLib**.

RETURNS OK or ERROR.

ERRNO **S_smObjLib_NOT_INITIALIZED**

SEE ALSO **memLib**, **smMemLib**

memPartRealloc( )

NAME *memPartRealloc***( )** – reallocate a block of memory in a specified partition

SYNOPSIS

```
void *memPartRealloc
    (
    PART_ID  partId, /* partition ID */
    char *   pBlock, /* block to be reallocated */
    unsigned nBytes  /* new block size in bytes */
    )
```

DESCRIPTION This routine changes the size of a specified block of memory and returns a pointer to the new block. The contents that fit inside the new size (or old size if smaller) remain unchanged. The memory alignment of the new block is not guaranteed to be the same as the original block.

If *pBlock* is NULL, this call is equivalent to ***memPartAlloc*( )**.

RETURNS A pointer to the new block of memory, or NULL if the call fails.

ERRNO **S_smObjLib_NOT_INITIALIZED**

SEE ALSO **memLib**, **smMemLib**

memPartShow( )

NAME *memPartShow*( ) – show partition blocks and statistics

SYNOPSIS

```
STATUS memPartShow
    (
    PART_ID partId, /* partition ID */
    int     type    /* 0 = statistics, 1 = statistics & list */
    )
```

DESCRIPTION This routine displays statistics about the available and allocated memory in a specified memory partition. It shows the number of bytes, the number of blocks, and the average block size in both free and allocated memory, and also the maximum block size of free memory. It also shows the number of blocks currently allocated and the average allocated block size.

In addition, if *type* is 1, the routine displays a list of all the blocks in the free list of the specified partition.

RETURNS OK or ERROR.

ERRNO **S_smObjLib_NOT_INITIALIZED**

SEE ALSO *memShow*( ), *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

memPartSmCreate( )

NAME *memPartSmCreate*( ) – create a shared memory partition (VxMP Opt.)

SYNOPSIS

```
PART_ID memPartSmCreate
    (
    char *   pPool,   /* global address of shared memory area */
    unsigned poolSize /* size in bytes */
    )
```

DESCRIPTION This routine creates a shared memory partition that can be used by tasks on all CPUs in the system. It returns a partition ID which can then be passed to generic **memPartLib** routines to manage the partition (i.e., to allocate and free memory blocks in the partition).

pPool is the global address of shared memory dedicated to the partition. The memory area pointed to by *pPool* must be in the same address space as the shared memory anchor and shared memory pool.

poolSize is the size in bytes of shared memory dedicated to the partition.

Before this routine can be called, the shared memory objects facility must be initialized (see **smMemLib**).

NOTE The descriptor for the new partition is allocated out of an internal dedicated shared memory partition. The maximum number of partitions that can be created is **SM_OBJ_MAX_MEM_PART**.

Memory pool size is rounded down to a 16-byte boundary.

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS The partition ID, or NULL if there is insufficient memory in the dedicated partition for a new partition descriptor.

ERRNO **S_memLib_NOT_ENOUGH_MEMORY**
S_smObjLib_LOCK_TIMEOUT

SEE ALSO **smMemLib**, **memLib**

---

memset( )

NAME *memset*( ) – set a block of memory (ANSI)

SYNOPSIS

```
void * memset
    (
    void * m,   /* block of memory */
    int    c,   /* character to store */
    size_t size /* size of memory */
    )
```

DESCRIPTION This routine stores *c* converted to an **unsigned char** in each of the elements of the array of **unsigned char** beginning at *m*, with size *size*.

INCLUDE FILES **string.h**

RETURNS A pointer to *m*.

SEE ALSO **ansiString**

memShow( )

NAME

memShow( ) – show system memory partition blocks and statistics

SYNOPSIS

```
void memShow
    (
    int type /* 1 = list all blocks in the free list */
    )
```

DESCRIPTION

This routine displays statistics about the available and allocated memory in the system memory partition. It shows the number of bytes, the number of blocks, and the average block size in both free and allocated memory, and also the maximum block size of free memory. It also shows the number of blocks currently allocated and the average allocated block size.

In addition, if *type* is 1, the routine displays a list of all the blocks in the free list of the system partition.

EXAMPLE

```
-> memShow 1
FREE LIST:
  num     addr      size
  --- ---------- ----------
    1   0x3fee18         16
    2   0x3b1434         20
    3    0x4d188    2909400
SUMMARY:
 status   bytes    blocks   avg block  max block
 ------ --------- -------- ---------- ----------
current
   free   2909436        3     969812   2909400
  alloc    969060    16102         60         -
cumulative
  alloc   1143340    16365         69         -
```

RETURNS

N/A

SEE ALSO

memShow, *memPartShow*( ), *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

memShowInit( )

NAME

memShowInit( ) – initialize the memory partition show facility

SYNOPSIS

```
void memShowInit (void)
```

DESCRIPTION

This routine links the memory partition show facility into the VxWorks system. These routines are included automatically when this show facility is configured into VxWorks using either of the following methods:

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

– If you use the Tornado project facility, select **INCLUDE_MEM_SHOW**.

RETURNS

N/A

SEE ALSO

memShow

mib2ErrorAdd( )

NAME

mib2ErrorAdd( ) – change a MIB-II error count

SYNOPSIS

```
STATUS mib2ErrorAdd
    (
    M2_INTERFACETBL * pMib,
    int               errCode,
    int               value
    )
```

DESCRIPTION

This function adds a specified value to one of the MIB-II error counters in a MIB-II interface table. The counter to be altered is specified by the errCode argument. Specifying a negative value reduces the error count, a positive value increases the error count.

RETURNS

OK or ERROR.

SEE ALSO

endLib

mib2Init( )

NAME

mib2Init( ) – initialize a MIB-II structure

SYNOPSIS

```
STATUS mib2Init
    (
    M2_INTERFACETBL * pMib,        /* struct to be initialized */
    long              ifType,      /* ifType from m2Lib.h */
    UCHAR *           phyAddr,     /* MAC/PHY address */
    int               addrLength,  /* MAC/PHY address length */
    int               mtuSize,     /* MTU size */
    int               speed        /* interface speed */
    )
```

DESCRIPTION

Initialize a MIB-II structure. Set all error counts to zero. Assume a 10Mbps Ethernet device.

RETURNS

OK or ERROR.

SEE ALSO

endLib

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 either a full or relative pathname.

This call is supported by the VxWorks NFS and dosFs file systems.

RETURNS

OK, or ERROR if the directory cannot be created.

SEE ALSO

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** components of the **tm**structure are ignored, and the original values of the other components are not restricted to the ranges indicated in **time.h**. On successful completion, the values of **tm_wday** and **tm_yday** are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to the ranges indicated in **time.h**; the final value of **tm_mday** is not set until **tm_mon** and **tm_year** are determined.

INCLUDE FILES

time.h

RETURNS

The calendar time in seconds, or ERROR (-1) if calendar time cannot be calculated.

SEE ALSO

ansiTime

mlock( )

NAME

mlock( ) – lock specified pages into memory (POSIX)

SYNOPSIS

```
int mlock
    (
    const void * addr,
    size_t       len
    )
```

DESCRIPTION

This routine guarantees that the specified pages are memory resident. In VxWorks, the *addr* and *len* arguments are ignored, since all pages are memory resident.

RETURNS

0 (OK) always.

SEE ALSO

mmanPxLib

mlockall( )

NAME

mlockall( ) – lock all pages used by a process into memory (POSIX)

SYNOPSIS

```
int mlockall
    (
    int flags
    )
```

DESCRIPTION

This routine guarantees that all pages used by a process are memory resident. In VxWorks, the *flags* argument is ignored, since all pages are memory resident.

RETURNS

0 (OK) always.

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

mmuPro32LibInit( )

NAME *mmuPro32LibInit*( ) – initialize module

SYNOPSIS

```
STATUS mmuPro32LibInit
    (
    int pageSize /* system pageSize (must be 4KB or 4MB) */
    )
```

DESCRIPTION Build a dummy translation table that will hold the page table entries for the global translation table. The mmu remains disabled upon completion.

RETURNS OK if no error, ERROR otherwise

ERRNO **S_mmuLib_INVALID_PAGE_SIZE**

SEE ALSO **mmuPro32Lib**

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. The same **sysPhysMemDesc** is used by this routine as well as ***usrMmuInit*( )** in usrConfig to maintain consistency.

RETURNS OK.

SEE ALSO **mmuSparcILib**

2

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*

motCpmEndLoad( )

NAME

motCpmEndLoad( ) – initialize the driver and device

SYNOPSIS

```
END_OBJ *motCpmEndLoad
    (
    char * initString /* parameter string */
    )
```

DESCRIPTION

This routine initializes the driver and the device to the operational state. All of the device specific parameters are passed in the *initString*, which is of the following format:

unit:*motCpmAddr*:*ivec*:*sccNum*:*txBdNum*:*rxBdNum*:*txBdBase*:*rxBdBase*:*bufBase*

The parameters of this string are individually described in the **motCpmEnd** man page.

The SCC 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 obtains this memory region by making calls to ***cacheDmaMalloc*( )**. Non-cacheable memory space is important whenever the host processor uses cache memory. This is also the case when the MC68EN360 is operating in companion mode and is attached to a processor with cache memory.

After 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. An additional number of buffers are reserved as receive loaner buffers. The number of loaner buffers is a default number of 16.

The user must specify the location of the transmit and receive BDs in the processor's dual ported RAM. *txBdBase* and *rxBdBase* give the offsets from *motCpmAddr* 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.

Multiple individual device units are supported by this driver. Device units can reside on different chips, or could be on different SCCs within a single processor. 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 connects up the interrupt vector *ivec*.

RETURNS

An END object pointer or NULL on error.

SEE ALSO

motCpmEnd, *Motorola MC68EN360 User's Manual* , *Motorola MPC860 User's Manual* , *Motorola MPC821 User's Manual*

motFecEndLoad( )

NAME *motFecEndLoad*( ) – initialize the driver and device

SYNOPSIS

```
END_OBJ* motFecEndLoad
    (
    char * initString /* parameter string */
    )
```

DESCRIPTION This routine initializes both driver and device to an operational state using device specific parameters specified by *initString*.

The parameter string, *initString*, is an ordered list of parameters each separated by a colon. The format of *initString* is:

"*motCpmAddr*:*ivec*:*bufBase*:*bufSize*:*fifoTxBase*:*fifoRxBase*:*tbdNum*:*rbdNum*:*phyAddr*:*isoPhyAddr*:*phyDefMode*:*userFlags*"

The FEC 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.

A default number of transmit/receive buffer descriptors of 32 can be selected by passing zero in the parameters *tbdNum* and *rbdNum*. In other cases, the number of buffers selected should be greater than two.

The *bufBase* 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. The *bufSize* parameter is used to check that this region is large enough with respect to the provided values of both transmit/receive buffer descriptors.

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 cache-safe memory. The attributes of this memory will be checked, and if the memory is not write coherent, this routine will abort and return NULL.

RETURNS an END object pointer, or NULL on error.

SEE ALSO **motFecEnd**, **ifLib**, *MPC860T Fast Ethernet Controller (Supplement to MPC860 User's Manual)*

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. It is called automatically when this show facility is configured into VxWorks using either of the following methods:

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

– If you use the Tornado project facility, select **INCLUDE_POSIX_MQ_SHOW**.

RETURNS OK, or ERROR if an error occurs installing the file pointer show routine.

SEE ALSO **mqPxShow**

mq_close( )

NAME

mq_close( ) – close a message queue (POSIX)

SYNOPSIS

```
int mq_close
    (
    mqd_t mqdes /* message queue descriptor */
    )
```

DESCRIPTION

This routine is used to indicate that the calling task is finished with the specified message queue *mqdes*. The ***mq_close*( )** call deallocates any system resources allocated by the system for use by this task for its message queue. The behavior of a task that is blocked on either a ***mq_send*( )** or ***mq_receive*( )** is undefined when ***mq_close*( )** is called. The *mqdes* parameter will no longer be a valid message queue ID.

RETURNS

0 (OK) if the message queue is closed successfully, otherwise -1 (ERROR).

ERRNO

EBADF

SEE ALSO

mqPxLib, *mq_open*( )

mq_getattr( )

NAME

mq_getattr( ) – get message queue attributes (POSIX)

SYNOPSIS

```
int mq_getattr
    (
    mqd_t            mqdes,  /* message queue descriptor */
    struct mq_attr * pMqStat /* buffer in which to return attributes */
    )
```

DESCRIPTION

This routine gets status information and attributes associated with a specified message queue *mqdes*. Upon return, the following members of the **mq_attr** structure referenced by *pMqStat* will contain the values set when the message queue was created but with modifications made by subsequent calls to ***mq_setattr*( )**:

mq_flags
May be modified by ***mq_setattr*( )**.

The following were set at message queue creation:

mq_maxmsg
Maximum number of messages.

2

mq_msgsize
Maximum message size.

mq_curmsgs
The number of messages currently in the queue.

RETURNS 0 (OK) if message attributes can be determined, otherwise -1 (ERROR).

ERRNO **EBADF**

SEE ALSO **mqPxLib**, ***mq_open*( )**, ***mq_send*( )**, ***mq_setattr*( )**

---

mq_notify( )

NAME *mq_notify*( ) – notify a task that a message is available on a queue (POSIX)

SYNOPSIS

```
int mq_notify
    (
    mqd_t                    mqdes,        /* message queue descriptor */
    const struct sigevent * pNotification /* real-time signal */
    )
```

DESCRIPTION If *pNotification* is not NULL, this routine attaches the specified *pNotification* request by the calling task to the specified message queue *mqdes* associated with the calling task. The real-time signal specified by *pNotification* will be sent to the task when the message queue changes from empty to non-empty. If a task has already attached a notification request to the message queue, all subsequent attempts to attach a notification to the message queue will fail. A task is able to attach a single notification to each *mqdes* it has unless another task has already attached one.

If *pNotification* is NULL and the task has previously attached a notification request to the message queue, the attached notification request is detached and the queue is available for another task to attach a notification request.

If a notification request is attached to a message queue and any task is blocked in ***mq_receive*( )** waiting to receive a message when a message arrives at the queue, then the appropriate ***mq_receive*( )** will be completed and the notification request remains pending.

RETURNS 0 (OK) if successful, otherwise -1 (ERROR).

ERRNO **EBADF**, **EBUSY**, **EINVAL**

SEE ALSO **mqPxLib**, ***mq_open*( )**, ***mq_send*( )**

mq_open( )

NAME

mq_open( ) – open a message queue (POSIX)

SYNOPSIS

```
mqd_t mq_open
    (
    const char * mqName, /* name of queue to open */
    int          oflags  /* open flags */
    )
```

DESCRIPTION

This routine establishes a connection between a named message queue and the calling task. After a call to ***mq_open*( )**, the task can reference the message queue using the address returned by the call. The message queue remains usable until the queue is closed by a successful call to ***mq_close*( )**.

The *oflags* argument controls whether the message queue is created or merely accessed by the ***mq_open*( )** call. The following flag bits can be set in *oflags*:

O_RDONLY
Open the message queue for receiving messages. The task can use the returned message queue descriptor with ***mq_receive*( )**, but not ***mq_send*( )**.

O_WRONLY
Open the message queue for sending messages. The task can use the returned message queue descriptor with ***mq_send*( )**, but not ***mq_receive*( )**.

O_RDWR
Open the queue for both receiving and sending messages. The task can use any of the functions allowed for **O_RDONLY** and **O_WRONLY**.

Any combination of the remaining flags can be specified in *oflags*:

O_CREAT
This flag is used to create a message queue if it does not already exist. If **O_CREAT** is set and the message queue already exists, then **O_CREAT** has no effect except as noted below under **O_EXCL**. Otherwise, ***mq_open*( )** creates a message queue. The **O_CREAT** flag requires a third and fourth argument: *mode*, which is of type **mode_t**, and *pAttr*, which is of type pointer to an **mq_attr** structure. The value of *mode* has no effect in this implementation. If *pAttr* is NULL, the message queue is created with implementation-defined default message queue attributes. If *pAttr*is non-NULL, the message queue attributes **mq_maxmsg** and **mq_msgsize** are set to the values of the corresponding members in the **mq_attr** structure referred to by *pAttr*; if either attribute is less than or equal to zero, an error is returned and errno is set to **EINVAL**.

O_EXCL
This flag is used to test whether a message queue already exists. If **O_EXCL** and **O_CREAT** are set, ***mq_open*( )** fails if the message queue name exists.

O_NONBLOCK
The setting of this flag is associated with the open message queue descriptor and determines whether a ***mq_send*( )** or ***mq_receive*( )** will wait for resources or messages that are not currently available, or fail with errno set to **EAGAIN**.

The ***mq_open*( )** call does not add or remove messages from the queue.

NOTE Some POSIX functionality is not yet supported:

– A message queue cannot be closed with calls to ***_exit*( )** or ***exec*( )**.
– A message queue cannot be implemented as a file.
– Message queue names will not appear in the file system.

RETURNS A message queue descriptor, otherwise -1 (ERROR).

ERRNO **EEXIST**, **EINVAL**, **ENOENT**, **ENOSPC**

SEE ALSO **mqPxLib**, ***mq_send*( )**, ***mq_receive*( )**, ***mq_close*( )**, ***mq_setattr*( )**, ***mq_getattr*( )**, ***mq_unlink*( )**

---

mq_receive( )

NAME ***mq_receive*( )** – receive a message from a message queue (POSIX)

SYNOPSIS

```
ssize_t mq_receive
    (
    mqd_t  mqdes,   /* message queue descriptor */
    void * pMsg,    /* buffer to receive message */
    size_t msgLen,  /* size of buffer, in bytes */
    int *  pMsgPrio /* if not NULL, priority of message */
    )
```

DESCRIPTION This routine receives the oldest of the highest priority message from the message queue specified by *mqdes*. If the size of the buffer in bytes, specified by the *msgLen* argument, is less than the **mq_msgsize**attribute of the message queue, ***mq_receive*( )** will fail and return an error. Otherwise, the selected message is removed from the queue and copied to *pMsg*.

If *pMsgPrio* is not NULL, the priority of the selected message will be stored in *pMsgPrio*.

If the message queue is empty and **O_NONBLOCK** is not set in the message queue's description, ***mq_receive*( )** will block until a message is added to the message queue, or until it is interrupted by a signal. If more than one task is waiting to receive a message when a message arrives at an empty queue, the task of highest priority that has been waiting the longest will be selected to receive the message. If the specified message queue is empty and **O_NONBLOCK** is set in the message queue's description, no message is removed from the queue, and ***mq_receive*( )** returns an error.

RETURNS
The length of the selected message in bytes, otherwise -1 (ERROR).

ERRNO
EAGAIN, **EBADF**, **EMSGSIZE**, **EINTR**

SEE ALSO
mqPxLib, ***mq_send*( )**

mq_send( )

NAME
***mq_send*( )** – send a message to a message queue (POSIX)

SYNOPSIS

```
int mq_send
    (
    mqd_t        mqdes,  /* message queue descriptor */
    const void * pMsg,   /* message to send */
    size_t       msgLen, /* size of message, in bytes */
    int          msgPrio /* priority of message */
    )
```

DESCRIPTION
This routine adds the message *pMsg* to the message queue *mqdes*. The *msgLen* parameter specifies the length of the message in bytes pointed to by *pMsg*. The value of *pMsg* must be less than or equal to the **mq_msgsize** attribute of the message queue, or ***mq_send*( )** will fail.

If the message queue is not full, ***mq_send*( )** will behave as if the message is inserted into the message queue at the position indicated by the *msgPrio* argument. A message with a higher numeric value for *msgPrio*is inserted before messages with a lower value. The value of *msgPrio* must be less than or equal to 31.

If the specified message queue is full and **O_NONBLOCK** is not set in the message queue's, ***mq_send*( )** will block until space becomes available to queue the message, or until it is interrupted by a signal. The priority scheduling option is supported in the event that there is more than one task waiting on space becoming available. If the message queue is full and **O_NONBLOCK** is set in the message queue's description, the message is not queued, and ***mq_send*( )** returns an error.

USE BY INTERRUPT SERVICE ROUTINES
This routine can be called by interrupt service routines as well as by tasks. This is one of the primary means of communication between an interrupt service routine and a task. If ***mq_send*( )** is called from an interrupt service routine, it will behave as if the **O_NONBLOCK** flag were set.

RETURNS 0 (OK), otherwise -1 (ERROR).

ERRNO **EAGAIN**, **EBADF**, **EINTR**, **EINVAL**, **EMSGSIZE**

SEE ALSO **mqPxLib**, *mq_receive*( )

mq_setattr( )

NAME *mq_setattr*( ) – set message queue attributes (POSIX)

SYNOPSIS

```
int mq_setattr
    (
    mqd_t                  mqdes,     /* message queue descriptor */
    const struct mq_attr * pMqStat,   /* new attributes */
    struct mq_attr *       pOldMqStat /* old attributes */
    )
```

DESCRIPTION This routine sets attributes associated with the specified message queue *mqdes*.

The message queue attributes corresponding to the following members defined in the **mq_attr** structure are set to the specified values upon successful completion of the call:

mq_flags
The value the **O_NONBLOCK** flag.

If *pOldMqStat* is non-NULL, *mq_setattr*( ) will store, in the location referenced by *pOldMqStat*, the previous message queue attributes and the current queue status. These values are the same as would be returned by a call to *mq_getattr*( ) at that point.

RETURNS 0 (OK) if attributes are set successfully, otherwise -1 (ERROR).

ERRNO **EBADF**

SEE ALSO **mqPxLib**, *mq_open*( ), *mq_send*( ), *mq_getattr*( )

mq_unlink( )

NAME *mq_unlink*( ) – remove a message queue (POSIX)

SYNOPSIS

```
int mq_unlink
    (
    const char * mqName /* name of message queue */
    )
```

DESCRIPTION This routine removes the message queue named by the pathname *mqName*. After a successful call to ***mq_unlink*( )**, a call to ***mq_open*( )** on the same message queue will fail if the flag **O_CREAT** is not set. If one or more tasks have the message queue open when ***mq_unlink*( )** is called, removal of the message queue is postponed until all references to the message queue have been closed.

RETURNS 0 (OK) if the message queue is unlinked successfully, otherwise -1 (ERROR).

ERRNO ENOENT

SEE ALSO **mqPxLib**, *mq_close*( ), *mq_open*( )

mRegs( )

NAME *mRegs*( ) – modify registers

SYNOPSIS

```
STATUS mRegs
    (
    char * regName,     /* register name, NULL for all */
    int    taskNameOrId /* task name or task ID, 0 = default task */
    )
```

DESCRIPTION This command modifies the specified register for the specified task. If *taskNameOrId* is omitted or zero, the last task referenced is assumed. If the specified register is not found, it prints out the valid register list and returns ERROR. If no register is specified, it sequentially prompts the user for new values for a task's registers. It displays each register and the current contents of that register, in turn. The user can respond in one of several ways:

RETURN
    Do not change this register, but continue, prompting at the next register.

number
Set this register to *number*.

. (dot)
Do not change this register, and quit.

EOF
Do not change this register, and quit.

All numbers are entered and displayed in hexadecimal, except floating-point values, which may be entered in double precision.

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

SEE ALSO
usrLib, ***m*( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

---

mRouteAdd( )

NAME
***mRouteAdd*( )** – add multiple routes to the same destination

SYNOPSIS
```
STATUS mRouteAdd
    (
    char * pDest, /* destination addr in internet dot notation */
    char * pGate, /* gateway address in internet dot notation */
    long   mask,  /* mask for destination */
    int    tos,   /* type of service */
    int    flags  /* route flags */
    )
```

DESCRIPTION
This routine is similar to ***routeAdd*( )**, except that you can use multiple ***mRouteAdd*( )** calls to add multiple routes to the same location. Use *pDest* to specify the destination, *pGate* to specify the gateway to that destination, *mask* to specify destination mask, and *tos* to specify the type of service. For *tos*, **netinet/ip.h** defines the following constants as valid values:

IPTOS_LOWDELAY
IPTOS_THROUGHPUT
IPTOS_RELIABILITY
IPTOS_MINCOST

Use *flags* to specify any flags you want to associate with this entry. The valid non-zero values are **RTF_HOST** and **RTF_CLONING** defined in **net/route.h**.

EXAMPLE
To add a route to the 90.0.0.0 network through 91.0.0.3:

```
-> mRouteAdd ("90.0.0.0", "91.0.0.3", 0xffffff00, 0, 0);
```

Using ***mRouteAdd*( )**, you could create multiple routes to the same destination. VxWorks would distinguish among these routes based on factors such as the netmask or the type of service. Thus, it is perfectly legal to say:

```
-> mRouteAdd ("90.0.0.0", "91.0.0.3", 0xffffff00, 0, 0);
-> mRouteAdd ("90.0.0.0", "91.0.0.254", 0xffff0000, 0, 0);
```

This adds two routes to the same network, "90.0.0.0", that go by two different gateways. The differentiating factor is the netmask.

This routine adds a route of type **M2_ipRouteProto_other**, which is a static route. This route will not be modified or deleted until a call to ***mRouteDelete*( )** removes it.

RETURNS OK or ERROR.

SEE ALSO **routeLib**, ***mRouteEntryAdd*( )**, ***mRouteDelete*( )**, ***routeAdd*( )**

mRouteDelete( )

NAME ***mRouteDelete*( )** – delete a route from the routing table

SYNOPSIS

```
STATUS mRouteDelete
    (
    char * pDest, /* destination address */
    long   mask,  /* mask for destination */
    int    tos,   /* type of service */
    int    flags  /* either 0 or RTF_HOST */
    )
```

DESCRIPTION This routine deletes a routing table entry as specified by the destination, *pDest*, the destination mask, *mask*, and type of service, *tos*. The *tos* values are as defined in the reference entry for ***mRouteAdd*( )**.

EXAMPLE Consider the case of a route added in the following manner:

```
-> mRouteAdd ("90.0.0.0", "91.0.0.3", 0xffffff00, 0, 0);
```

To delete a route that was added in the above manner, call ***mRouteDelete*( )** as follows:

```
-> mRouteDelete("90.0.0.0", 0xffffff00, 0);
```

If the netmask and or type of service do not match, the route is not deleted.

The value of *flags* should be **RTF_HOST** for host routes, **RTF_CLONING** for routes which need to be cloned, and 0 in all other cases.

RETURNS OK or ERROR.

SEE ALSO **routeLib**, *mRouteAdd*( )

mRouteEntryAdd( )

NAME *mRouteEntryAdd*( ) – add a protocol-specific route to the routing table

SYNOPSIS

```
STATUS mRouteEntryAdd
    (
    long destIp, /* destination address, network order */
    long gateIp, /* gateway address, network order */
    long mask,   /* mask for destination, network order */
    int  tos,    /* type of service */
    int  flags,  /* route flags */
    int  proto   /* routing protocol */
    )
```

DESCRIPTION For a single destination *destIp*, this routine can add additional routes *gateIp* to the routing table. The different routes are distinguished by a destination mask *mask*, the type of service *tos*, and associated flag values *flags*. Valid values for *flags* are 0, **RTF_HOST**, **RTF_CLONING** (defined in **net/route.h**). The *proto* parameter identifies the protocol that generated this route. Values for *proto* may be found in **m2Lib.h**. The *tos* parameter takes one of following values (defined in **netinet/ip.h**):

IPTOS_LOWDELAY
IPTOS_THROUGHPUT
IPTOS_RELIABILITY
IPTOS_MINCOST

RETURNS OK or ERROR.

SEE ALSO **routeLib**, **m2Lib.h**, *mRouteAdd*( ), *mRouteDelete*( )

mRouteEntryDelete( )

NAME

mRouteEntryDelete( ) – delete route from the routing table

SYNOPSIS

```
STATUS mRouteEntryDelete
    (
    long destIp, /* destination address, network order */
    long gateIp, /* gateway address, network order */
    long mask,   /* mask for destination, network order */
    int  tos,    /* type of service */
    int  flags,  /* route flags */
    int  proto   /* routing protocol */
    )
```

DESCRIPTION

This routine deletes a protocol-specific route from the routing table. Specify the route using a destination *pDest*, a gateway *pGate*, a destination mask *mask*, the type of service *tos*, a *flags* value, and a *proto* value that identifies the routing protocol that added the route. The valid values for *flags* are 0 and **RTF_HOST** (defined in **net/route.h**). Values for *proto* may be found in **m2Lib.h** and *tos* is one of the following values defined in **netinet/ip.h**:

IPTOS_LOWDELA
IPTOS_THROUGHPU
IPTOS_RELIABILIT
IPTOS_MINCOST

An existing route is deleted only if it is owned by the protocol specified by *proto*.

RETURNS

OK or ERROR.

SEE ALSO

routeLib

mRouteShow( )

NAME

mRouteShow( ) – print the entries of the routing table

SYNOPSIS

```
void mRouteShow
    (
    )
```

DESCRIPTION

This routine prints the route entries in the routing table.

RETURNS N/A

SEE ALSO **netShow**

msgQCreate( )

NAME *msgQCreate*( ) – create and initialize a message queue

SYNOPSIS

```
MSG_Q_ID msgQCreate
    (
    int maxMsgs,      /* max messages that can be queued */
    int maxMsgLength, /* max bytes in a message */
    int options       /* message queue options */
    )
```

DESCRIPTION This routine creates a message queue capable of holding up to *maxMsgs*messages, each up to *maxMsgLength* bytes long. The routine returns a message queue ID used to identify the created message queue in all subsequent calls to routines in this library. The queue can be created with the following options:

MSG_Q_FIFO (0x00)
queue pended tasks in FIFO order.

MSG_Q_PRIORITY (0x01)
queue pended tasks in priority order.

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

ERRNO **S_memLib_NOT_ENOUGH_MEMORY**, **S_intLib_NOT_ISR_CALLABLE**

SEE ALSO **msgQLib**, **msgQSmLib**

msgQDelete( )

NAME *msgQDelete*( ) – delete a message queue

SYNOPSIS

```
STATUS msgQDelete
    (
    MSG_Q_ID msgQId /* message queue to delete */
    )
```

DESCRIPTION
This routine deletes a message queue. 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, S_intLib_NOT_ISR_CALLABLE

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_distLib_NOT_INITIALIZED, S_smObjLib_NOT_INITIALIZED, S_objLib_OBJ_ID_ERROR**

SEE ALSO **msgQShow**

msgQNumMsgs( )

NAME *msgQNumMsgs*( ) – get the number of messages queued to a message queue

SYNOPSIS

```
int msgQNumMsgs
    (
    MSG_Q_ID msgQId /* message queue to examine */
    )
```

DESCRIPTION This routine returns the number of messages currently queued to a specified message queue.

RETURNS The number of messages queued, or ERROR.

ERRNO **S_distLib_NOT_INITIALIZED, S_smObjLib_NOT_INITIALIZED, S_objLib_OBJ_ID_ERROR**

SEE ALSO **msgQLib**, **msgQSmLib**

msgQReceive( )

NAME *msgQReceive*( ) – receive a message from a message queue

SYNOPSIS

```
int msgQReceive
    (
    MSG_Q_ID msgQId,    /* message queue from which to receive */
    char *   buffer,    /* buffer to receive message */
    UINT     maxNBytes, /* length of buffer */
    int      timeout    /* ticks to wait */
    )
```

DESCRIPTION This routine receives a message from the message queue *msgQId*. The received message is copied into the specified *buffer*, which is *maxNBytes* in length. If the message is longer than *maxNBytes*, the remainder of the message is discarded (no error indication is returned).

The *timeout* parameter specifies the number of ticks to wait for a message to be sent to the queue, if no message is available when ***msgQReceive*( )** is called. The *timeout* parameter can also have the following special values:

NO_WAIT (0)
return immediately, 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_distLib_NOT_INITIALIZED**, **S_smObjLib_NOT_INITIALIZED**, **S_objLib_OBJ_ID_ERROR**, **S_objLib_OBJ_DELETED**, **S_objLib_OBJ_UNAVAILABLE**, **S_objLib_OBJ_TIMEOUT**, **S_msgQLib_INVALID_MSG_LENGTH**

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. The possible values are:

MSG_PRI_NORMAL (0)
normal priority; add the message to the tail of the list of queued messages.

MSG_PRI_URGENT (1)
urgent priority; add the message to the head of the list of queued messages.

USE BY INTERRUPT SERVICE ROUTINES

This routine can be called by interrupt service routines as well as by tasks. This is one of the primary means of communication between an interrupt service routine and a task. When called from an interrupt service routine, *timeout* must be **NO_WAIT**.

RETURNS OK or ERROR.

ERRNO **S_distLib_NOT_INITIALIZED**, **S_objLib_OBJ_ID_ERROR**, **S_objLib_OBJ_DELETED**, **S_objLib_OBJ_UNAVAILABLE**, **S_objLib_OBJ_TIMEOUT**, **S_msgQLib_INVALID_MSG_LENGTH**, **S_msgQLib_NON_ZERO_TIMEOUT_AT_INT_LEVEL**

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
```

2

```
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.

ERRNO **S_distLib_NOT_INITIALIZED**, **S_smObjLib_NOT_INITIALIZED**

SEE ALSO **msgQShow**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

---

msgQShowInit( )

NAME *msgQShowInit*( ) – initialize the message queue show facility

SYNOPSIS **void msgQShowInit (void)**

DESCRIPTION This routine links the message queue show facility into the VxWorks system. It is called automatically when the message queue show facility is configured into VxWorks using either of the following methods:

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

– If you use the Tornado project facility, select **INCLUDE_MSG_Q_SHOW**.

RETURNS N/A

SEE ALSO **msgQShow**

msgQSmCreate( )

NAME

msgQSmCreate( ) – create and initialize a shared memory message queue (VxMP Opt.)

SYNOPSIS

```
MSG_Q_ID msgQSmCreate
    (
    int maxMsgs,       /* max messages that can be queued */
    int maxMsgLength,  /* max bytes in a message */
    int options        /* message queue options */
    )
```

DESCRIPTION

This routine creates a shared memory message queue capable of holding up to *maxMsgs* messages, each up to *maxMsgLength* bytes long. It returns a message queue ID used to identify the created message queue. The queue can only be created with the option **MSG_Q_FIFO** (0), thus queuing pended tasks in FIFO order.

The global message queue identifier returned can be used directly by generic message queue handling routines in **msgQLib** -- *msgQSend*( ), *msgQReceive*( ), and *msgQNumMsgs*( ) -- and by the show routines *show*( ) and *msgQShow*( ).

If there is insufficient memory to store the message queue structure in the shared memory message queue partition or if the shared memory system pool cannot handle the requested message queue size, shared memory message queue creation will fail with **errno** set to **S_memLib_NOT_ENOUGH_MEMORY**. This problem can be solved by incrementing the value of **SM_OBJ_MAX_MSG_Qand**/or the shared memory objects dedicated memory size **SM_OBJ_MEM_SIZE**.

Before this routine can be called, the shared memory objects facility must be initialized (see **msgQSmLib**).

AVAILABILITY

This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS

MSG_Q_ID, or NULL if error.

ERRNO

S_memLib_NOT_ENOUGH_MEMORY, **S_intLib_NOT_ISR_CALLABLE**, **S_msgQLib_INVALID_QUEUE_TYPE**, **S_smObjLib_LOCK_TIMEOUT**

SEE ALSO

msgQSmLib, **smObjLib**, **msgQLib**, **msgQShow**

munlock( )

NAME *munlock*( ) – unlock specified pages (POSIX)

SYNOPSIS

```
int munlock
    (
    const void * addr,
    size_t       len
    )
```

DESCRIPTION This routine unlocks specified pages from being memory resident.

RETURNS 0 (OK) always.

ERRNO N/A

SEE ALSO **mmanPxLib**

munlockall( )

NAME *munlockall*( ) – unlock all pages used by a process (POSIX)

SYNOPSIS

```
int munlockall (void)
```

DESCRIPTION This routine unlocks all pages used by a process from being memory resident.

RETURNS 0 (OK) always.

ERRNO N/A

SEE ALSO **mmanPxLib**

muxAddressForm( )

NAME

muxAddressForm( ) – form an address into a packet

SYNOPSIS

```
M_BLK_ID muxAddressForm
    (
    void*    pCookie,  /* cookie that identifies the device */
    M_BLK_ID pMblk,    /* structure to contain packet */
    M_BLK_ID pSrcAddr, /* structure containing source address */
    M_BLK_ID pDstAddr  /* structure containing destination address */
    )
```

DESCRIPTION

This routine accepts the source and destination addressing information through the *pSrcAddr* and *pDstAddr* mBlks and returns an **M_BLK_ID** that points to the assembled link-level header. This routine prepends the link-level header into *pMblk* if there is enough space available or it allocates a new **mBlk-clBlk**-cluster and prepends the new *mBlk* to the **mBlk** chain passed in *pMblk*. This routine returns a pointer to an **mBlk** that contains the link-level header information.

pCookie
: Expects the pointer returned from the *muxBind*( ). This pointer identifies the device to which the MUX has bound this protocol.

pMblk
: Expects a pointer to the **mBlk** structure that contains the packet.

pSrcAddr
: Expects a pointer to the **mBlk** that contains the source address.

pDstAddr
: Expects a pointer to the **mBlk** that contains the destination address.

RETURNS

M_BLK_ID or NULL.

ERRNO

S_muxLib_NO_DEVICE

SEE ALSO

muxLib

muxAddrResFuncAdd( )

NAME

muxAddrResFuncAdd( ) – add an address resolution function

SYNOPSIS

```
STATUS muxAddrResFuncAdd
    (
    long    ifType,     /* Media interface type from m2Lib.h */
    long    protocol,   /* Protocol type from RFC 1700 */
    FUNCPTR addrResFunc /* Function to call. */
    )
```

DESCRIPTION

This routine takes an ifType from **m2Lib.h**, a protocol number from RFC 1700 and a pointer to an address resolution function and installs that function for later retrieval by *muxAddrResFuncGet*( ).

ifType
Expects a media interface or network driver type from **m2Lib.h**

protocol
Expects a network service or protocol type from RFC 1700

addrResFunc
Expects a pointer to an address resolution function for this driver and protocol

RETURNS

OK or ERROR.

SEE ALSO

muxLib

muxAddrResFuncDel( )

NAME

muxAddrResFuncDel( ) – delete an address resolution function

SYNOPSIS

```
STATUS muxAddrResFuncDel
    (
    long ifType,  /* ifType of function you want to delete */
    long protocol /* protocol from which to delete the function */
    )
```

DESCRIPTION

This function takes an ifType (from **m2Lib.h**) and a protocol (from RFC 1700) and deletes the associated address resolution routine (if such exists).

ifType
Expects a media interface or network driver type from **m2Lib.h**

protocol
Expects a network service or protocol type from RFC 1700

RETURNS OK or ERROR.

SEE ALSO **muxLib**

muxAddrResFuncGet( )

NAME *muxAddrResFuncGet*( ) – get the address resolution function for ifType/protocol

SYNOPSIS

```
FUNCPTR muxAddrResFuncGet
    (
    long ifType,  /* ifType from m2Lib.h */
    long protocol /* protocol from RFC 1700 */
    )
```

DESCRIPTION This routine takes an *ifType* (from **m2Lib.h**) and a protocol (from RFC 1700) and returns a pointer to the address resolution function registered for this *ifType*/protocol pair. If no such function exists then NULL is returned.

ifType
Expects a media interface or network driver type from **m2Lib.h**

protocol
Expects a network service or protocol type from RFC 1700

RETURNS FUNCPTR to the routine or NULL.

SEE ALSO **muxLib**

muxBind( )

NAME

muxBind( ) – bind a protocol to the MUX given a driver name

SYNOPSIS

```
END_OBJ* muxBind
    (
    char *                       pName, /* interface name, for example, ln, */
    int                          unit,  /* unit number */
    BOOL (* stackRcvRtn) (void*
    )
```

DESCRIPTION

A protocol uses this routine to bind to a specific driver. The driver is specified by the *pName* and *unit* arguments (for example, ln and 0, ln and 1, ei and 0, ...). The *stackRcvRtn* is called whenever the MUX has a packet of the specified type. If the type is **MUX_PROTO_PROMISC**, the protocol is considered promiscuous and will get all of the packets that the MUX sees.

pName
: Expects a pointer to a character string that contains the name of the device to which this protocol wants to use to send and receive packets.

unit
: Expects a number which is the unit of the device of the type indicated by *pName*.

stackRcvRtn
: Expects a pointer function that the MUX can call when it wants to pass a packet up to the protocol. For a description of how you should write this routine, see the description of a ***stackRcvRtn*( )** provided in *Network Protocol Toolkit User's Guide.*

stackShutdownRtn
: Expects a pointer to the function that the MUX can call to shutdown the protocol. For a description of how to write such a routine, see ***stackShutdownRtn*( )** see the description of a ***stackRcvRtn*( )** provided in *Network Protocol Toolkit User's Guide.*

stackErrorRtn
: Expects a pointer to the function that the MUX can call to give errors to the protocol.

type
: Expects a value that indicates the protocol type. The MUX uses this type to prioritize the protocol. For example, a protocol of type **MUX_PROTO_SNARF** has the highest priority (see the description of protocol prioritizing provided in *Network Protocol Toolkit User's Guide: Writing an NPT Protocol.* Aside from **MUX_PROTO_SNARF** and **MUX_PROTO_PROMISC**, valid protocol types include any of the values specified in RFC1700. If the type is **MUX_PROTO_OUTPUT**, this protocol is an output protocol and all packets that are going to be output on this device are passed to the ***stackRcvRtn*( )** routine before actually being sent down to the device. This would be useful, for instance, for a network service that needs to send packets directly to

another network service, or for loop-back testing. If the ***stackRcvRtn*( )** returns OK, the packet is considered to have been consumed and is no longer available. An output protocol may return ERROR from its ***stackRcvRtn*( )** in order to look at the packet without consuming it.

pProtoName
Expects a pointer to a character string for the name of this protocol. This string can be NULL, in which case a protocol name is assigned internally.

pSpare
Expects a pointer to a structure defined by the protocol. This argument is passed up to the protocol with each received packet.

RETURNS A cookie identifying the network driver to which the mux has bound the protocol.

ERRNO **S_muxLib_NO_DEVICE, S_muxLib_ALREADY_BOUND, S_muxLib_ALLOC_FAILED**

SEE ALSO **muxLib**

muxDevExists( )

NAME *muxDevExists*( ) – tests whether a device is already loaded into the MUX

SYNOPSIS
```
BOOL muxDevExists
    (
    char* pName, /* string containing a device name (ln, ei, ...) */
    int   unit   /* unit number */
    )
```

DESCRIPTION This routine takes a string device name (for example, ln or ei) and a unit number. If this device is already known to the MUX, it returns TRUE. Otherwise, this routine returns FALSE.

pName
Expects a pointer to a string containing the device name

unit
Expects the unit number of the device

RETURNS TRUE if the device exists, else FALSE.

SEE ALSO **muxLib**

2

muxDevLoad( )

NAME

muxDevLoad( ) – load a driver into the MUX

SYNOPSIS

```
END_OBJ* muxDevLoad
    (
    int                          unit,       /* unit number of device */
    END_OBJ* (* endLoad) (char* ,
    void*                        ),          /* load function of the driver */
    char*                        pInitString, /* init string for the driver */
    BOOL                         loaning,    /* we loan buffers */
    void*                        pBSP        /* for BSP group */
    )
```

DESCRIPTION

The ***muxDevLoad*( )** routine loads a network driver into the MUX. Internally, this routine calls the specified ***endLoad*( )** to initialize the software state of the device. After the device is initialized, ***muxDevStart*( )** must be called to start the device.

unit
Expects the unit number of the device.

endLoad
Expects a pointer to the network driver's ***endLoad*( )** entry point.

pInitString
Expects a pointer to an initialization string, a colon-delimited list of options. The ***muxDevLoad*( )** routine passes this along blindly to the ***endLoad*( )** function.

loaning
Expects a boolean value that tells the MUX whether the driver supports buffer loaning on this device. If the low-level device cannot support buffer loaning, passing in TRUE has no effect.

pBSP
This argument is passed blindly to the driver, which may or may not use it. It is provided so that the BSP can pass in tables of functions that the driver can use but which are specific to the particular BSP on which it runs.

RETURNS

A pointer to the new device or NULL if an error occurred.

ERRNO

S_muxLib_LOAD_FAILED

SEE ALSO

muxLib

muxDevStart( )

NAME

muxDevStart( ) – start a device by calling its start routine

SYNOPSIS

```
STATUS muxDevStart
    (
    void* pCookie /* a pointer to cookie returned by muxDevLoad() */
    )
```

DESCRIPTION

This routine starts a device that is already initialized and loaded into the MUX. Internally, ***muxDevStart*( )** calls the device's ***endStart*( )**, which handles registering the driver's interrupt service routine and whatever else is needed to allow the device to handle receiving and transmitting. This call to ***endStart*( )** provides a device-dependent way to put the device into a running state.

pCookie
Expects a pointer to the **END_OBJ** returned from the ***muxDevLoad*( )** that loaded this driver into the MUX. This "cookie" is an identifier for the device.

RETURNS

OK, ENETDOWN if *pCookie* does not represent a valid device, or ERROR if the start routine for the device fails.

ERRNO

S_muxLib_NO_DEVICE

SEE ALSO

muxLib

muxDevStop( )

NAME

muxDevStop( ) – stop a device by calling its stop routine

SYNOPSIS

```
STATUS muxDevStop
    (
    void* pCookie /* pointer to cookie that identifies the device */
    )
```

DESCRIPTION

This routine stops the device specified in the *pCookie* parameter. Internally, ***muxDevStop*( )** calls the device's own stop routine, thus putting the device into a stopped state in a device-dependent manner.

pCookie
Expects the pointer returned as the function value of the ***muxDevLoad*( )** call for this

device. This pointer identifies the device to which the MUX has bound this protocol.

RETURNS OK, ENETDOWN if *pCookie* does not represent a valid device, or ERROR if the stop routine for the device fails.

ERRNO **S_muxLib_NO_DEVICE**

SEE ALSO **muxLib**

---

muxDevUnload( )

NAME ***muxDevUnload*( )** – remove a driver from the MUX

SYNOPSIS

```
STATUS muxDevUnload
    (
    char * pName, /* a string containing the name of the device */
                  /* for example, ln or ei */
    int    unit   /* the unit number */
    )
```

DESCRIPTION This routine unloads a driver from the MUX. This breaks any network connections an application might have open. The ***stackShutdownRtn*( )** of each protocol bound to the END via ***muxBind*( )** will be called. Each ***stackShutdownRtn*( )** is expected to call ***muxUnbind*( )** to detach from the END.

pName
Expects a pointer to a string containing the name of the device, for example **ln** or **ei**

unit
Expects the unit number of the device indicated by *pName*

RETURNS OK on success, **EINVAL** or ERROR if the device's registered ***endUnload*( )** function failed, if the specified device was not found, or some other error occurred

ERRNO **S_muxLib_UNLOAD_FAILED, S_muxLib_NO_DEVICE**

SEE ALSO **muxLib**

muxIoctl( )

NAME ***muxIoctl*( )** – send control information to the MUX or to a device

SYNOPSIS

```
STATUS muxIoctl
    (
    void*    pCookie, /* cookie identifying the device to access */
    int      cmd,     /* command to pass to ioctl */
    caddr_t  data     /* data need for command in cmd */
    )
```

DESCRIPTION This routine gives the protocol access to the network driver's control functions. The MUX itself can implement some of the standard control functions, so not all commands necessarily pass down to the device. Otherwise, both command and data pass down to the device unmolested.

This routine also lets the protocol change the routine that the MUX uses to pass data up to the protocol as well as the routine that the MUX uses to shutdown the protocol.

pCookie
: Expects the pointer returned as the function value of ***muxBind*( )**. The pointer identifies the device to which this protocol is bound.

cmd
: Expects a value indicating the control command you want to execute. For valid *cmd* values, see the description of the ***endIoctl*( )** routine provided in *Network Protocol Toolkit User's Guide* .

data
: Expects the data or a pointer to the data needed to carry out the command specified in *cmd*.

RETURNS OK, ENETDOWN if pCookie does not represent a bound device, or ERROR if the command fails.

ERRNO **S_muxLib_NO_DEVICE**

SEE ALSO **muxLib**

muxLibInit( )

NAME *muxLibInit*( ) – initialize global state for the MUX

SYNOPSIS

```
STATUS muxLibInit (void)
```

DESCRIPTION This routine initializes all global state for the MUX.

RETURNS OK or ERROR.

SEE ALSO **muxLib**

muxMCastAddrAdd( )

NAME *muxMCastAddrAdd*( ) – add a multicast address to multicast table for a device

SYNOPSIS

```
STATUS muxMCastAddrAdd
    (
    void*  pCookie, /* returned by the muxBind() call */
    char * pAddress /* address to add to the table */
    )
```

DESCRIPTION This routine adds an address to the multicast table maintained for a device. Internally, this function uses *pCookie* to find the device-specific routine that handles adding an address to the device's multicast table.

pCookie
: Expects the pointer returned as the function value of the ***muxBind*( )** call. This pointer identifies the device to which the MUX has bound this protocol.

pAddress
: Expects a pointer to a character string containing the address you want to add.

RETURNS OK, ENETDOWN if *pCookie* doesn't represent a valid device, or ERROR if the device's ***endMCastAddrAdd*( )** function fails.

ERRNO **ENOTSUP, S_muxLib_NO_DEVICE**

SEE ALSO **muxLib**

muxMCastAddrDel( )

NAME

muxMCastAddrDel( ) – delete a multicast address from a device's multicast table

SYNOPSIS

```
STATUS muxMCastAddrDel
    (
    void* pCookie, /* Returned by the muxBind() call */
    char* pAddress /* Address to delete from the table. */
    )
```

DESCRIPTION

This routine deletes an address from the multicast table maintained for a device. Internally, this function uses *pCookie* to find the device-specific routine that handles deleting an address from the device's multicast table.

pCookie
Expects the pointer returned as the function value of the ***muxBind*( )** call. This pointer identifies the device to which the MUX bound this protocol.

pAddress
Expects a pointer to a character string containing the address you want to delete.

RETURNS

OK, ENETDOWN if *pCookie* does not represent a valid driver, or ERROR if the driver's registered ***endMCastAddrDel*( )** function fails.

ERRNO

ENOTSUP, EINVAL, S_muxLib_NO_DEVICE

SEE ALSO

muxLib

muxMCastAddrGet( )

NAME

muxMCastAddrGet( ) – get the multicast address table from the MUX/Driver

SYNOPSIS

```
int muxMCastAddrGet
    (
    void*         pCookie, /* returned by the muxBind() call */
    MULTI_TABLE * pTable   /* ptr to a table to be filled and returned. */
    )
```

DESCRIPTION

This routine expects a buffer into which it can write the list of multicast addresses for the specified device. Internally, this routine uses *pCookie* to access the device-specific routine needed to retrieve the multicast address table.

pCookie
Expects the pointer returned as the function value of the ***muxBind*( )** call. This pointer identifies the device to which the MUX has bound this protocol.

pTable
Expects the pointer to a **MULTI_TABLE** structure. You must have allocated this structure at some time before the call to ***muxMCastAddrGet*( )**. The **MULTI_TABLE** structure is defined in **end.h** as:

```
typedef struct multi_table
    {
    int     tableLen;   /* length of table in bytes */
    char    *pTable;    /* pointer to entries */
    } MULTI_TABLE;
```

RETURNS OK, ENETDOWN if *pCookie* does not represent a valid driver, or ERROR if the driver's registered ***endMCastAddrGet*( )** function fails.

ERRNO **S_muxLib_NO_DEVICE**

SEE ALSO **muxLib**

---

muxPacketAddrGet( )

NAME ***muxPacketAddrGet*( )** – get addressing information from a packet

SYNOPSIS

```
STATUS muxPacketAddrGet
    (
    void*    pCookie,   /* cookie that identifies the device */
    M_BLK_ID pMblk,     /* structure to contain packet */
    M_BLK_ID pSrcAddr,  /* structure containing source address */
    M_BLK_ID pDstAddr,  /* structure containing destination address */
    M_BLK_ID pESrcAddr, /* structure containing the end source */
    M_BLK_ID pEDstAddr  /* structure containing the end destination */
    )
```

DESCRIPTION This routine takes a pointer to cookie that was handed back by ***muxBind*( )**, an **M_BLK_ID** that came from a device and up to four **M_BLK_ID**'s that can receive data pointers.

The routine returns appropriate information on the immediate source, immediate destination, ultimate source and, ultimate destination addresses from the packet pointed to in the first **M_BLK_ID**. This routine is a pass through to the device's own routine which knows how to interpret packets that it has received.

pCookie
Expects the cookie returned from the ***muxBind*( )** call. This cookie identifies the device to which the MUX bound this protocol.

pMblk
Expects an **M_BLK_ID** representing packet data from which the addressing information is to be extracted

pSrcAddr
Expects NULL or an **M_BLK_ID** which will hold the local source address extracted from the packet

pDstAddr
Expects NULL or an **M_BLK_ID** which will hold the local destination address extracted from the packet

pESrcAddr
Expects NULL or an **M_BLK_ID** which will hold the end source address extracted from the packet

pEDstAddr
Expects NULL or an **M_BLK_ID** which will hold the end destination address extracted from the packet

RETURNS OK or ERROR.

ERRNO **S_muxLib_NO_DEVICE**

SEE ALSO **muxLib**

muxPacketDataGet( )

NAME ***muxPacketDataGet*( )** – return the data from a packet

SYNOPSIS

```
STATUS muxPacketDataGet
    (
    void*         pCookie,     /* cookie that identifies the device */
    M_BLK_ID      pMblk,       /* returns the packet data */
    LL_HDR_INFO * pLinkHdrInfo /* the new data is returned here */
    )
```

DESCRIPTION This routine copies the header information from the packet referenced in *pMblk* into the **LL_HDR_INFO** structure referenced in *pLinkHdrInfo*.

pCookie
Expects the cookie returned from the ***muxBind*( )** call. This cookie identifies the device to which the MUX bound this protocol.

pMblk
Expects a pointer to an **mBlk** or **mBlk** cluster representing a packet containing the data to be returned

pLinkHdrInfo
Expects a pointer to an **LL_HDR_INFO** structure into which the packet header information is copied from the incoming **mBlk**

RETURNS OK or ERROR if the device type is not recognized.

ERRNO **S_muxLib_NO_DEVICE**

SEE ALSO **muxLib**

---

muxPollReceive( )

NAME ***muxPollReceive*( )** – poll for a packet from a device driver

SYNOPSIS

```
STATUS muxPollReceive
    (
    void*    pCookie, /* cookie passed in endLoad call */
    M_BLK_ID pNBuff   /* a vector of buffers passed to us */
    )
```

DESCRIPTION This is the routine that an upper layer can call to poll for a packet.

pCookie
Expects the cookie that was returned from ***muxBind*( )**. This "cookie" is an identifier for the driver.

pNBuff
Expects a pointer to a buffer chain into which incoming data will be put.

RETURNS OK, ENETDOWN if *pCookie* does not represent a loaded driver, or an error value returned from the driver's registered ***endPollReceive*( )** function.

ERRNO **S_muxLib_NO_DEVICE**

SEE ALSO **muxLib**

muxPollSend( )

NAME

muxPollSend( ) – send a packet on a network interface

SYNOPSIS

```
STATUS muxPollSend
    (
    void*    pCookie, /* cookie the protocol got from muxBind() */
    M_BLK_ID pNBuff   /* data to be sent */
    )
```

DESCRIPTION

This routine takes a cookie which was returned by ***muxBind*( )** and uses it to determine which network interface driver should be used in transmitting the data. The routine takes the data pointed to by *pNBuff* and sends it to the destination specified by calling the functions in that driver.

pCookie
Expects the cookie returned from ***muxBind*( )**. This Cookie identifies the device to which the MUX has bound the protocol calling ***muxPollSend*( )**.

pNBuff
Expects a pointer to the buffer(**mBlk**) chain that contains the packet to be transmitted.

RETURNS

OK, **ENETDOWN** if *pCookie* doesn't represent a valid device, or ERROR if the device type is not recognized or if the ***endPollSend*( )** routine for the driver fails.

ERRNO

S_muxLib_NO_DEVICE

SEE ALSO

muxLib

muxSend( )

NAME

muxSend( ) – send a packet out on a network interface

SYNOPSIS

```
STATUS muxSend
    (
    void*    pCookie, /* cookie that identifies a network interface */
                      /* by muxBind() */
    M_BLK_ID pNBuff   /* data to be sent */
    )
```

DESCRIPTION

This routine uses the *pCookie* value returned during the bind to identify the network interface through which the packet is to be transmitted.

pCookie
: Expects the pointer returned from ***muxBind*( )**. This pointer identifies the device to which the MUX has bound this protocol.

pNBuff
: Expects a pointer to the buffer that contains the packet you want to transmit. Before you call ***muxSend*( )**, you need to put the addressing information at the head of the buffer. To do this, call ***muxAddressForm*( )**.

Also, the buffer should probably be reserved from the MUX- managed memory pool. To reserve a buffer from this pool, the protocol should call ***muxBufAlloc*( )**.

RETURNS

OK, ENETDOWN if *pCookie* does not represent a valid interface, or ERROR if the driver's ***endSend*( )** routine fails.

ERRNO

S_muxLib_NO_DEVICE

SEE ALSO

muxLib

---

muxShow( )

NAME

***muxShow*( )** – all configured Enhanced Network Drivers

SYNOPSIS

```
void muxShow
    (
    char * pDevName, /* pointer to device name */
    int    unit      /* unit number for the device */
    )
```

DESCRIPTION

If a driver is specified *pDevName* and *unit*, this routine reports the name and type of each protocol bound to it. If a *pDevName* is not given, the entire list of devices and their protocols is shown.

pDevName
: Expects a pointer to a string containing the device name, or NULL

unit
: Expects a unit number for the device

RETURNS

N/A

SEE ALSO

muxLib

muxUnbind( )

NAME

muxUnbind( ) – detach a protocol from the specified driver

SYNOPSIS

```
STATUS muxUnbind
    (
    void*   pCookie,    /* pointer to identifier for device */
    long    type,       /* device type passed in muxBind() call */
    FUNCPTR stackRcvRtn /* pointer to stack receive routine */
    )
```

DESCRIPTION

This routine disconnects a protocol from the specified driver.

pCookie
Expects the pointer returned as the function value from the ***muxBind*( )** call. This pointer identifies the device to which the MUX has bound this protocol.

type
This is the type that you passed down in the ***muxBind*( )** call.

stackRcvRtn
Expects a pointer to the stack receive routine you specified when you called ***muxBind*( )** to bind the driver and protocol.

RETURNS

OK, **EINVAL** if *pCookie* does not represent a valid driver or the protocol is not attached, ERROR if ***muxUnbind*( )** fails.

ERRNO

EINVAL, S_muxLib_NO_DEVICE

SEE ALSO

muxLib

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*( )

---

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 **ncr710Lib**2

---

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   /* priority of task when doing SCSI */
    )
```

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 **ncr710Lib**2, ***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 {0,0,0,0,1,0,0,0,0,0,0,0,0,1,0}.

```
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 */
```

2

```
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 you cannot use it if there is any SCSI bus activity.

RETURNS OK, or ERROR if any input parameter is NULL.

SEE ALSO **ncr710Lib**2, ***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 the Istat register. 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
value = 0 = 0x0
```

RETURNS OK, or ERROR if *pScsiCtrl* and *pSysScsiCtrl* are both NULL.

SEE ALSO **ncr710Lib**2, ***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**

2

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 {0,0,0,0,0,1,0,0,0,0,0}.

```
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 detail on 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
value = 0 = 0x0
```

RETURNS OK, or ERROR if *pScsiCtrl* and *pSysScsiCtrl* are both NULL.

SEE ALSO **ncr810Lib**, *ncr810CtrlCreate*( )

2

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:

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

---

ncr710SingleStep( )

NAME
ncr710SingleStep( ) – perform a single-step

SYNOPSIS
```
void ncr710SingleStep
    (
    SIOP * pSiop,  /* pointer to SIOP info */
    BOOL   verbose /* show all registers */
    )
```

DESCRIPTION
This routine performs a single-step by writing the STD bit in the DCNTL register. The *pSiop* parameter is a pointer to the SIOP information. Before executing, enable the single-step facility by calling *ncr710StepEnable*( ).

RETURNS
N/A

SEE ALSO
ncr710CommLib, *ncr710StepEnable*( )

---

ncr710StepEnable( )

NAME
ncr710StepEnable( ) – enable/disable script single-step

SYNOPSIS
```
void ncr710StepEnable
    (
    SIOP * pSiop,    /* pointer to SIOP info */
    BOOL   boolValue /* TRUE/FALSE to enable/disable single step */
    )
```

DESCRIPTION

This routine enables/disables the single-step facility on the chip. It also unmasks/masks the single-step interrupt in the Dien register. Before executing any SCSI routines, enable the single-step facility by calling ***ncr710StepEnable*( )** with *boolValue* set to TRUE. To disable, call it with *boolValue* set to FALSE.

RETURNS

N/A

SEE ALSO

ncr710CommLib, *ncr710SingleStep*( )

ne2000EndLoad( )

NAME

ne2000EndLoad( ) – initialize the driver and device

SYNOPSIS

```
END_OBJ* ne2000EndLoad
    (
    char* initString, /* String to be parsed by the driver. */
    void* pBSP        /* for BSP group */
    )
```

DESCRIPTION

This routine initializes the driver and the device to the operational state. All of the device specific parameters are passed in the initString.

The string contains the target specific parameters like this:

"unit:register addr:int vector:int level:shmem addr:shmem size:shmem width"

RETURNS

An END object pointer or NULL on error.

SEE ALSO

ne2000End

ne2000Parse( )

NAME

ne2000Parse( ) – parse the init string

SYNOPSIS

```
STATUS ne2000Parse
    (
    NE2000END_DEVICE * pDrvCtrl,
    char *             initString
    )
```

DESCRIPTION Parse the input string. Fill in values in the driver control structure.

The initialization string format is:
unit:*adrs*:*vecnum*:*intLvl*:*byteAccess*:*usePromEnetAddr*:*offset*

unit
Device unit number, a small integer.

adrs
Base address

vecNum
Interrupt vector number (used with sysIntConnect)

intLvl
Interrupt level (used with sysLanIntEnable)

byteAccess
Use 8-bit access mode.

usePromEnetAddr
get ethernet address from PROM.

offset
offset for memory alignment

RETURNS OK or ERROR for invalid arguments.

SEE ALSO **ne2000End**

---

netBufLibInit( )

NAME *netBufLibInit*( ) – initialize netBufLib

SYNOPSIS

```
STATUS netBufLibInit (void)
```

DESCRIPTION This routine initializes **netBufLib**. If you defined **INCLUDE_NETWORKin configAll.h**, this configured VxWorks to include **netBufLib**.

RETURNS OK or ERROR.

SEE ALSO **netBufLib**

netClBlkFree( )

NAME

netClBlkFree( ) – free a **clBlk**-cluster construct back to the memory pool

SYNOPSIS

```
void netClBlkFree
    (
    NET_POOL_ID pNetPool, /* pointer to the net pool */
    CL_BLK_ID   pClBlk    /* pointer to the clBlk to free */
    )
```

DESCRIPTION

This routine decrements the reference counter in the specified **clBlk**. If the reference count falls to zero, this routine frees both the **clBlk**and its associated cluster back to the specified memory pool.

RETURNS

N/A

SEE ALSO

netBufLib

netClBlkGet( )

NAME

netClBlkGet( ) – get a **clBlk**

SYNOPSIS

```
CL_BLK_ID netClBlkGet
    (
    NET_POOL_ID pNetPool, /* pointer to the net pool */
    int         canWait   /* M_WAIT/M_DONTWAIT */
    )
```

DESCRIPTION

This routine gets a **clBlk** from the specified memory pool.

pNetPool
: Expects a pointer to the pool from which you want a **clBlk**.

canWait
: Expects either **M_WAIT** or **M_DONTWAIT**. If *canWait* is **M_WAIT**, this routine blocks until an **clBlk** is available. If *canWait* is **M_DONTWAITand** no **clBlk** is immediately available, this routine returns immediately (no blocking) with a NULL value.

RETURNS

CL_BLK_ID or a NULL if no **clBlk** was available.

SEE ALSO

netBufLib

2

netClBlkJoin( )

NAME *netClBlkJoin*( ) – join a cluster to a **clBlk** structure

SYNOPSIS

```
CL_BLK_ID netClBlkJoin
    (
    CL_BLK_ID pClBlk,   /* pointer to a cluster Blk */
    char *    pClBuf,   /* pointer to a cluster buffer */
    int       size,     /* size of the cluster buffer */
    FUNCPTR   pFreeRtn, /* pointer to the free routine */
    int       arg1,     /* argument 1 of the free routine */
    int       arg2,     /* argument 2 of the free routine */
    int       arg3      /* argument 3 of the free routine */
    )
```

DESCRIPTION This routine joins the previously reserved cluster specified by *pClBuf* to the previously reserved **clBlk** structure specified by *pClBlk*. The *size* parameter passes in the size of the cluster referenced in *pClBuf*. The arguments *pFreeRtn*, *arg1*, *arg2*, *arg3* set the values of the **pCLFreeRtn**, **clFreeArg1**, **clFreeArg2**, and **clFreeArg1**, members of the specified **clBlk** structure.

RETURNS **CL_BLK_ID** or NULL.

SEE ALSO **netBufLib**

netClFree( )

NAME *netClFree*( ) – free a cluster back to the memory pool

SYNOPSIS

```
void netClFree
    (
    NET_POOL_ID pNetPool, /* pointer to the net pool */
    UCHAR *     pClBuf    /* pointer to the cluster buffer */
    )
```

DESCRIPTION This routine returns the specified cluster buffer back to the specified memory pool.

RETURNS N/A

SEE ALSO **netBufLib**

netClPoolIdGet( )

NAME

netClPoolIdGet( ) – return a **CL_POOL_ID** for a specified buffer size

SYNOPSIS

```
CL_POOL_ID netClPoolIdGet
    (
    NET_POOL_ID pNetPool, /* pointer to the net pool */
    int         bufSize,  /* size of the buffer */
    BOOL        bestFit   /* TRUE/FALSE */
    )
```

DESCRIPTION

This routine returns a **CL_POOL_ID** for a cluster pool containing clusters that match the specified *bufSize*. If bestFit is TRUE, this routine returns a **CL_POOL_ID** for a pool that contains clusters greater than or equal to *bufSize*. If *bestFit* is FALSE, this routine returns a **CL_POOL_ID** for a cluster from whatever cluster pool is available. If the memory pool specified by *pNetPool* contains only one cluster pool, *bestFit* should always be FALSE.

RETURNS

CL_POOL_ID or NULL.

SEE ALSO

netBufLib

netClusterGet( )

NAME

netClusterGet( ) – get a cluster from the specified cluster pool

SYNOPSIS

```
char * netClusterGet
    (
    NET_POOL_ID pNetPool, /* pointer to the net pool */
    CL_POOL_ID  pClPool   /* ptr to the cluster pool */
    )
```

DESCRIPTION

This routine gets a cluster from the specified cluster pool *pClPool* within the specified memory pool *pNetPool*.

RETURNS

This routine returns a character pointer to a cluster buffer or NULL if none was available.

SEE ALSO

netBufLib

2

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 the configuration macro **INCLUDE_NETWORK** is defined.

RETURNS OK or ERROR.

SEE ALSO **netDrv**

netHelp( )

NAME

netHelp( ) – print a synopsis of network routines

SYNOPSIS

```
void netHelp (void)
```

DESCRIPTION

This command prints a brief synopsis of network facilities that are typically called from the shell.

```
hostAdd      "hostname","inetaddr" - add a host to remote host table;
                                     "inetaddr" must be in standard
                                     Internet address format e.g.
"90.0.0.4"
hostShow                           - print current remote host table
netDevCreate "devname","hostname",protocol
                                   - create an I/O device to access
                                     files on the specified host
                                     (protocol 0=rsh, 1=ftp)
routeAdd     "destaddr","gateaddr" - add route to route table
routeDelete  "destaddr","gateaddr" - delete route from route table
routeShow                          - print current route table
iam          "usr"[,"passwd"]      - specify the user name by which
                                     you will be known to remote
                                     hosts (and optional password)
whoami                             - print the current remote ID
rlogin       "host"                - log in to a remote host;
                                     "host" can be inet address or
                                     host name in remote host table
ifShow       ["ifname"]            - show info about network interfaces
inetstatShow                       - show all Internet protocol sockets
tcpstatShow                        - show statistics for TCP
udpstatShow                        - show statistics for UDP
ipstatShow                         - show statistics for IP
icmpstatShow                       - show statistics for ICMP
arptabShow                         - show a list of known ARP entries
mbufShow                           - show mbuf statistics
EXAMPLE:  -> hostAdd "wrs", "90.0.0.2"
          -> netDevCreate "wrs:", "wrs", 0
          -> iam "fred"
          -> copy <wrs:/etc/passwd   /* copy file from host "wrs" */
          -> rlogin "wrs"            /* rlogin to host "wrs"      */
```

RETURNS

N/A

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 the configuration macro **INCLUDE_NETWORK** is defined.

RETURNS OK, or ERROR if network support cannot be initialized.

SEE ALSO **netLib**, usrConfig, *netTask*( )

netMblkChainDup( )

NAME *netMblkChainDup*( ) – duplicate an **mBlk** chain

SYNOPSIS

```
M_BLK_ID netMblkChainDup
    (
    NET_POOL_ID pNetPool, /* pointer to the pool */
    M_BLK_ID    pMblk,    /* pointer to source mBlk chain */
    int         offset,   /* offset to duplicate from */
    int         len,      /* length to copy */
    int         canWait   /* M_DONTWAIT/M_WAIT */
    )
```

DESCRIPTION This routine makes a copy of an **mBlk** chain starting at *offset* bytes from the beginning of the chain and continuing for *len* bytes. If *len* is **M_COPYALL**, then this routine will copy the entire **mBlk** chain from the *offset*.

This routine copies the references from a source *pMblk* chain to a newly allocated **mBlk** chain. This lets the two **mBlk** chains share the same **clBlk**-cluster constructs. This routine also increments the reference count in the shared **clBlk**. The *pMblk* expects a pointer to the source **mBlk**chain. The *pNetPool* parameter expects a pointer to the netPool from which the new **mBlk** chain is allocated.

The *canWait* parameter expects either **M_WAIT** or **M_DONTWAIT**. If *canWait* is **M_WAIT**, this routine blocks until **mBlk** is available. If *canWait* is **M_DONTWAIT** and no **mBlk** is immediately available, this routine returns immediately (no blocking) with a NULL value.

SEE ALSO *netMblkDup***( )**

RETURNS A pointer to the newly allocated **mBlk** chain or NULL.

ERRNO **S_netBufLib_INVALID_ARGUMENT**
S_netBufLib_NO_POOL_MEMORY

*netMblkClChainFree***( )**

NAME *netMblkClChainFree***( )** – free a chain of **mBlk**-**clBlk**-cluster constructs

SYNOPSIS

```
void netMblkClChainFree
    (
    M_BLK_ID pMblk /* pointer to the mBlk */
    )
```

DESCRIPTION For the specified chain of **mBlk**-**clBlk**-cluster constructs, this routine frees all the **mBlk** structures back to the specified memory pool. It also decrements the reference count in all the **clBlk** structures. If the reference count in a **clBlk** falls to zero, this routine also frees that **clBlk** and its associated cluster back to the specified memory pool.

RETURNS N/A

ERRNO **S_netBufLib_MBLK_INVALID**

SEE ALSO **netBufLib**

*netMblkClFree***( )**

NAME *netMblkClFree***( )** – free an **mBlk**-**clBlk**-cluster construct

SYNOPSIS

```
M_BLK_ID netMblkClFree
    (
    M_BLK_ID pMblk /* pointer to the mBlk */
    )
```

DESCRIPTION For the specified **mBlk**-**clBlk**-cluster construct, this routine frees the **mBlk** back to the specified memory pool. It also decrements the reference count in the **clBlk** structure. If the reference count falls to zero, no other **mBlk** structure reference this **clBlk**. In that case, this routine also frees the **clBlk** structure and its associated cluster back to the specified memory pool.

RETURNS
If the specified **mBlk** was part of an **mBlk** chain, this routine returns a pointer to the next **mBlk**. Otherwise, it returns a NULL.

ERRNO
S_netBufLib_MBLK_INVALID

SEE ALSO
netBufLib

---

netMblkClGet( )

NAME
netMblkClGet( ) – get a **clBlk**-cluster and join it to the specified **mBlk**

SYNOPSIS
```
STATUS netMblkClGet
    (
    NET_POOL_ID pNetPool, /* pointer to the net pool */
    M_BLK_ID    pMblk,    /* mBlk to embed the cluster in */
    int         bufSize,  /* size of the buffer to get */
    int         canWait,  /* wait or dontwait */
    BOOL        bestFit   /* TRUE/FALSE */
    )
```

DESCRIPTION
This routine gets a **clBlk**-cluster construct from the specified memory pool and joins it to the specified **mBlk** structure. This creates an **mBlk**-**clBlk**-cluster construct that you can use to pass data across the layers of the network stack.

pNetPool
Expects a pointer to the memory pool from which you want to get a free **clBlk**-cluster construct.

pMbkl
Expects a pointer to the **mBlk** structure (previously allocated) to which you want to join the retrieved **clBlk**-cluster construct.

bufSize
Expects the size, in bytes, of the cluster in the **clBlk**-cluster construct.

canWait
Expects either **M_WAIT** or **M_DONTWAIT**. If *canWait* is **M_WAIT**, this routine blocks until a **clBlk**-cluster construct is available. If *canWait* is **M_DONTWAIT** and no **clBlk**-cluster construct is immediately available, this routine returns immediately (no blocking) with an ERROR value.

bestFit

Expects either TRUE or FALSE. If *bestFit* is TRUE and a cluster of the exact size is unavailable, this routine gets a larger cluster (if available). If *bestFit* is FALSE and an exact size cluster is unavailable, this routine gets either a smaller or a larger cluster (depending on what is available). Otherwise, it returns immediately with an ERROR value. For memory pools containing only one cluster size, *bestFit* should always be set to FALSE.

RETURNS OK or ERROR.

ERRNO **S_netBufLib_CLSIZE_INVALID**

SEE ALSO **netBufLib**

netMblkClJoin( )

NAME *netMblkClJoin*( ) – join an **mBlk** to a **clBlk**-cluster construct

SYNOPSIS

```
M_BLK_ID netMblkClJoin
    (
    M_BLK_ID  pMblk, /* pointer to an mBlk */
    CL_BLK_ID pClBlk /* pointer to a cluster Blk */
    )
```

DESCRIPTION This routine joins the previously reserved **mBlk** referenced in *pMblk* to the **clBlk**-cluster construct referenced in *pClBlk*. Internally, this routine sets the **M_EXT** flag in **mBlk.mBlkHdr.mFlags**. It also and sets the **mBlk.mBlkHdr.mData** to point to the start of the data in the cluster.

RETURNS **M_BLK_ID** or NULL.

SEE ALSO **netBufLib**

2

netMblkDup( )

NAME *netMblkDup*( ) – duplicate an **mBlk**

SYNOPSIS

```
M_BLK_ID netMblkDup
    (
    M_BLK_ID pSrcMblk, /* pointer to source mBlk */
    M_BLK_ID pDestMblk /* pointer to the destination mBlk */
    )
```

DESCRIPTION This routine copies the references from a source **mBlk** in an **mBlk**-**clBlk**-cluster construct to a stand-alone **mBlk**. This lets the two **mBlk** structures share the same **clBlk**-cluster construct. This routine also increments the reference count in the shared **clBlk**. The *pSrcMblk* expects a pointer to the source **mBlk**. The *pDescMblk* parameter expects a pointer to the destination **mBlk**.

RETURNS A pointer to the destination **mBlk** or NULL if the source **mBlk** referenced in *pSrcMblk* is not part of a valid **mBlk**-**clBlk**-cluster construct.

SEE ALSO **netBufLib**

netMblkFree( )

NAME *netMblkFree*( ) – free an **mBlk** back to its memory pool

SYNOPSIS

```
void netMblkFree
    (
    NET_POOL_ID pNetPool, /* pointer to the net pool */
    M_BLK_ID    pMblk     /* mBlk to free */
    )
```

DESCRIPTION This routine frees the specified **mBlk** back to the specified memory pool.

RETURNS N/A

SEE ALSO **netBufLib**

netMblkGet( )

NAME

netMblkGet( ) – get an **mBlk**

SYNOPSIS

```
M_BLK_ID netMblkGet
    (
    NET_POOL_ID pNetPool, /* pointer to the net pool */
    int         canWait,  /* M_WAIT/M_DONTWAIT */
    UCHAR       type      /* mBlk type */
    )
```

DESCRIPTION

This routine gets a **mBlk** from the specified memory pool.

pNetPool
Expects a pointer to the pool from which you want an **mBlk**.

canWait
Expects either **M_WAIT** or **M_DONTWAIT**. If *canWait* is **M_WAIT**, this routine blocks until an **mBlk** is available. If *canWait* is **M_DONTWAITand** no **mBlk** is immediately available, this routine returns immediately (no blocking) with a NULL value.

type
Expects the type value that you want to associate with the returned **mBlk**.

RETURNS

M_BLK_ID, or a NULL if no **mBlk** was available.

ERRNO

S_netBufLib_MBLK_INVALID

SEE ALSO

netBufLib

netMblkToBufCopy( )

NAME

netMblkToBufCopy( ) – copy data from an **mBlk** to a buffer

SYNOPSIS

```
int netMblkToBufCopy
    (
    M_BLK_ID pMblk,   /* pointer to an mBlk */
    char *   pBuf,    /* pointer to the buffer to copy */
    FUNCPTR  pCopyRtn /* function pointer for copy routine */
    )
```

DESCRIPTION This routine copies data from the **mBlk** chain referenced in *pMblk* to the buffer referenced in *pBuf*. It is assumed that *pBuf* points to enough memory to contain all the data in the entire **mBlk** chain. The argument *pCopyRtn* expects either a NULL or a function pointer to a copy routine. The arguments passed to the copy routine are source pointer, destination pointer and the length of data to copy. If *pCopyRtn* is NULL, ***netMblkToBufCopy*( )** uses a default routine to extract the data from the chain.

RETURNS The length of data copied or zero.

SEE ALSO **netBufLib**

---

netPoolDelete( )

NAME *netPoolDelete*( ) – delete a memory pool

SYNOPSIS

```
STATUS netPoolDelete
    (
    NET_POOL_ID pNetPool /* pointer to a net pool */
    )
```

DESCRIPTION This routine deletes the specified **netBufLib**-managed memory pool.

RETURNS OK or ERROR.

ERRNO **S_netBufLib_NETPOOL_INVALID**

SEE ALSO **netBufLib**

---

netPoolInit( )

NAME *netPoolInit*( ) – initialize a **netBufLib**-managed memory pool

SYNOPSIS

```
STATUS netPoolInit
    (
    NET_POOL_ID    pNetPool,       /* pointer to a net pool */
    M_CL_CONFIG *  pMclBlkConfig,  /* pointer to a mBlk configuration */
    CL_DESC *      pClDescTbl,     /* pointer to cluster desc table */
    int            clDescTblNumEnt,/* number of cluster desc entries */
    POOL_FUNC *    pFuncTbl        /* pointer to pool function table */
    )
```

DESCRIPTION

Call this routine to set up a **netBufLib**-managed memory pool. Within this pool, ***netPoolInit*( )** organizes several sub-pools: one for **mBlk** structures, one for **clBlk** structures, and as many cluster sub-pools are there are cluster sizes. As input, this routine expects the following parameters:

pNetPool
: Expects a **NET_POOL_ID** that points to a previously allocated **NET_POOL** structure. You need not initialize any values in this structure. That is handled by ***netPoolInit*( )**.

pMclBlkConfig
: Expects a pointer to a previously allocated and initialized **M_CL_CONFIG** structure. Within this structure, you must provide four values: **mBlkNum**, a count of **mBlk** structures; **clBlkNum**, a count of **clBlk** structures; **memArea**, a pointer to an area of memory that can contain all the **mBlk** and **clBlk** structures; and **memSize**, the size of that memory area. For example, you can set up an **M_CL_CONFIG** structure as follows:

```
M_CL_CONFIG mClBlkConfig = /* mBlk, clBlk configuration table */
    {
    mBlkNum     clBlkNum        memArea         memSize
    ----------  ----            -------         -------
    400,        245,            0xfe000000,     21260
    };
```

You can calculate the **memArea** and **memSize** values. Such code could first define a table as shown above, but set both **memArea** and **memSize** as follows:

```
mClBlkConfig.memSize = (mClBlkConfig.mBlkNum * (M_BLK_SZ + sizeof(long))) +
                       (mClBlkConfig.clBlkNum * CL_BLK_SZ);
```

You can set the memArea value to a pointer to private memory, or you can reserve the memory with a call to ***malloc*( )**. For example:

```
mClBlkConfig.memArea = malloc(mClBlkConfig.memSize);
```

The **netBufLib.h** file defines **M_BLK_SZ** as:

```
sizeof(struct mBlk)
```

Currently, this evaluates to 32 bytes. Likewise, this file defines **CL_BLK_SZ** as:

```
sizeof(struct clBlk)
```

Currently, this evaluates to 32 bytes.

When choosing values for **mBlkNum** and **clBlkNum**, remember that you need as many **clBlk** structures as you have clusters (data buffers). You also need at least as many **mBlk** structures as you have **clBlk** structures, but you will most likely need more. That is because **netBufLib** shares buffers by letting multiple **mBlk** structures join to the same **clBlk** and thus to its underlying cluster. The **clBlk** keeps a count of the number of **mBlk** structures that reference it.

pClDescTbl
Expects a pointer to a table of previously allocated and initialized **CL_DESC** structures. Each structure in this table describes a single cluster pool. You need a dedicated cluster pool for each cluster size you want to support. Within each **CL_DESC** structure, you must provide four values: **clusterSize**, the size of a cluster in this cluster pool; **num**, the number of clusters in this cluster pool; **memArea**, a pointer to an area of memory that can contain all the clusters; and **memSize**, the size of that memory area.

Thus, if you need to support six different cluster sizes, this parameter must point to a table containing six **CL_DESC** structures. For example, consider the following:

```
CL_DESC clDescTbl [] =   /* cluster descriptor table */
    {
    /*
    clusterSize        num     memArea         memSize
    ----------         ----    -------         -------
    */
    {64,               100,    0x10000,        6800},
    {128,              50,     0x20000,        6600},
    {256,              50,     0x30000,        13000},
    {512,              25,     0x40000,        12900},
    {1024,             10,     0x50000,        10280},
    {2048,             10,     0x60000,        20520}
    };
```

As with the **memArea** and **memSize** members in the **M_CL_CONFIG** structure, you can set these members of the **CL_DESC** structures by calculation after you create the table. The formula would be as follows:

```
clDescTbl[n].memSize =
    (clDescTbl[n].num * (clDescTbl[n].clusterSize + sizeof(long)));
```

The **memArea** member can point to a private memory area that you know to be available for storing clusters, or you can use ***malloc*( )**.

```
clDescTbl[n].memArea =  malloc( clDescTbl[n].memSize );
```

Valid cluster sizes range from 64 bytes to 65536 bytes. If there are multiple cluster pools, valid sizes are further restricted to powers of two (for example, 64, 128, 256, and so on). If there is only one cluster pool (as is often the case for the memory pool specific to a single device driver), there is no power of two restriction. Thus, the cluster can be of any size between 64 bytes and 65536 bytes on 4-byte alignment. A typical buffer size for Ethernet devices is 1514 bytes. However, because a cluster size requires a 4-byte alignment, the cluster size for this Ethernet buffer would have to be increased to at least 1516 bytes.

clDescTblNumEnt
Expects a count of the elements in the **CL_DESC** table referenced by the *pClDescTbl* parameter. This is a count of the number of cluster pools. You can get this value

using the NELEMENTS macro defined in **vxWorks.h**. For example:

```
int clDescTblNumEnt = (NELEMENTS(clDescTbl));
```

pFuncTbl
Expects a NULL or a pointer to a function table. This table contains pointers to the functions used to manage the buffers in this memory pool. Using a NULL for this parameter tells **netBufLib** to use its default function table. If you opt for the default function table, every **mBlk** and every cluster is prepended by a 4-byte header (which is why the size calculations above for clusters and **mBlk** structures contained an extra **sizeof(long)**). However, users need not concern themselves with this header when accessing these buffers. The returned pointers from functions such as *netClusterGet***( )** return pointers to the start of data, which is just after the header.

Assuming you have set up the configuration tables as shown above, a typical call to *netPoolInit***( )** would be as follows:

```
    int clDescTblNumEnt = (NELEMENTS(clDescTbl));
    NET_POOL  netPool;
    NET_POOL_ID       pNetPool = &netPool;
    if (netPoolInit (pNetPool, &mClBlkConfig, &clDescTbl [0],
clDescTblNumEnt,
        NULL) != OK)
        return (ERROR);
```

RETURNS OK or ERROR.

ERRNO
S_netBufLib_MEMSIZE_INVALID
S_netBufLib_CLSIZE_INVALID
S_netBufLib_NO_SYSTEM_MEMORY
S_netBufLib_MEM_UNALIGNED
S_netBufLib_MEMSIZE_UNALIGNED
S_netBufLib_MEMAREA_INVALID

SEE ALSO **netBufLib**, *netPoolDelete***( )**

*netPoolShow***( )**

NAME *netPoolShow***( )** – show pool statistics

SYNOPSIS

```
void netPoolShow
    (
    NET_POOL_ID pNetPool
    )
```

DESCRIPTION This routine displays the distribution of **mBlk**s and clusters in a given network pool ID.

RETURNS N/A

SEE ALSO **netShow**

netShowInit( )

NAME *netShowInit*( ) – initialize network show routines

SYNOPSIS

```
void netShowInit (void)
```

DESCRIPTION This routine links the network show facility into the VxWorks system. These routines are included automatically if **INCLUDE_NET_SHOW** is defined in **configAll.h**.

RETURNS N/A

SEE ALSO **netShow**

netStackDataPoolShow( )

NAME *netStackDataPoolShow*( ) – show network stack data pool statistics

SYNOPSIS

```
void netStackDataPoolShow (void)
```

DESCRIPTION This routine displays the distribution of **mBlk**s and clusters in a the network data pool. The network data pool is used only for data transfer through the network stack.

RETURNS N/A

SEE ALSO **netShow**, *netStackSysPoolShow*( ), **netBufLib**

netStackSysPoolShow( )

NAME *netStackSysPoolShow*( ) – show network stack system pool statistics

SYNOPSIS

```
void netStackSysPoolShow (void)
```

DESCRIPTION This routine displays the distribution of **mBlk**s and clusters in a the network system pool. The network system pool is used only for system structures such as sockets, routes, interface addresses, protocol control blocks, multicast addresses, and multicast route entries.

RETURNS N/A

SEE ALSO **netShow**, *netStackDataPoolShow*( ), **netBufLib**

netTask( )

NAME *netTask*( ) – network task entry point

SYNOPSIS

```
void netTask (void)
```

DESCRIPTION This routine is the VxWorks network support task. Most of the VxWorks network runs in this task's context.

NOTE To prevent an application task from monopolizing the CPU if it is in an infinite loop or is never blocked, the priority of ***netTask*( )** relative to an application may need to be adjusted. Network communication may be lost if ***netTask*( )** is "starved" of CPU time. The default task priority of ***netTask*( )** is 50. Use ***taskPrioritySet*( )** to change the priority of a task.

This task is spawned by ***netLibInit*( )**.

RETURNS N/A

SEE ALSO *netLibInit*( )

netTupleGet( )

NAME

netTupleGet( ) – get an **mBlk-clBlk**-cluster

SYNOPSIS

```
M_BLK_ID netTupleGet
    (
    NET_POOL_ID pNetPool, /* pointer to the net pool */
    int         bufSize,  /* size of the buffer to get */
    int         canWait,  /* wait or dontwait */
    UCHAR       type,     /* type of data */
    BOOL        bestFit   /* TRUE/FALSE */
    )
```

DESCRIPTION

This routine gets a **mBlk-clBlk**-cluster construct from the specified memory pool. Use this construct to pass data across the layers of the network stack.

pNetPool
: Expects a pointer to the memory pool from which you want to get a free **mBlk-clBlk**-cluster construct.

bufSize
: Expects the size, in bytes, of the cluster in the **clBlk**-cluster construct.

canWait
: Expects either **M_WAIT** or **M_DONTWAIT**. If *canWait* is **M_WAIT**, this routine blocks until an **mBlk-clBlk**-cluster construct is available. If *canWait* is **M_DONTWAIT** and no **mBlk-clBlk**-cluster construct is immediately available, this routine returns immediately (no blocking) with a NULL value.

type
: Expects the type of data. For example **MT_DATA**, **MT_HEADER**. The various values for this type are defined in **netBufLib.h**.

bestFit
: Expects either TRUE or FALSE. If TRUE and a cluster of the exact size is unavailable, this routine gets a larger cluster (if available). If *bestFit* is FALSE and an exact size cluster is unavailable, this routine gets either a smaller or a larger cluster (depending on what is available). Otherwise, it returns immediately with an ERROR value. For memory pools containing only one cluster size, *bestFit* should always be set to FALSE.

RETURNS

M_BLK_ID or NULL.

ERRNO

S_netBufLib_MBLK_INVALID
S_netBufLib_CLSIZE_INVALID
S_netBufLib_NETPOOL_INVALID

SEE ALSO

netBufLib

nfsAuthUnixGet( )

NAME

nfsAuthUnixGet( ) – get the NFS UNIX authentication parameters

SYNOPSIS

```
void nfsAuthUnixGet
    (
    char * machname, /* where to store host machine */
    int *  pUid,     /* where to store user ID */
    int *  pGid,     /* where to store group ID */
    int *  pNgids,   /* where to store number of group IDs */
    int *  gids      /* where to store array of group IDs */
    )
```

DESCRIPTION

This routine gets the previously set UNIX authentication values.

RETURNS

N/A

SEE ALSO

nfsLib, *nfsAuthUnixPrompt*( ), *nfsAuthUnixShow*( ), *nfsAuthUnixSet*( ), *nfsIdSet*( )

nfsAuthUnixPrompt( )

NAME

nfsAuthUnixPrompt( ) – modify the NFS UNIX authentication parameters

SYNOPSIS

```
void nfsAuthUnixPrompt (void)
```

DESCRIPTION

This routine allows UNIX authentication parameters to be changed from the shell. The user is prompted for each parameter, which can be changed by entering the new value next to the current one.

EXAMPLE

```
-> nfsAuthUnixPrompt
machine name:   yuba
user ID:        2001 128
group ID:       100
num of groups:  1 3
group #1:        100 100
group #2:        0 120
group #3:        0 200
value = 3 = 0x3
```

SEE ALSO

nfsLib, *nfsAuthUnixShow*( ), *nfsAuthUnixSet*( ), *nfsAuthUnixGet*( ), *nfsIdSet*( )

2

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**. **machname** should be set with the name of the mounted system (i.e. the target name itself) to distinguish hosts from hosts on a NFS network.

RETURNS N/A

SEE ALSO **nfsLib**, *nfsAuthUnixPrompt*( ), *nfsAuthUnixShow*( ), *nfsAuthUnixGet*( ), *nfsIdSet*( ), 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      /* no. of elements available in the list */
    )
```

DESCRIPTION
This routine fills the array *nfsDevlist* up to *listSize*, with handles to NFS devices currently in the system.

RETURNS
The number of entries filled in the *nfsDevList* array.

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. This is done automatically when the configuration macro **INCLUDE_NFSis** defined.

RETURNS OK, or ERROR if there is no room for the driver.

SEE ALSO **nfsDrv**

---

nfsDrvNumGet( )

NAME *nfsDrvNumGet*( ) – return the IO system driver number for the nfs driver

SYNOPSIS `int nfsDrvNumGet (void)`

DESCRIPTION This routine returns the nfs driver number allocated by iosDrvInstall during the nfs driver initialization. If the nfs driver has yet to be initialized, or if initialization failed, nfsDrvNumGet will return ERROR.

RETURNS the nfs driver number or ERROR

SEE ALSO **nfsDrv**

---

nfsdStatusGet( )

NAME *nfsdStatusGet*( ) – get the status of the NFS server

SYNOPSIS

```
STATUS nfsdStatusGet
    (
    NFS_SERVER_STATUS * serverStats /* pointer to status structure */
    )
```

DESCRIPTION This routine gets status information about the NFS server.

RETURNS OK, or ERROR if the information cannot be obtained.

SEE ALSO **nfsdLib**

---

nfsdStatusShow( )

NAME *nfsdStatusShow*( ) – show the status of the NFS server

SYNOPSIS

```
STATUS nfsdStatusShow
    (
    int options /* unused */
    )
```

DESCRIPTION This routine shows status information about the NFS server.

RETURNS OK, or ERROR if the information cannot be obtained.

SEE ALSO **nfsdLib**

nfsExport( )

NAME

nfsExport( ) – specify a file system to be NFS exported

SYNOPSIS

```
STATUS nfsExport
    (
    char * directory, /* Directory to export - FS must support NFS */
    int    id,        /* ID number for file system */
    BOOL   readOnly,  /* TRUE if file system is exported read-only */
    int    options    /* Reserved for future use - set to 0 */
    )
```

DESCRIPTION

This routine makes a file system available for mounting by a client. The client should be in the local host table (see ***hostAdd*( )**), although this is not required.

The *id* parameter can either be set to a specific value, or to 0. If it is set to 0, an ID number is assigned sequentially. Every time a file system is exported, it must have the same ID number, or clients currently mounting the file system will not be able to access files.

To display a list of exported file systems, use:

```
-> nfsExportShow "localhost"
```

RETURNS

OK, or ERROR if the file system could not be exported.

SEE ALSO

mountLib, **nfsLib**, *nfsExportShow*( ), *nfsUnexport*( )

nfsExportShow( )

NAME

nfsExportShow( ) – display the exported file systems of a remote host

SYNOPSIS

```
STATUS nfsExportShow
    (
    char * hostName /* host machine to show exports for */
    )
```

DESCRIPTION

This routine displays the file systems of a specified host and the groups that are allowed to mount them.

EXAMPLE

```
-> nfsExportShow "wrs"
/d0               staff
/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 */
                       /* 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.

ERRNO ENOENT

SEE ALSO **mountLib**, **nfsLib**, *nfsExportShow*( ), *nfsExport*( )

nfsUnmount( )

NAME *nfsUnmount*( ) – unmount an NFS device

SYNOPSIS

```
STATUS nfsUnmount
    (
    char * localName /* local of nfs device */
    )
```

DESCRIPTION This routine unmounts file systems that were previously mounted via NFS.

RETURNS OK, or ERROR if *localName* is not an NFS device or cannot be mounted.

SEE ALSO **nfsDrv**, *nfsMount*( )

nicEndLoad( )

NAME *nicEndLoad*( ) – initialize the driver and device

SYNOPSIS

```
END_OBJ* nicEvbEndLoad
    (
    char* initString /* string to be parse by the driver */
    )
```

DESCRIPTION This routine initializes the driver and the device to the operational state. All of the device-specific parameters are passed in *initString*, which expects a string of the following format:

unit:*base_addr*:*int_vector*:*int_level*

This routine can be called in two modes. If it is called with an empty but allocated string, it places the name of this device (that is, "ln") into the *initString* and returns 0.

If the string is allocated and not empty, the routine attempts to load the driver using the values specified in the string.

RETURNS An END object pointer, or NULL on error, or 0 and the name of the device if the *initString* was NULL.

SEE ALSO **nicEvbEnd**

nicEvbattach( )

NAME *nicEvbattach*( ) – publish and initialize the **nicEvb** network interface driver

SYNOPSIS

```
STATUS nicEvbattach
    (
    int          unit, /* unit number */
    NIC_DEVICE * pNic, /* address of NIC chip */
    int          ivec  /* interrupt vector to use */
    )
```

DESCRIPTION This routine publishes the **nicEvb** 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 if the receive buffer memory could not be allocated.

SEE ALSO **if_nicEvb**

nicEvbInitParse( )

NAME *nicEvbInitParse*( ) – parse the initialization string

SYNOPSIS

```
STATUS nicEvbInitParse
    (
    NICEVB_END_DEVICE * pDrvCtrl,
    char *              initString
    )
```

DESCRIPTION Parse the input string. Fill in values in the driver control structure. The initialization string format is: *unit*:*base_adrs*:*vecnum*:*intLvl*

unit
Device unit number, a small integer.

base_adrs
Base address for NIC device

vecNum
Interrupt vector number (used with ***sysIntConnect*( )**).

intLvl
Interrupt level.

RETURNS
OK, or ERROR if any arguments are invalid.

SEE ALSO
nicEvbEnd

nicTxStartup( )

NAME
nicTxStartup( ) – the driver's actual output routine

SYNOPSIS
```
#ifdef BSD43_DRIVER LOCAL STATUS nicTxStartup
    (
    int unit
    )
```

DESCRIPTION
This routine accepts outgoing packets from the if_snd queue, and then gains exclusive access to the DMA (through a mutex semaphore), then calls ***nicTransmit*( )** to send the packet out onto the interface.

RETURNS
OK, or ERROR if the packet could not be transmitted.

SEE ALSO
if_nicEvb

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 /* pointer to channel */
    )
```

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 /* pointer to channel */
    )
```

DESCRIPTION This routine handles four sources of interrupts from the UART. They are prioritized in the following order by the Interrupt Identification Register: Receiver Line Status, Received Data Ready, Transmit Holding Register Empty and Modem Status.

When a modem status interrupt occurs, the transmit interrupt is enabled if the CTS signal is TRUE.

RETURNS N/A

SEE ALSO **ns16550Sio**

ns16550IntEx( )

NAME *ns16550IntEx*( ) – miscellaneous interrupt processing

SYNOPSIS

```
void ns16550IntEx
    (
    NS16550_CHAN * pChan /* pointer to channel */
    )
```

DESCRIPTION This routine handles miscellaneous interrupts on the UART. Not implemented yet.

RETURNS N/A

SEE ALSO **ns16550Sio**

ns16550IntRd( )

NAME *ns16550IntRd*( ) – handle a receiver interrupt

SYNOPSIS

```
void ns16550IntRd
    (
    NS16550_CHAN * pChan /* pointer to channel */
    )
```

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 /* pointer to channel */
    )
```

DESCRIPTION This routine handles write interrupts from the UART. It reads a character and puts it in the transmit holding register of the device for transfer.

If there are no more characters to transmit, transmission is disabled by clearing the transmit interrupt enable bit in the IER(int enable register).

RETURNS N/A

SEE ALSO **ns16550Sio**

ntInt( )

NAME *ntInt*( ) – handle controller interrupt

SYNOPSIS

```
void ntInt
    (
    NTEND_DEVICE * pDrvCtrl
    )
```

DESCRIPTION This routine is called at interrupt level in response to an interrupt from the controller.

RETURNS N/A.

SEE ALSO **ntEnd**

ntLoad( )

NAME

ntLoad( ) – initialize the driver and device

SYNOPSIS

```
END_OBJ* ntLoad
    (
    char* initString, /* String to be parse by the driver. */
    void* nothing
    )
```

DESCRIPTION

This routine initializes the driver and the device to the operational state. All of the device specific parameters are passed in the initString.

The string contains the target specific parameters like this:

"unit:register addr:int vector:int level:shmem addr:shmem size:shmem width"

RETURNS

An END object pointer or NULL on error.

SEE ALSO

ntEnd

ntMemInit( )

NAME

ntMemInit( ) – initialize memory for the chip

SYNOPSIS

```
STATUS ntMemInit
    (
    NTEND_DEVICE * pDrvCtrl /* device to be initialized */
    )
```

DESCRIPTION

This routine is highly specific to the device.

RETURNS

OK or ERROR.

SEE ALSO

ntEnd

ntParse( )

NAME *ntParse*( ) – parse the init string

SYNOPSIS

```
STATUS ntParse
    (
    NTEND_DEVICE * pDrvCtrl,
    char *         initString
    )
```

DESCRIPTION Parse the input string. Fill in values in the driver control structure. The initialization string format is:
"unit:csrAdr:rapAdr:vecnum:intLvl:memAdrs:memSize:memWidth"

unit
Device unit number, a small integer.

vecNum
Interrupt vector number (used with ***sysIntConnect*( )**)

intLvl
Interrupt level (isn't really used)

RETURNS OK or ERROR for invalid arguments.

SEE ALSO **ntEnd**

ntPassFsDevInit( )

NAME ***ntPassFsDevInit*( )** – associate a device with ntPassFs file system functions

SYNOPSIS

```
void *ntPassFsDevInit
    (
    char * devName /* device name */
    )
```

DESCRIPTION This routine associates the name *devName* with the file system and installs it in the I/O System's device table. The driver number used when the device is added to the table is that which was assigned to the ntPassFs library during ***ntPassFsInit*( )**.

RETURNS A pointer to the volume descriptor, or NULL if there is an error.

SEE ALSO **ntPassFsLib**

ntPassFsInit( )

NAME

ntPassFsInit( ) – prepare to use the ntPassFs library

SYNOPSIS

```
STATUS ntPassFsInit
    (
    int nPassfs /* number of ntPass-through file systems */
    )
```

DESCRIPTION

This routine initializes the ntPassFs library. It must be called exactly once, before any other routines in the library. The argument specifies the number of ntPassFs devices that may be open at once. This routine installs **ntPassFsLib** as a driver in the I/O system driver table, allocates and sets up the necessary memory structures, and initializes semaphores.

Normally this routine is called from the root task, ***usrRoot*( )**, in ***usrConfig*( )**. To enable this initialization, define **INCLUDE_PASSFS** in **configAll.h**.

NOTE

Maximum number of ntPass-through file systems is 1.

RETURNS

OK, or ERROR.

SEE ALSO

ntPassFsLib

ntPollStart( )

NAME

ntPollStart( ) – start polled mode operations

SYNOPSIS

```
STATUS ntPollStart
    (
    NTEND_DEVICE* pDrvCtrl
    )
```

RETURNS

OK or ERROR.

SEE ALSO

ntEnd

ntPollStop( )

NAME *ntPollStop*( ) – stop polled mode operations

SYNOPSIS

```
STATUS ntPollStop
    (
    NTEND_DEVICE* pDrvCtrl
    )
```

DESCRIPTION This function terminates polled mode operation. The device returns to interrupt mode.

The device interrupts are enabled, the current mode flag is switched to indicate interrupt mode and the device is then reconfigured for interrupt operation.

RETURNS OK or ERROR.

SEE ALSO **ntEnd**

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*

*open***( )**

NAME

*open***( )** – open a file

SYNOPSIS

```
int open
    (
    const char * name,  /* name of the file to open */
    int          flags, /* O_RDONLY, O_WRONLY, O_RDWR, or O_CREAT */
    int          mode   /* mode of file to create (UNIX chmod style) */
    )
```

DESCRIPTION

This routine opens a file for reading, writing, or updating, and returns a file descriptor for that file. The arguments to *open***( )** are the filename and the type of access:

O_RDONLY (0)	(or READ)	- open for reading only.
O_WRONLY (1)	(or WRITE)	- open for writing only.
O_RDWR (2)	(or UPDATE)	- open for reading and writing.
O_CREAT (0x0200)		- create a file.

In general, *open***( )** can only open pre-existing devices and files. However, for NFS network devices only, files can also be created with *open***( )** by performing a logical OR operation with **O_CREAT** and the *flags* argument. In this case, the file is created with a UNIX chmod-style file mode, as indicated with *mode*. For example:

```
fd = open ("/usr/myFile", O_CREAT | O_RDWR, 0644);
```

Only the NFS driver uses the *mode* argument.

NOTE

For more information about situations when there are no file descriptors available, see the manual entry for *iosInit***( )**.

RETURNS

A file descriptor number, or ERROR if a file name is not specified, the device does not exist, no file descriptors are available, or the driver returns ERROR.

ERRNO

ELOOP

SEE ALSO

ioLib, *creat***( )**

opendir( )

NAME

opendir( ) – open a directory for searching (POSIX)

SYNOPSIS

```
DIR *opendir
    (
    char * dirName /* name of directory to open */
    )
```

DESCRIPTION

This routine opens the directory named by *dirName* and allocates a directory descriptor (DIR) for it. A pointer to the DIR structure is returned. The return of a NULL pointer indicates an error.

After the directory is opened, ***readdir*( )** is used to extract individual directory entries. Finally, ***closedir*( )** is used to close the directory.

WARNING

For remote file systems mounted over **netDrv**, *opendir*( ) fails, because the **netDrv** implementation strategy does not provide a way to distinguish directories from plain files. To permit use of ***opendir*( )** on remote files, use NFS rather than netDrv.

RETURNS

A pointer to a directory descriptor, or NULL if there is an error.

SEE ALSO

dirLib, *closedir*( ), *readdir*( ), *rewinddir*( ), *ls*( )

operator delete( )

NAME

operator delete( ) – default run-time support for memory deallocation (C++)

SYNOPSIS

```
extern void operator delete
    (
    void * pMem /* pointer to dynamically-allocated object */
    )
```

DESCRIPTION

This function provides the default implementation of operator delete. It returns the memory, previously allocated by operator new, to the VxWorks system memory partition.

RETURNS

N/A

SEE ALSO

cplusLib

operator new( )

NAME

operator new( ) – default run-time support for operator new (C++)

SYNOPSIS

```
extern void * operator new
    (
    size_t n /* size of object to allocate */
    ) throw (std::bad_alloc)
```

DESCRIPTION

This function provides the default implementation of operator new. It allocates memory from the system memory partition for the requested object. The value, when evaluated, is a pointer of the type **pointer-to-***T* where *T* is the type of the new object.

If allocation fails a new-handler, if one is defined, is called. If the new-handler returns, presumably after attempting to recover from the memory allocation failure, allocation is retried. If there is no new-handler an exception of type "bad_alloc" is thrown.

RETURNS

Pointer to new object.

THROWS

std::bad_alloc if allocation failed.

SEE ALSO

cplusLib

operator new( )

NAME

operator new( ) – default run-time support for operator new (nothrow) (C++)

SYNOPSIS

```
extern void * operator new
    (
    size_t          n, /* size of object to allocate */
    const nothrow_t &  /* supply argument of "nothrow" here */
    ) throw ()
```

DESCRIPTION

This function provides the default implementation of operator new (nothrow). It allocates memory from the system memory partition for the requested object. The value, when evaluated, is a pointer of the type **pointer-to-***T* where *T* is the type of the new object.

If allocation fails, a new-handler, if one is defined, is called. If the new-handler returns, presumably after attempting to recover from the memory allocation failure, allocation is retried. If the new_handler throws a bad_alloc exception, the exception is caught and 0 is returned. If allocation fails and there is no new_handler 0 is returned.

RETURNS Pointer to new object or 0 if allocation fails.

INCLUDE FILES **new**

SEE ALSO **cplusLib**

operator new( )

NAME *operator new*( ) – run-time support for operator new with placement (C++)

SYNOPSIS

```
extern void * operator new
    (
    size_t n,   /* size of object to allocate (unused) */
    void * pMem /* pointer to allocated memory */
    )
```

DESCRIPTION This function provides the default implementation of the global new operator, with support for the placement syntax. New-with-placement is used to initialize objects for which memory has already been allocated. *pMem* points to the previously allocated memory. memory.

RETURNS *pMem*

INCLUDE FILES **new**

SEE ALSO **cplusLib**

ospfExtRouteAdd( )

NAME *ospfExtRouteAdd*( ) – import external route into OSPF domain (OSPF Opt.)

SYNOPSIS

```
STATUS ospfExtRouteAdd
    (
    uint32_t destIp,       /* destination IP address */
    uint32_t destMask,     /* destination mask */
    uint32_t nextHopIp,    /* IP address of next hop */
    int      cost,         /* cost to advertise in domain */
    int      extRouteType, /* 1 = external type1, 2 = external type2 */
    int      tos           /* type of service */
    )
```

DESCRIPTION
This function is used to import an external route into the OSPF domain The destination address and mask are *destIp* and *destMask* respectively while *nextHopIp* is the IP address of the next hop. The cost to advertise in the OSPF domain is *cost* and route type is *routeType*, which can have the value 1 or 2 for type 1 and type 2 routes respectively. All IP addresses and masks in this call are in network byte order.

RETURNS
OK or ERROR.

SEE ALSO
ospfLib

ospfExtRouteDelete( )

NAME
ospfExtRouteDelete( ) – delete external route imported into OSPF (OSPF Opt.)

SYNOPSIS

```
STATUS ospfExtRouteDelete
    (
    uint32_t destIp,       /* destination IP address */
    uint32_t destMask,     /* destination mask */
    int      extRouteType, /* 1 = external type1, 2 = external type2 */
    int      tos           /* type of service */
    )
```

DESCRIPTION
This function is used to delete an external route imported into the OSPF domain. The destination address and mask are *destIp* and *destMask* respectively. The route type is *extRouteType* which may have the value 1 or 2 for type 1 and type 2 routes, respectively. All IP addresses and masks in this call are in network byte order.

RETURNS
OK or ERROR.

SEE ALSO
ospfLib

ospfInit( )

NAME
ospfInit( ) – function to initialize OSPF routing (OSPF Opt.)

SYNOPSIS

```
STATUS ospfInit
    (
    int     priority,  /* task priority */
    int     options,   /* task options */
    int     stackSize, /* task stack size */
```

```
    int      routerId,  /* routerId, host byte order */
    FUNCPTR pAuthHook  /* ospf authentication hook */
    )
```

DESCRIPTION

This function initializes the OSPF facilities. This includes creating OSPF tasks, which are created with a priority of *priority*, options set to *options*, a stack size of *stackSize*, and an OSPF router ID of *routerid*. If *routerId* is 0, the IP address of one of the interfaces is used as the router ID. The *pAuthHook* parameter expects a pointer to a user-provided authentication routine. For every received packet, the authentication function:

```
    (*ospfAuthHook) (pIfkey, pPktKey, ipAddr)
```

The *pIfkey* parameter is a pointer to the authorization key associated with the interface. The *pPktKey* parameter is a pointer to the key in the received packet. The *ipAddr* is the IP address in network byte order of the interface on which the packet was received. To set the interface authorization key, call ***m2OspfIfEntrySet*( )**. The ***ospfAuthHook*( )** routine returns TRUE if the packet is acceptable. Otherwise, it returns FALSE.

After this function has returned, you can use the m2Ospf****Set*( )** configuration routines to alter the settings.

RETURNS

OK or ERROR.

SEE ALSO

ospfLib

---

ospfNbmaDstAdd( )

NAME

***ospfNbmaDstAdd*( )** – add NBMA destination

SYNOPSIS

```
STATUS ospfNbmaDstAdd
    (
    uint32_t ipAddress,   /* neighbor IP address, network order */
    uint32_t ifIpAddress, /* local interface IP address, network order */
    BOOL     eligible     /* TRUE if neighbor is eligible to be DR */
    )
```

DESCRIPTION

On a non-broadcast multiple access network, a router capable of becoming designated router must be made aware of the IP addresses of all other routers on the network. The neighbor router is specified by its IP address *ipAddress*, the local interface IP address is *ifIpAddress* and *eligible* specifies if the neighbor is capable of acting as a designated router.

RETURNS

OK or ERROR.

SEE ALSO

ospfLib

ospfNbmaDstDelete( )

NAME

ospfNbmaDstDelete( ) – delete NBMA destination

SYNOPSIS

```
STATUS ospfNbmaDstDelete
    (
    uint32_t ipAddress,  /* neighbor IP address, network order */
    uint32_t ifIpAddress /* local interface IP address, network order */
    )
```

DESCRIPTION

Delete neighbor on a NBMA network, previously created with ***ospfNbmaDstAdd*( )**. The neighbor is specified by its IP address *ipAddress* and the local interface IP address is *ifIpAddress*.

RETURNS

OK or ERROR.

SEE ALSO

ospfLib

ospfTerminate( )

NAME

ospfTerminate( ) – free OSPF resources and delete OSPF tasks

SYNOPSIS

```
void ospfTerminate ()
```

DESCRIPTION

This function frees all the resources used by OSPF. This includes deleting the two VxWorks tasks used to manage OSPF. You are free to restart OSPF after this function has returned.

RETURNS

N/A

SEE ALSO

ospfLib

passFsDevInit( )

NAME *passFsDevInit*( ) – associate a device with passFs file system functions

SYNOPSIS

```
void *passFsDevInit
    (
    char * devName /* device name */
    )
```

DESCRIPTION This routine associates the name *devName* with the file system and installs it in the I/O System's device table. The driver number used when the device is added to the table is that which was assigned to the passFs library during ***passFsInit*( )**.

RETURNS A pointer to the volume descriptor, or NULL if there is an error.

SEE ALSO **passFsLib**

passFsInit( )

NAME *passFsInit*( ) – prepare to use the passFs library

SYNOPSIS

```
STATUS passFsInit
    (
    int nPassfs /* number of pass-through file systems */
    )
```

DESCRIPTION This routine initializes the passFs library. It must be called exactly once, before any other routines in the library. The argument specifies the number of passFs devices that may be open at once. This routine installs **passFsLib** as a driver in the I/O system driver table, allocates and sets up the necessary memory structures, and initializes semaphores.

Normally this routine is called from the root task, ***usrRoot*( )**, in ***usrConfig*( )**. This initialization is enabled when the configuration macro **INCLUDE_PASSFS** is defined.

NOTE Maximum number of pass-through file systems is 1.

RETURNS OK, or ERROR.

SEE ALSO **passFsLib**

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**

2

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**

pccardTffsEnabler( )

NAME *pccardTffsEnabler*( ) – enable the PCMCIA-TFFS driver

SYNOPSIS

```
STATUS pccardTffsEnabler
    (
    int             sock,          /* socket no. */
    TFFS_RESOURCE * pTffsResource, /* pointer to TFFS resources */
    int             numEnt,        /* number of SRAM resource entries */
    FUNCPTR         showRtn        /* TFFS show routine */
    )
```

DESCRIPTION This routine enables the PCMCIA-TFFS driver.

RETURNS OK, **ERROR_FIND** if there is no TFFS(Flash) 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*

pentiumBtc( )

NAME *pentiumBtc*( ) – execute atomic compare-and-exchange instruction to clear a bit

SYNOPSIS

```
STATUS pentiumBtc
    (
    )
 char * pFlag; /* flag address */
```

DESCRIPTION This routine compares a byte specified by the first parameter with TRUE. If it is TRUE, it changes it to 0 and returns OK. If it is not TRUE, it returns ERROR. LOCK and CMPXCHGB are used to get the atomic memory access.

RETURNS OK or ERROR if the specified flag is not TRUE

SEE ALSO **pentiumALib**

pentiumBts( )

NAME *pentiumBts*( ) – execute atomic compare-and-exchange instruction to set a bit

SYNOPSIS

```
STATUS pentiumBts
    (
    )
 char * pFlag; /* flag address */
```

DESCRIPTION This routine compares a byte specified by the first parameter with 0. If it is 0, it changes it to TRUE and returns OK. If it is not 0, it returns ERROR. LOCK and CMPXCHGB are used to get the atomic memory access.

RETURNS OK or ERROR if the specified flag is not zero

SEE ALSO **pentiumALib**

pentiumCr4Get( )

NAME *pentiumCr4Get*( ) – Get a content of CR4 register

SYNOPSIS

```
int pentiumCr4Get (void)
```

DESCRIPTION This routine gets a content of CR4 register.

RETURNS a content of CR4 register

SEE ALSO **pentiumALib**

pentiumCr4Set( )

NAME *pentiumCr4Set*( ) – Set a specified value to CR4 register

SYNOPSIS

```
void pentiumCr4Set
    (
    )
 int cr4; /* value to write CR4 register */
```

DESCRIPTION This routine sets a specified value to CR4 register.

RETURNS N/A

SEE ALSO **pentiumALib**

pentiumMcaShow( )

NAME *pentiumMcaShow*( ) – show MCA (Machine Check Architecture) registers

SYNOPSIS

```
void pentiumMcaShow (void)
```

DESCRIPTION This routine shows Machine-Check global control registers and Error-Reporting register banks. Number of the Error-Reporting register banks is kept in a variable mcaBanks. **MCi_ADDR** and **MCi_MISC** registers in the Error-Reporting register bank are showed if **MCi_STATUS** indicates that these registers are valid.

RETURNS N/A

SEE ALSO **pentiumShow**

pentiumMsrGet( )

NAME *pentiumMsrGet*( ) – get a content of the specified MSR (Model Specific Register)

SYNOPSIS

```
void pentiumMsrGet
    (
    )
 int addr; /* MSR address */
 long long int * pData; /* MSR data */
```

DESCRIPTION This routine gets a content of the specified MSR. The first parameter is an address of the MSR. The second parameter is a pointer of 64Bit variable.

RETURNS N/A

SEE ALSO **pentiumALib**

pentiumMsrSet( )

NAME *pentiumMsrSet*( ) – set a value to the specified MSR (Model Specific Registers)

SYNOPSIS

```
void pentiumMsrSet
    (
    )
 int addr; /* MSR address */
 long long int * pData; /* MSR data */
```

DESCRIPTION This routine sets a value to a specified MSR. The first parameter is an address of the MSR. The second parameter is a pointer of 64Bit variable.

RETURNS N/A

SEE ALSO **pentiumALib**

pentiumMtrrDisable( )

NAME *pentiumMtrrDisable*( ) – disable MTRR (Memory Type Range Register)

SYNOPSIS

```
void pentiumMtrrDisable (void)
```

DESCRIPTION This routine disables the MTRR that provide a mechanism for associating the memory types with physical address ranges in system memory.

RETURNS N/A

SEE ALSO **pentiumLib**

pentiumMtrrEnable( )

NAME *pentiumMtrrEnable*( ) – enable MTRR (Memory Type Range Register)

SYNOPSIS

```
void pentiumMtrrEnable (void)
```

DESCRIPTION This routine enables the MTRR that provide a mechanism for associating the memory types with physical address ranges in system memory.

RETURNS N/A

SEE ALSO **pentiumLib**

pentiumMtrrGet( )

NAME *pentiumMtrrGet*( ) – get MTRRs to a specified MTRR table

SYNOPSIS

```
STATUS pentiumMtrrGet
    (
    MTRR * pMtrr /* MTRR table */
    )
```

DESCRIPTION This routine gets MTRRs to a specified MTRR table with RDMSR instruction. The read MTRRs are CAP register, DEFTYPE register, fixed range MTRRs, and variable range MTRRs.

RETURNS OK, or ERROR if MTRR is being accessed.

SEE ALSO **pentiumLib**

pentiumMtrrSet( )

NAME *pentiumMtrrSet*( ) – set MTRRs from specified MTRR table with WRMSR instruction.

SYNOPSIS

```
STATUS pentiumMtrrSet
    (
    MTRR * pMtrr /* MTRR table */
    )
```

DESCRIPTION This routine sets MTRRs from specified MTRR table with WRMSR instruction. The written MTRRs are DEFTYPE register, fixed range MTRRs, and variable range MTRRs.

RETURNS OK, or ERROR if MTRR is enabled or being accessed.

SEE ALSO **pentiumLib**

pentiumPmcGet( )

NAME *pentiumPmcGet*( ) – get contents of PMC0 and PMC1

SYNOPSIS

```
void pentiumPmcGet
    (
    )
 long long int * pPmc0; /* Performance Monitoring Counter 0 */
 long long int * pPmc1; /* Performance Monitoring Counter 1 */
```

DESCRIPTION This routine gets contents of both PMC0 (Performance Monitoring Counter 0) and PMC1. The first parameter is a pointer of 64Bit variable to store the content of the Counter 0, and the second parameter is for the Counter 1.

RETURNS N/A

SEE ALSO **pentiumALib**

pentiumPmcGet0( )

NAME *pentiumPmcGet0*( ) – get a content of PMC0

SYNOPSIS

```
void pentiumPmcGet0
    (
    )
 long long int * pPmc0; /* Performance Monitoring Counter 0 */
```

DESCRIPTION This routine gets a content of PMC0 (Performance Monitoring Counter 0). Parameter is a pointer of 64Bit variable to store the content of the Counter.

RETURNS N/A

SEE ALSO **pentiumALib**

pentiumPmcGet1( )

NAME *pentiumPmcGet1*( ) – get a content of PMC1

SYNOPSIS

```
void pentiumPmcGet1
    (
    )
 long long int * pPmc1; /* Performance Monitoring Counter 1 */
```

DESCRIPTION This routine gets a content of PMC1 (Performance Monitoring Counter 1). Parameter is a pointer of 64Bit variable to store the content of the Counter.

RETURNS N/A

SEE ALSO **pentiumALib**

pentiumPmcReset( )

NAME *pentiumPmcReset*( ) – reset both PMC0 and PMC1

SYNOPSIS

```
void pentiumPmcReset (void)
```

DESCRIPTION This routine resets both PMC0 (Performance Monitoring Counter 0) and PMC1.

RETURNS N/A

SEE ALSO **pentiumALib**

pentiumPmcReset0( )

NAME *pentiumPmcReset0*( ) – reset PMC0

SYNOPSIS

```
void pentiumPmcReset0 (void)
```

DESCRIPTION This routine resets PMC0 (Performance Monitoring Counter 0).

RETURNS N/A

SEE ALSO **pentiumALib**

pentiumPmcReset1( )

NAME *pentiumPmcReset1*( ) – reset PMC1

SYNOPSIS

```
void pentiumPmcReset1 (void)
```

DESCRIPTION This routine resets PMC1 (Performance Monitoring Counter 1).

RETURNS N/A

SEE ALSO **pentiumALib**

pentiumPmcShow( )

NAME *pentiumPmcShow*( ) – show PMCs (Performance Monitoring Counters)

SYNOPSIS

```
void pentiumPmcShow
    (
    BOOL zap /* 1: reset PMC0 and PMC1 */
    )
```

DESCRIPTION This routine shows Performance Monitoring Counter 0 and 1. Monitored events are selected by Performance Event Select Registers in in pentiumPmcStart (). These counters are cleared to 0 if the parameter "zap" is TRUE.

RETURNS N/A

SEE ALSO **pentiumShow**

pentiumPmcStart( )

NAME *pentiumPmcStart*( ) – start both PMC0 and PMC1

SYNOPSIS

```
STATUS pentiumPmcStart
    (
    )
 int pmcEvtSel0; /* Performance Event Select Register 0 */
 int pmcEvtSel1; /* Performance Event Select Register 1 */
```

DESCRIPTION This routine starts both PMC0 (Performance Monitoring Counter 0) and PMC1 by writing specified events to Performance Event Select Registers. The first parameter is a content of Performance Event Select Register 0, and the second parameter is for the Performance Event Select Register 1.

RETURNS OK or ERROR if PMC is already started

SEE ALSO **pentiumALib**

pentiumPmcStop( )

NAME *pentiumPmcStop*( ) – stop both PMC0 and PMC1

SYNOPSIS

```
void pentiumPmcStop (void)
```

DESCRIPTION This routine stops both PMC0 (Performance Monitoring Counter 0) and PMC1 by clearing two Performance Event Select Registers.

RETURNS N/A

SEE ALSO **pentiumALib**

pentiumPmcStop1( )

NAME *pentiumPmcStop1*( ) – stop PMC1

SYNOPSIS

```
void pentiumPmcStop1 (void)
```

DESCRIPTION This routine stops only PMC1 (Performance Monitoring Counter 1) by clearing the Performance Event Select Register 1. Note, clearing the Performance Event Select Register 0 stops both counters, PMC0 and PMC1.

RETURNS N/A

SEE ALSO **pentiumALib**

pentiumSerialize( )

NAME *pentiumSerialize*( ) – execute a serializing instruction CPUID

SYNOPSIS

```
void pentiumSerialize (void)
```

DESCRIPTION This routine executes a serializing instruction CPUID. Serialization means that all modifications to flags, registers, and memory by previous instructions are completed before the next instruction is fetched and executed and all buffered writes have drained to memory.

RETURNS N/A

SEE ALSO **pentiumALib**

pentiumTlbFlush( )

NAME *pentiumTlbFlush*( ) – flush TLBs (Translation Lookaside Buffers)

SYNOPSIS

```
void pentiumTlbFlush (void)
```

DESCRIPTION This routine flushes TLBs by loading CR3 register. All of the TLBs are automatically invalidated any time the CR3 register loaded. The page global enable (PGE) flag in register CR4 and the global flag in a page-directory or page-table entry can be used to frequently used pages from being automatically invalidated in the TLBs on a load of CR3 register. The only way to deterministically invalidate global page entries is to clear the PGE flag and then invalidate the TLBs.

RETURNS N/A

SEE ALSO **pentiumALib**

pentiumTscGet32( )

NAME *pentiumTscGet32*( ) – get a lower half of the 64Bit TSC (Timestamp Counter)

SYNOPSIS

```
UINT32 pentiumTscGet32 (void)
```

DESCRIPTION This routine gets a lower half of the 64Bit TSC by RDTSC instruction. RDTSC instruction saves the lower 32Bit in EAX register, so this routine simply returns after executing RDTSC instruction.

RETURNS Lower half of the 64Bit TSC (Timestamp Counter)

SEE ALSO **pentiumALib**

pentiumTscGet64( )

NAME *pentiumTscGet64*( ) – get 64Bit TSC (Timestamp Counter)

SYNOPSIS

```
void pentiumTscGet64
    (
    )
 long long int * pTsc; /* Timestamp Counter */
```

DESCRIPTION This routine gets 64Bit TSC by RDTSC instruction. Parameter is a pointer of 64Bit variable to store the content of the Counter.

RETURNS N/A

SEE ALSO **pentiumALib**

pentiumTscReset( )

NAME *pentiumTscReset*( ) – reset the TSC (Timestamp Counter)

SYNOPSIS

```
void pentiumTscReset (void)
```

DESCRIPTION This routine resets the TSC by writing zero to the TSC with WRMSR instruction.

RETURNS N/A

SEE ALSO **pentiumALib**

period( )

NAME *period*( ) – spawn a task to call a function periodically

SYNOPSIS

```
int period
    (
    int     secs, /* period in seconds */
    FUNCPTR func, /* function to call repeatedly */
    int     arg1, /* first of eight args to pass to func */
```

```
    int     arg2,
    int     arg3,
    int     arg4,
    int     arg5,
    int     arg6,
    int     arg7,
    int     arg8
    )
```

DESCRIPTION This command spawns a task that repeatedly calls a specified function, with up to eight of its arguments, delaying the specified number of seconds between calls.

For example, to have ***i*( )** display task information every 5 seconds, just type:

```
-> period 5, i
```

NOTE The task is spawned using the ***sp*( )** routine. See the description of ***sp*( )** for details about priority, options, stack size, and task ID.

RETURNS A task ID, or ERROR if the task cannot be spawned.

SEE ALSO **usrLib**, ***periodRun*( )**, ***sp*( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

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.

NOTE

The following global variables can be set from the target shell or Windsh to configure the ***ping*( )** parameters:

_pingTxLen
: Size of the ICMP echo packet (default 64).

_pingTxInterval
: Packet interval in seconds (default 1 second).

_pingTxTmo
: Packet timeout in seconds (default 5 seconds).

RETURNS

OK, or ERROR if the remote host is not reachable.

ERRNO

EINVAL, **S_pingLib_NOT_INITIALIZED**, **S_pingLib_TIMEOUT**

SEE ALSO

pingLib

---

pingLibInit( )

NAME

***pingLibInit*( )** – initialize the ***ping*( )** utility

SYNOPSIS

```
STATUS pingLibInit (void)
```

DESCRIPTION

This routine allocates resources used by the ***ping*( )** utility. It must be called before ***ping*( )** is used. It is called automatically when the configuration macro **INCLUDE_PING** is defined.

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 the configuration macro **INCLUDE_PIPES** is defined.

RETURNS OK, or ERROR if the driver installation fails.

SEE ALSO **pipeDrv**

pow( )

NAME *pow*( ) – compute the value of a number raised to a specified power (ANSI)

SYNOPSIS

```
double pow
    (
    double x, /* operand */
    double y  /* exponent */
    )
```

DESCRIPTION This routine returns x to the power of y in double precision (IEEE double, 53 bits).

A domain error occurs if *x* is negative and *y* is not an integral value. A domain error occurs if the result cannot be represented when *x* is zero and *y* is less than or equal to zero. A range error may occur.

INCLUDE FILES **math.h**

RETURNS The double-precision value of *x* to the power of *y*.

Special cases:

(anything) ** 0	is	1
(anything) ** 1	is	itself
(anything) ** NaN	is	NaN
NaN ** (anything except 0)	is	NaN
+-(anything> 1) ** +INF	is	+INF
+-(anything> 1) ** -INF	is	+0
+-(anything \< 1) ** +INF	is	+0
+-(anything \< 1) ** -INF	is	+INF
+-1 ** +-INF	is	NaN, signal INVALID
+0 ** +(anything non-0, NaN)	is	+0
-0 ** +(anything non-0, NaN, odd int)	is	+0
+0 ** -(anything non-0, NaN)	is	+INF, signal DIV-BY-ZERO
-0 ** -(anything non-0, NaN, odd int)	is	+INF with signal
-0 ** (odd integer)	=	-(+0 ** (odd integer))
+INF ** +(anything except 0, NaN)	is	+INF
+INF ** -(anything except 0, NaN)	is	+0
-INF ** (odd integer)	=	-(+INF ** (odd integer))
-INF ** (even integer)	=	(+INF ** (even integer))
-INF ** -(any non-integer, NaN)	is	NaN with signal
-(x=anything) ** (k=integer)	is	(-1)**k * (x ** k)
-(anything except 0) ** (non-integer)	is	NaN with signal

SEE ALSO **ansiMath**, **mathALib**

powf( )

NAME *powf*( ) – compute the value of a number raised to a specified power (ANSI)

SYNOPSIS

```
float powf
    (
    float x, /* operand */
```

```
    float y  /* exponent */
    )
```

DESCRIPTION This routine returns the value of *x* to the power of *y* in single precision.

INCLUDE FILES **math.h**

RETURNS The single-precision value of *x* to the power of *y*.

SEE ALSO **mathALib**

---

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*( )**

2

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.

-
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 */
    )
```

DESCRIPTION This routine writes a formatted string to standard error. Its function and syntax are otherwise identical to *printf*( ).

RETURNS The number of characters output, or ERROR if there is an error during output.

SEE ALSO **fioLib**, *printf*( )

printErrno( )

NAME *printErrno*( ) – print the definition of a specified error status value

SYNOPSIS

```
void printErrno
    (
    int errNo /* status code whose name is to be printed */
    )
```

DESCRIPTION This command displays the error-status string, corresponding to a specified error-status value. It is only useful if the error-status symbol table has been built and included in the system. If *errNo* is zero, then the current task status is used by calling ***errnoGet*( )**.

This facility is described in **errnoLib**.

RETURNS N/A

SEE ALSO **usrLib**, **errnoLib**, ***errnoGet*( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

printf( )

NAME *printf*( ) – write a formatted string to the standard output stream (ANSI)

SYNOPSIS

```
int printf
    (
    const char * fmt /* format string to write */
    )
```

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 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 **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 the port, *port*, to be forwarded to and from the proxy network. To enable all ports, specify zero for *port*.

RETURNS OK, or ERROR if unsuccessful.

SEE ALSO **proxyArpLib**

proxyPortShow( )

NAME *proxyPortShow*( ) – show 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*

---

psrShow( )

NAME *psrShow*( ) – display the meaning of a specified PSR value, symbolically (ARM)

SYNOPSIS

```
STATUS psrShow
    (
    UINT32 psrval /* psr value to show */
    )
```

DESCRIPTION This routine displays the meaning of all fields in a specified PSR value, symbolically.

RETURNS OK, always.

SEE ALSO **dbgArchLib**

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 is copied to a private buffer.

RETURNS
OK, or ERROR if space cannot be malloc'd.

SEE ALSO
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

r0( )

NAME

r0( ) – return the contents of register **r0** (also **r1** – **r14**) (ARM)

SYNOPSIS

```
int r0
    (
    int taskId /* task ID, 0 means default task */
    )
```

DESCRIPTION

This command extracts the contents of register **r0** from the TCB of a specified task. If *taskId* is omitted or zero, the last task referenced is assumed.

Similar routines are provided for registers (**r1** – **r14**): ***r1( )*** – ***r14( )***.

RETURNS The contents of register **r0** (or the requested register).

SEE ALSO **dbgArchLib**, *VxWorks Programmer's Guide: Debugging*

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.

This initialization is enabled when the configuration macro **INCLUDE_RAWFS** is defined; *rawFsInit*( ) is then 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 */
                       /* 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.

For BSD 4.4 sockets a copy of the **mp>msg_iov** array will be made. This requires a cluster from the network stack system pool of **size mp>msg_iovlen * sizeof (struct iovec)** or 8 bytes.

RETURNS

The number of bytes received, or ERROR if the call fails.

SEE ALSO

sockLib

---

reld( )

NAME

***reld*( )** – reload an object module

SYNOPSIS

```
MODULE_ID reld
    (
    void * nameOrId, /* name or ID of the object module file */
    int    options   /* options, 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 Only certain devices support ***rename*( )**. To confirm that your device supports it, consult the respective **xxDrv** or xxFs listings to verify that ioctl **FIORENAME** exists. For example, dosFs and rt11Fs support ***rename*( )**, but **netDrv** and **nfsDrv** do not.

RETURNS OK, or ERROR if the file could not be opened or renamed.

SEE ALSO **ioLib**

repeat( )

NAME

repeat( ) – spawn a task to call a function repeatedly

SYNOPSIS

```
int repeat
    (
    int     n,    /* no. of times to call func (0=forever) */
    FUNCPTR func, /* function to call repeatedly */
    int     arg1, /* first of eight args to pass to func */
    int     arg2,
    int     arg3,
    int     arg4,
    int     arg5,
    int     arg6,
    int     arg7,
    int     arg8
    )
```

DESCRIPTION

This command spawns a task that calls a specified function *n* times, with up to eight of its arguments. If *n* is 0, the routine is called endlessly, or until the spawned task is deleted.

NOTE

The task is spawned using ***sp*( )**. See the description of ***sp*( )** for details about priority, options, stack size, and task ID.

RETURNS

A task ID, or ERROR if the task cannot be spawned.

SEE ALSO

usrLib, ***repeatRun*( )**, ***sp*( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

repeatRun( )

NAME

repeatRun( ) – call a function repeatedly

SYNOPSIS

```
void repeatRun
    (
    int     n,    /* no. of times to call func (0=forever) */
    FUNCPTR func, /* function to call repeatedly */
    int     arg1, /* first of eight args to pass to func */
    int     arg2,
    int     arg3,
```

```
    int     arg4,
    int     arg5,
    int     arg6,
    int     arg7,
    int     arg8
    )
```

DESCRIPTION This command calls a specified function *n* times, with up to eight of its arguments. If *n* is 0, the routine is called endlessly.

Normally, this routine is called only by ***repeat*( )**, which spawns it as a task.

RETURNS N/A

SEE ALSO **usrLib**, ***repeat*( )**, *VxWorks Programmer's Guide: Target Shell*

---

resolvDNComp( )

NAME ***resolvDNComp*( )** – compress a DNS name in a DNS packet

SYNOPSIS

```
int resolvDNComp
    (
    const u_char * exp_dn,   /* ptr to the expanded domain name */
    u_char *       comp_dn,  /* ptr to where to output the compressed name */
    int            length,   /* length of the buffer pointed by comp_dn */
    u_char * *     dnptrs,   /* ptr to a ptr list of compressed names */
    u_char * *     lastdnptr /* ptr to the last entry pointed by dnptrs */
    )
```

DESCRIPTION This routine takes the expanded domain name referenced in the *exp_dn* parameter, compresses it, and stores the compressed name in the location pointed to by the *comp_dn* parameter. The *length* parameter passes in the length of the buffer starting at *comp_dn*. The *dnptrs* parameter is a pointer to a list of pointers to previously compressed names. The *lastdnptr* parameter points to the last entry in the *dnptrs* array.

RETURNS The size of the compressed name, or ERROR.

SEE ALSO **resolvLib**, ***resolvGetHostByName*( )**, ***resolvGetHostByAddr*( )**, ***resolvDNExpand*( )**, ***resolvInit*( )**, ***resolvSend*( )**, ***resolvParamsSet*( )**, ***resolvParamsGet*( )**, ***resolvMkQuery*( )**, ***resolvQuery*( )**

resolvDNExpand( )

NAME

resolvDNExpand( ) – expand a DNS compressed name from a DNS packet

SYNOPSIS

```
int resolvDNExpand
    (
    const u_char * msg,     /* ptr to the start of the DNS packet */
    const u_char * eomorig, /* ptr to last location +1 of the DNS packet */
    const u_char * comp_dn, /* ptr to the compressed domain name */
    u_char *       exp_dn,  /* ptr to where the expanded DN is output */
    int            length   /* length of the buffer pointed by expd_dn */
    )
```

DESCRIPTION

This functions expands a compressed DNS name from a DNS packet. The *msg*parameter points to that start of the DNS packet. The *eomorig* parameter points to the last location of the DNS packet plus 1. The *comp_dn* parameter points to the compress domain name, and *exp_dn* parameter expects a pointer to a buffer. Upon function completion, this buffer contains the expanded domain name. Use the *length* parameter to pass in the size of the buffer referenced by the *exp_dn* parameter.

RETURNS

The length of the expanded domain name, or ERROR on failure.

SEE ALSO

resolvLib, *resolvGetHostByName*( ), *resolvGetHostByAddr*( ), *resolvInit*( ), *r esolvDNComp*( ), *resolvSend*( ), *resolvParamsSet*( ), *resolvParamsGet*( ), *resolvMkQuery*( ), *resolvQuery*( )

resolvGetHostByAddr( )

NAME

resolvGetHostByAddr( ) – query the DNS server for the host name of an IP address

SYNOPSIS

```
struct hostent * resolvGetHostByAddr
    (
    const char * pInetAddr,
    char *       pHostBuf,
    int          bufLen
    )
```

DESCRIPTION

This function returns a **hostent** structure, which is defined as follows:

```
struct   hostent
    {
```

```
char *   h_name;            /* official name of host */
char **  h_aliases;         /* alias list */
int      h_addrtype;        /* address type */
int      h_length;          /* length of address */
char **  h_addr_list;       /* list of addresses from name server */
unsigned int h_ttl;         /* Time to Live in Seconds for this entry */
}
```

The **h_aliases** and **h_addr_type** vectors are NULL-terminated.

The *pinetAddr* parameter passes in the IP address (in network byte order) for the host whose name you want to discover. The *pBuf* and *bufLen* parameters specify the location and size (512 bytes or more) of the buffer that is to receive the hostent structure. *resolvGetHostByAddr*( ) returns host addresses are returned in network byte order.

RETURNS

A pointer to a **hostent** structure if the host is found, or NULL if the parameters are invalid, host is not found, or the buffer is too small.

ERRNO

S_resolvLib_INVALID_PARAMETER
S_resolvLib_BUFFER_2_SMALL
S_resolvLib_TRY_AGAIN
S_resolvLib_HOST_NOT_FOUND
S_resolvLib_NO_DATA
S_resolvLib_NO_RECOVERY

SEE ALSO

resolvLib, *resolvGetHostByName*( ), *resolvInit*( ), *resolvDNExpand*( ), *resolvDNComp*( ), *resolvSend*( ), *resolvParamsSet*( ), *resolvParamsGet*( ), *resolvMkQuery*( ), *resolvQuery*( )

---

resolvGetHostByName( )

NAME

resolvGetHostByName( ) – query the DNS server for the IP address of a host

SYNOPSIS

```
struct hostent * resolvGetHostByName
    (
    char * pHostName, /* ptr to the name of the host */
    char * pHostBuf,  /* ptr to the buffer used by hostent structure */
    int    bufLen     /* length of the buffer */
    )
```

DESCRIPTION

This function returns a **hostent** structure. This structure is defined as follows:

```
struct   hostent
{
char *   h_name;           /* official name of host */
```

```
    char **  h_aliases;       /* alias list */
    int      h_addrtype;      /* address type */
    int      h_length;        /* length of address */
    char **  h_addr_list;     /* list of addresses from name server */
    unsigned int h_ttl;       /* Time to Live in Seconds for this entry */
    }
```

The **h_aliases** and **h_addr_type** vectors are NULL-terminated.

Specify the host you want to query in *pHostname*. Use *pBuf* and *bufLen* to specify the location and size of a buffer to receive the **hostent** structure and its associated contents. Host addresses are returned in network byte order. Given the information this routine retrieves, the *pBuf* buffer should be 512 bytes or larger.

RETURNS

A pointer to a **hostent** structure if the host is found, or NULL if the parameters are invalid, the host is not found, or the buffer is too small.

ERRNO

S_resolvLib_INVALID_PARAMETER
S_resolvLib_BUFFER_2_SMALL
S_resolvLib_TRY_AGAIN
S_resolvLib_HOST_NOT_FOUND
S_resolvLib_NO_DATA
S_resolvLib_NO_RECOVERY

SEE ALSO

resolvLib, ***resolvInit*( )**, ***resolvGetHostByAddr*( )**, ***resolvDNExpand*( )**, ***resolvDNComp*( )**, ***resolvSend*( )**, ***resolvParamsSet*( )**, ***resolvParamsGet*( )**, ***resolvMkQuery*( )**, ***resolvQuery*( )**

---

resolvInit( )

NAME

***resolvInit*( )** – initialize the resolver library

SYNOPSIS

```
STATUS resolvInit
    (
    char *  pNameServer,        /* pointer to Name server IP address */
    char *  pDefaultDomainName, /* default domain name */
    FUNCPTR pdnsDebugRtn        /* function ptr to debug routine */
    )
```

DESCRIPTION

This function initializes the resolver. *pNameServer* is a single IP address for a name server in dotted decimal notation. *pDefaultDomainName* is the default domain name to be appended to names without a dot. The function pointer *pdnsDebugRtn* is set to the resolver debug function. Additional name servers can be configured using the function ***resolvParamsSet*( )**.

RETURNS OK or ERROR.

SEE ALSO **resolvLib**, *resolvGetHostByName*( ), *resolvGetHostByAddr*( ), *resolvDNExpand*( ), *resolvDNComp*( ), *resolvSend*( ), *resolvParamsSet*( ), *resolvParamsGet*( ), *resolvQuery*( )

---

resolvMkQuery( )

NAME *resolvMkQuery*( ) – create all types of DNS queries

SYNOPSIS

```
int resolvMkQuery
    (
    int          op,       /* set to desire query QUERY or IQUERY */
    const char * dname,    /* domain name to be use in the query */
    int          class,    /* query class for IP is C_IN */
    int          type,     /* type is T_A, T_PTR, ... */
    const char * data,     /* resource Record (RR) data */
    int          datalen,  /* length of the RR */
    const char * newrr_in, /* not used always set to NULL */
    char *       buf,      /* out of the constructed query */
    int          buflen    /* length of the buffer for the query */
    )
```

DESCRIPTION This routine uses the input parameters to create a domain name query. You can set the *op* parameter to QUERY or IQUERY. Specify the domain name in *dname*, the class in *class*, the query type in *type*. Valid values for type include **T_A**, **T_PTR**, and so on. Use *data* to add Resource Record data to the query. Use *datalen* to pass in the length of the data buffer. Set *newrr_in* to NULL. This parameter is reserved for future use. The *buf* parameter expects a pointer to the output buffer for the constructed query. Use *buflen* to pass in the length of the buffer referenced in *buf*.

RETURNS The length of the constructed query or ERROR.

SEE ALSO **resolvLib**, *resolvGetHostByName*( ), *resolvGetHostByAddr*( ), *resolvDNExpand*( ), *resolvDNComp*( ), *resolvSend*( ), *resolvParamsSet*( ), *resolvParamsGet*( ), *resolvInit*( ), *resolvQuery*( )

resolvParamsGet( )

NAME

resolvParamsGet( ) – get the parameters which control the resolver library

SYNOPSIS

```
void resolvParamsGet
    (
    RESOLV_PARAMS_S * pResolvParams /* ptr to resolver parameter struct */
    )
```

DESCRIPTION

This routine copies the resolver parameters to the **RESOLV_PARAMS_S** structure referenced in the *pResolvParms* parameter. The **RESOLV_PARAMS_S** structure is defined in **resolvLib.h** as follows:

```
typedef struct
   {
   char   queryOrder;
   char   domainName [MAXDNAME];
   char   nameServersAddr [MAXNS][MAXIPADDRLEN];
   } RESOLV_PARAMS_S;
```

Typically, you call this function just before calling *resolvParamsSet*( ). The *resolvParamsGet*( ) call populates the **RESOLV_PARAMS_S** structure. You can then modify the default values just before calling *resolvParamsSet*( ).

RETURNS

N/A

SEE ALSO

resolvLib, *resolvGetHostByName*( ), *resolvGetHostByAddr*( ), *resolvDNExpand*( ), *resolvDNComp*( ), *resolvSend*( ), *resolvParamsSet*( ), *resolvInit*( ), *resolvMkQuery*( ), *resolvQuery*( )

resolvParamsSet( )

NAME

resolvParamsSet( ) – set the parameters which control the resolver library

SYNOPSIS

```
STATUS resolvParamsSet
    (
    RESOLV_PARAMS_S * pResolvParams /* ptr to resolver parameter struct */
    )
```

DESCRIPTION

This routine sets the resolver parameters. *pResolvParams* passes in a pointer to a **RESOLV_PARAMS_S** structure, which is defined as follows:

```
typedef struct
   {
   char   queryOrder;
   char   domainName [MAXDNAME];
   char   nameServersAddr [MAXNS][MAXIPADDRLEN];
   } RESOLV_PARAMS_S;
```

Use the members of this structure to specify the settings you want to apply to the resolver. It is important to remember that multiple tasks can use the resolver library and that the settings specified in this **RESOLV_PARAMS_S** structure affect all queries from all tasks. In addition, you should set resolver parameters at initialization and not while queries could be in progress. Otherwise, the results of the query are unpredictable.

Before calling *resolvParamsSet*( ), you should first call *resolvParamsGet*( ) to populate a **RESOLV_PARAMS_S** structure with the current settings. Then you change the values of the members that interest you.

Valid values for the **queryOrder** member of **RESOLV_PARAMS_S** structure are defined in **resolvLib.h**. Set the **domainName** member to the domain to which this resolver belongs. Set the **nameServersAddr** member to the IP addresses of the DNS server that the resolver can query. You must specify the IP addresses in standard dotted decimal notation. This function tries to validate the values in the **queryOrder** and **nameServerAddr** members. This function does not try to validate the domain name.

RETURNS OK if the parameters are valid, ERROR otherwise.

SEE ALSO **resolvLib**, *resolvGetHostByName*( ), *resolvGetHostByAddr*( ), *resolvDNExpand*( ), *resolvDNComp*( ), *resolvSend*( ), *resolvInit*( ), *resolvParamsGet*( ), *resolvMkQuery*( ), *resolvQuery*( )

resolvQuery( )

NAME *resolvQuery*( ) – construct a query, send it, wait for a response

SYNOPSIS

```
int resolvQuery
    (
    char *   name,   /* domain name */
    int      class,  /* query class for IP is C_IN */
    int      type,   /* type is T_A, T_PTR, ... */
    u_char * answer, /* buffer to put answer */
    int      anslen  /* length of answer buffer */
    )
```

DESCRIPTION This routine constructs a query for the domain specified in the *name* parameter. The *class* parameter specifies the class of the query. The *type* parameter specifies the type of query. The routine then sends the query to the DNS server. When the server responds, the response is validated and copied to the buffer you supplied in the *answer* parameter. Use the *anslen* parameter to pass in the size of the buffer referenced in *answer*.

RETURNS The length of the response or ERROR.

ERRNO
S_resolvLib_TRY_AGAIN
S_resolvLib_HOST_NOT_FOUND
S_resolvLib_NO_DATA
S_resolvLib_NO_RECOVERY

SEE ALSO **resolvLib**, ***resolvGetHostByName*( )**, ***resolvGetHostByAddr*( )**, ***resolvDNExpand*( )**, ***resolvDNComp*( )**, ***resolvInit*( )**, ***resolvParamsSet*( )**, ***resolvParamsGet*( )**, ***resolvMkQuery*( )**

---

resolvSend( )

NAME ***resolvSend*( )** – send a pre-formatted query and return the answer

SYNOPSIS

```
int resolvSend
    (
    const char * buf,    /* pre-formatted query */
    int          buflen, /* length of query */
    char *       answer, /* buffer for answer */
    int          anslen  /* length of answer */
    )
```

DESCRIPTION This routine takes a pre-formatted DNS query and sends it to the domain server. Use *buf* to pass in a pointer to the query. Use *buflen* to pass in the size of the buffer referenced in *buf*. The *answer* parameter expects a pointer to a buffer into which this routine can write the answer retrieved from the server. Use *anslen* to pass in the size of the buffer you have provided in *anslen*.

RETURNS The length of the response or ERROR.

ERRNO
S_resolvLib_TRY_AGAIN
ECONNREFUSE
ETIMEDOU

SEE ALSO **resolvLib**, *resolvGetHostByName***( )**, *resolvGetHostByAddr***( )**, *resolvDNExpand***( )**, *resolvDNComp***( )**, *resolvInit***( )**, *resolvParamsSet***( )**, *resolvParamsGet***( )**, *resolvMkQuery***( )**, *resolvQuery***( )**

*rewind***( )**

NAME *rewind***( )** – set the file position indicator to the beginning of a file (ANSI)

SYNOPSIS

```
void rewind
    (
    FILE * fp /* stream */
    )
```

DESCRIPTION This routine sets the file position indicator for a specified stream to the beginning of the file.

It is equivalent to:

```
    (void) fseek (fp, 0L, SEEK_SET);
```

except that the error indicator for the stream is cleared.

INCLUDE FILES **stdio.h**

RETURNS N/A

SEE ALSO **ansiStdio**, *fseek***( )**, *ftell***( )**

*rewinddir***( )**

NAME *rewinddir***( )** – reset position to the start of a directory (POSIX)

SYNOPSIS

```
void rewinddir
    (
    DIR * pDir /* pointer to directory descriptor */
    )
```

DESCRIPTION This routine resets the position pointer in a directory descriptor (DIR). The *pDir* parameter is the directory descriptor pointer that was returned by ***opendir*( )**.

As a result, the next ***readdir*( )** will cause the current directory data to be read in again, as if an ***opendir*( )** had just been performed. Any changes in the directory that have occurred since the initial ***opendir*( )** will now be visible. The first entry in the directory will be returned by the next ***readdir*( )**.

RETURNS N/A

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

---

rindex( )

NAME ***rindex*( )** – find the last occurrence of a character in a string

SYNOPSIS

```
char *rindex
    (
    const char * s, /* string in which to find character */
    int          c  /* character to find in string */
    )
```

DESCRIPTION This routine finds the last occurrence of character *c*in string *s*.

RETURNS A pointer to *c*, or NULL if *c* is not found.

SEE ALSO **bLib**

---

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*

ripAuthHook( )

NAME *ripAuthHook*( ) – sample authentication hook

SYNOPSIS

```
STATUS ripAuthHook
    (
    char *    pKey, /* rip2IfConfAuthKey entry from MIB-II family */
    RIP_PKT * pRip  /* received RIP message */
    )
```

DESCRIPTION This hook demonstrates one possible authentication mechanism. It rejects all RIP-2 messages which used simple password authentication since they did not match the key contained in the MIB variable. All other RIP-2 messages are also rejected since no other authentication type is supported and all RIP-1 messages are also rejected, as recommended by the RFC specification. This behavior is the same as if no hook were installed.

RETURNS OK if message is acceptable, or ERROR otherwise.

ERRNO N/A

SEE ALSO **ripLib**

ripAuthHookAdd( )

NAME *ripAuthHookAdd*( ) – add an authentication hook to a RIP interface

SYNOPSIS

```
STATUS ripAuthHookAdd
    (
    char*   pIpAddr,  /* IP address in dotted decimal notation */
    FUNCPTR pAuthHook /* routine to handle message authentication */
    )
```

DESCRIPTION This routine installs a hook routine to validate incoming RIP messages for a registered interface given by *pIpAddr*. (Interfaces created or changed after a RIP session has started may be installed/updated with the ***ripIfSearch*( )** and ***ripIfReset*( )** routines). The hook is only called if an SNMP agent enables authentication for the corresponding interface. It uses the following prototype:

```
STATUS ripAuthHookRtn (char *pKey, RIP_PKT *pRip);
```

The first argument contains the authentication key for the message stored in the rip2IfConfAuthKey MIB variable and the second argument uses the **RIP_PKT** structure (defined in **rip/ripLib.h**) to access the message body. The routine must return OK if the message is acceptable, or ERROR otherwise. All RIP-2 messages sent to that routine already contain an authentication entry, but have not been verified. (Any unauthenticated RIP-2 messages have already been discarded as required by the RFC specification). RIP-1 messages may be accepted or rejected. RIP-2 messages requesting simple password authentication which match the key are accepted automatically before the hook is called. The remaining RIP-2 messages either did not match that key or are using an unknown authentication type. If any messages are rejected, the MIB-II counters are updated appropriately outside of the hook routine.

The current RIP implementation contains a sample authentication hook which may be added as follows:

```
if (ripAuthHookAdd ("90.0.0.1", ripAuthHook) == ERROR)
      logMsg ("Unable to add authorization hook.\n", 0, 0, 0, 0, 0, 0);
```

The sample routine only supports simple password authentication against the key included in the MIB variable. Since all such messages have already been accepted, all RIP-2 messages received by the routine are discarded. All RIP-1 messages are also discarded, so the hook actually has no effect. The body of that routine is:

```
STATUS ripAuthHook
   (
   char *     pKey,   /* rip2IfConfAuthKey entry from MIB-II family */
   RIP_PKT *  pRip    /* received RIP message */
   )
   {
   if (pRip->rip_vers == 1)
       {
       /*
        @ The RFC specification recommends, but does not require, rejecting
        @ version 1 packets when authentication is enabled.
        */
       return (ERROR);
       }
   /*
    @ The authentication type field in the RIP message corresponds to
    @ the first two bytes of the sa_data field overlayed on that
    @ message by the sockaddr structure contained within the RIP_PKT
    @ structure (see rip/ripLib.h).
    */
   if ( (pRip->rip_nets[0].rip_dst.sa_data[0] != 0) ||
```

```
        (pRip->rip_nets[0].rip_dst.sa_data[1] !=
        M2_rip2IfConfAuthType_simplePassword))
        {
        /* Unrecognized authentication type. */
        return (ERROR);
        }
    /*
     @ Discard version 2 packets requesting simple password authentication
     @ which did not match the MIB variable.
     */
    return (ERROR);
    }
```

A comparison against a different key could be performed as follows:

```
bzero ( (char *)&key, AUTHKEYLEN);    /* AUTHKEYLEN from rip/m2RipLib.h */
 /*
  @ The start of the authorization key corresponds to the third byte
  @ of the sa_data field in the sockaddr structure overlayed on the
  @ body of the RIP message by the RIP_PKT structure. It continues
  @ for the final 14 bytes of that structure and the first two bytes
  @ of the following rip_metric field.
  */
bcopy ( (char *)(pRip->rip_nets[0].rip_dst.sa_data + 2),
       (char *)&key, AUTHKEYLEN);
if (bcmp ( (char *)key, privateKey, AUTHKEYLEN) != 0)
    {
    /* Key does not match: reject message. */
    return (ERROR);
    }
return (OK);
```

The ***ripAuthHookDelete*( )** routine will remove the installed function. If authentication is still enabled for the interface, all incoming messages which do not use simple password authentication will be rejected until a routine is provided.

RETURNS
OK if hook added, or ERROR otherwise.

ERRNO
S_m2Lib_INVALID_PARAMETER
S_m2Lib_ENTRY_NOT_FOUND

SEE ALSO
ripLib

ripAuthHookDelete( )

NAME

ripAuthHookDelete( ) – remove an authentication hook from a RIP interface

SYNOPSIS

```
STATUS ripAuthHookDelete
    (
    char* pIpAddr /* IP address in dotted decimal notation */
    )
```

DESCRIPTION

This routine removes an assigned authentication hook from a registered interface indicated by *pIpAddr*. (Interfaces created or changed after a RIP session has started may be installed/updated with the ***ripIfSearch*( )** and ***ripIfReset*( )** routines). If authentication is still enabled for the interface, RIP-2 messages using simple password authentication will be accepted if they match the key in the MIB variable, but all other incoming messages will be rejected until a routine is provided.

RETURNS

OK, or ERROR if the interface could not be found.

ERRNO

S_m2Lib_INVALID_PARAMETER
S_m2Lib_ENTRY_NOT_FOUND

SEE ALSO

ripLib

ripDebugLevelSet( )

NAME

ripDebugLevelSet( ) – specify amount of debugging output

SYNOPSIS

```
void ripDebugLevelSet
    (
    int level /* verbosity level (0 - 3) */
    )
```

DESCRIPTION

This routine influences the amount of debugging information sent to standard output during the RIP session. Higher values of the *level*parameter result in increasingly verbose output. A *level* of zero restores the default behavior by disabling all debugging output.

RETURNS

N/A

ERRNO

N/A

SEE ALSO

ripLib

ripFilterDisable( )

NAME *ripFilterDisable*( ) – prevent strict border gateway filtering

SYNOPSIS

```
void ripFilterDisable (void)
```

DESCRIPTION This routine configures an active RIP session to ignore the restrictions necessary for RIP-1 and RIP-2 routers to operate correctly in the same network. All border gateway filtering is ignored and all routes to subnets, supernets, and specific hosts will be sent over any available interface. This operation is only correct if no RIP-1 routers are present anywhere on the network. Results are unpredictable if that condition is not met, but high rates of packet loss and widespread routing failures are likely.

The border gateway filtering rules are in force by default.

RETURNS N/A

ERRNO N/A

SEE ALSO **ripLib**

ripFilterEnable( )

NAME *ripFilterEnable*( ) – activate strict border gateway filtering

SYNOPSIS

```
void ripFilterEnable (void)
```

DESCRIPTION This routine configures an active RIP session to enforce the restrictions necessary for RIP-1 and RIP-2 routers to operate correctly in the same network as described in section 3.2 of RFC 1058 and section 3.3 of RFC 1723. When enabled, routes to portions of a logical network (including host routes) will be limited to routers within that network. Updates sent outside that network will only include a single entry representing the entire network. That entry will subsume all subnets and host-specific routes. If supernets are used, the entry will advertise the largest class-based portion of the supernet reachable through the connected interface.

RETURNS N/A

ERRNO N/A

SEE ALSO **ripLib**

ripIfReset( )

NAME

ripIfReset( ) – alter the RIP configuration after an interface changes

SYNOPSIS

```
STATUS ripIfReset
    (
    char * pIfName /* name of changed interface */
    )
```

DESCRIPTION

This routine updates the interface list and routing tables to reflect address and/or netmask changes for the device indicated by *pIfName*. To accommodate possible changes in the network number, all routes using the named interface are removed from the routing tables, but will be added in the next route update if appropriate. None of the removed routes are poisoned, so it may take some time for the routing tables of all the RIP participants to stabilize if the network number has changed.

RETURNS

OK, or ERROR if named interface not found or not added to list.

ERRNO

N/A

SEE ALSO

ripLib

ripIfSearch( )

NAME

ripIfSearch( ) – add new interfaces to the internal list

SYNOPSIS

```
void ripIfSearch (void)
```

DESCRIPTION

By default, a RIP session will not recognize any interfaces initialized after it has started. This routine schedules a search for additional interfaces which will occur during the next update of the internal routing table. Once completed, the session will accept and send RIP messages over the new interfaces.

RETURNS

N/A

ERRNO

N/A

SEE ALSO

ripLib

2

ripLeakHookAdd( )

NAME *ripLeakHookAdd*( ) – add a hook to bypass the RIP and kernel routing tables

SYNOPSIS

```
STATUS ripLeakHookAdd
    (
    char *  pIpAddr,  /* IP address in dotted decimal notation */
    FUNCPTR pLeakHook /* function pointer to hook */
    )
```

DESCRIPTION This routine installs a hook routine to support alternative routing protocols for the registered interface given by *pIpAddr*. (Interfaces created or changed after a RIP session has started may be installed/updated with the ***ripIfSearch*( )** and ***ripIfReset*( )** routines).

The hook uses the following interface:

```
STATUS ripLeakHookRtn (long dest, long gateway, long netmask)
```

The RIP session will not add the given route to any tables if the hook routine returns OK, but will create a route entry otherwise.

The ***ripLeakHookDelete*( )** will allow the RIP session to add new routes unconditionally.

RETURNS OK, or ERROR if the interface could not be found.

ERRNO **S_m2Lib_INVALID_PARAMETER**
S_m2Lib_ENTRY_NOT_FOUND

SEE ALSO **ripLib**

ripLeakHookDelete( )

NAME *ripLeakHookDelete*( ) – remove a table bypass hook from a RIP interface

SYNOPSIS

```
STATUS ripLeakHookDelete
    (
    char* pIpAddr /* IP address in dotted decimal notation */
    )
```

DESCRIPTION This routine removes the assigned bypass hook from a registered interface indicated by *pIpAddr*. (Interfaces created or changed after a RIP session has started may be installed/updated with the ***ripIfSearch*( )** and ***ripIfReset*( )** routines). The RIP session will return to the default behavior and add entries to the internal RIP table and kernel routing table unconditionally.

RETURNS OK, or ERROR if the interface could not be found.

ERRNO **S_m2Lib_INVALID_PARAMETER**
S_m2Lib_ENTRY_NOT_FOUND

SEE ALSO **ripLib**

ripLibInit( )

NAME *ripLibInit*( ) – initialize the RIP routing library

SYNOPSIS

```
STATUS ripLibInit
    (
    BOOL supplier,       /* operate in silent mode? */
    BOOL gateway,        /* act as gateway to the Internet? */
    BOOL multicast,      /* use multicast or broadcast addresses? */
    int  version,        /* 1 or 2: selects format of outgoing messages */
    int  timerRate,      /* update frequency for internal routing table */
    int  supplyInterval, /* update frequency for neighboring routers */
    int  expire,         /* maximum interval for renewing learned routes */
    int  garbage         /* elapsed time before deleting stale route */
    )
```

DESCRIPTION This routine creates and initializes the global data structures used by the RIP routing library and starts a RIP session to maintain routing tables for a host. It must be called before using any other library routines, and is invoked automatically if **INCLUDE_RIP** is defined at the time the system is built.

The resulting RIP session will monitor all network interfaces which are currently available for messages from other RIP routers. If the *supplier*parameter is true, it will also respond to specific requests from other routers and transmit route updates over every known interface at the interval specified by *supplyInterval*.

Specifying a *gateway* setting of true establishes this router as a gateway to the wider Internet, capable of routing packets anywhere within the local networks. The final *multicast* flag indicates whether the RIP messages are sent to the pre-defined multicast address of 224.0.0.9 (which requires a *version* setting of 2) or to the broadcast address of the interfaces.

The *version* parameter determines the format used for outgoing RIP messages, and also sets the initial settings of the MIB-II compatibility switches in combination with the *multicast* flag. A *version* of 1 will restrict all incoming traffic to that older message type. A *version* of 2 will set the receive switch to accept either type unless *multicast* is true, which limits reception to version 2 messages only. SNMP agents may alter those settings on a per-interface basis once startup is complete.

The remaining parameters set various system timers used to maintain the routing table. All of the values are expressed in seconds, and must be greater than or equal to 1. The *timerRate* determines how often the routing table is examined for changes and expired routes. The *supplyInterval* must be an exact multiple of that value. The *expire* parameter specifies the maximum time between updates before a route is invalidated and removed from the kernel table. Expired routes are then deleted from the internal RIP routing table if no update has been received within the time set by the *garbage* parameter.

The defaults for all the parameter settings are given by the following constants. The default timer values match the settings indicated in the RFC specification.

Parameter Name	Default Value	Symbolic Constant
supplier	0 (FALSE)	**RIP_SUPPLIER**
gateway	0 (FALSE)	**RIP_GATEWAY**
multicast	0 (FALSE)	**RIP_EXPIRE_TIME**
version	1	**RIP_SUPPLY_INTERVAL**
timerRate	1	**RIP_TIMER_RATE**
supplyInterval	30	**RIP_SUPPLY_INTERVAL**
expire	180	**RIP_EXPIRE_TIME**
garbage	300	**RIP_GARBAGE_TIME**

RETURNS
OK, or ERROR if configuration fails.

ERRNO
N/A

SEE ALSO
ripLib

ripRouteShow( )

NAME
***ripRouteShow*( )** – display the internal routing table maintained by RIP

SYNOPSIS

```
void ripRouteShow()
```

DESCRIPTION
This routine prints every entry in the local RIP routing table. The flags displayed below the destination, gateway, and netmask addresses indicate the current route status. Entries with the **RTS_INTERFACE** flag indicate routes to directly connected networks which are

generated locally. If **RTS_SUBNET** is set for an entry, it is subject to border gateway filtering (if enabled). When **RTS_INTERNAL** is also present, the corresponding entry is an "artificial" route created to supply distant networks with legitimate destinations if border filtering excludes the actual entry. Those entries are not copied to the kernel routing table. The **RTS_CHANGED** flag marks entries added or modified in the last timer interval which will be included in a triggered update.

RETURNS N/A

ERRNO N/A

SEE ALSO **ripLib**

---

ripSendHookAdd( )

NAME *ripSendHookAdd*( ) – add an update filter to a RIP interface

SYNOPSIS

```
STATUS ripSendHookAdd
    (
    char* pIpAddr,      /* IP address in dotted decimal notation */
    BOOL (*ripSendHook) (struct rt_entry* pRt) /* Routine to use. */
    )
```

DESCRIPTION This routine installs a hook routine to screen individual route entries for inclusion in a periodic update. The routine is installed for the registered interface given by *pIpAddr*. (Interfaces created or changed after a RIP session has started may be installed/updated with the ***ripIfSearch*( )** and ***ripIfReset*( )** routines).

The hook uses the following prototype:

```
BOOL ripSendHookRtn (struct rt_entry* pRt);
```

If the hook returns FALSE, the route is not included in the update. Otherwise, it is included if it meets the other restrictions, such as simple split horizon and border gateway filtering. The ***ripSendHookDelete*( )** routine removes this additional filter from the output processing.

RETURNS OK, or ERROR if the interface could not be found.

ERRNO **S_m2Lib_INVALID_PARAMETER**
S_m2Lib_ENTRY_NOT_FOUND

SEE ALSO **ripLib**

ripSendHookDelete( )

NAME

ripSendHookDelete( ) – remove an update filter from a RIP interface

SYNOPSIS

```
STATUS ripSendHookDelete
    (
    char* pIpAddr /* IP address in dotted decimal notation */
    )
```

DESCRIPTION

This routine removes the hook routine that allowed additional screening of route entries in periodic updates from the registered interface indicated by *pIpAddr*. (Interfaces created or changed after a RIP session has started may be installed/updated with the ***ripIfSearch*( )** and ***ripIfReset*( )** routines). The RIP session will return to the default behavior and include any entries which meet the other restrictions (such as simple split horizon).

RETURNS

OK, or ERROR if the interface could not be found.

ERRNO

S_m2Lib_INVALID_PARAMETER
S_m2Lib_ENTRY_NOT_FOUND

SEE ALSO

ripLib

ripShutdown( )

NAME

ripShutdown( ) – terminate all RIP processing

SYNOPSIS

```
STATUS ripShutdown (void)
```

DESCRIPTION

This routine "poisons" all routes in the current table by transmitting updates with an infinite metric for each entry over all available interfaces. It then halts all RIP processing and removes the associated tasks and data structures. When completed successfully, the RIP services are unavailable until restarted with the ***ripLibInit*( )** routine.

RETURNS

OK if shutdown completed, or ERROR otherwise.

ERRNO

N/A

SEE ALSO

ripLib

rlogin( )

NAME

rlogin( ) – log in to a remote host

SYNOPSIS

```
STATUS rlogin
    (
    char * host /* name of host to connect to */
    )
```

DESCRIPTION

This routine allows users to log in to a remote host. It may be called from the VxWorks shell as follows:

```
-> rlogin "remoteSystem"
```

where *remoteSystem* is either a host name, which has been previously added to the remote host table by a call to ***hostAdd*( )**, or an Internet address in dot notation (e.g., "90.0.0.2"). The remote system will be logged into with the current user name as set by a call to ***iam*( )**.

The user disconnects from the remote system by typing:

```
~.
```

as the only characters on the line, or by simply logging out from the remote system using ***logout*( )**.

RETURNS

OK, or ERROR if the host is unknown, no privileged ports are available, the routine is unable to connect to the host, or the child process cannot be spawned.

SEE ALSO

rlogLib, *iam*( ), *logout*( )

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 the configuration macro **INCLUDE_RLOGIN** is defined, ***rlogind*( )** is spawned by ***rlogInit*( )** at boot time.

Remote login requests will cause **stdin**, **stdout**, and **stderr** to be directed away from the console. When the remote user disconnects, **stdin**, **stdout**, and **stderr** are restored, and the shell is restarted. The ***rlogind*( )** routine uses the remote user verification protocol specified by the UNIX remote shell daemon documentation, but ignores all the information except the user name, which is used to set the VxWorks remote identity (see the manual entry for ***iam*( )**).

The remote login daemon requires the existence of a pseudo-terminal device, which is created by ***rlogInit*( )** before ***rlogind*( )** is spawned. The ***rlogind*( )** routine creates two child processes, **tRlogInTask** and **tRlogOutTask**, whenever a remote user is logged in. These processes exit when the remote connection is terminated.

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 the configuratiion macro **INCLUDE_RLOGIN** is defined, ***rlogInit*( )** 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).

You can specify both *destination* and *gateway* in standard Internet address format (for example, 90.0.0.2), or you can specify them using their host names, as specified with ***hostAdd*( )**.

EXAMPLE

Consider the following example:

```
-> routeAdd "90.0.0.0", "gate"
```

This call tells VxWorks that the machine with the host name "gate" is the gateway to network 90.0.0.0. The host "gate" must already have been created by ***hostAdd*( )**.

Consider the following example:

```
-> routeAdd "90.0.0.0", "91.0.0.3"
```

This call tells VxWorks that the machine with the Internet address 91.0.0.3 is the gateway to network 90.0.0.0.

Consider the following example:

```
-> routeAdd "destination", "gate"
```

This call tells VxWorks that the machine with the host name "gate" is the gateway to the machine named "destination". The host names "gate" and "destination" must already have been created by ***hostAdd*( )**.

Consider the following example:

```
-> routeAdd "0", "gate"
```

This call tells VxWorks that the machine with the host name "gate" is the default gateway. The host "gate" must already have been created by ***hostAdd*( )**. A default gateway is where Internet Protocol (IP) datagrams are routed when there is no specific routing table entry available for the destination IP network or host.

RETURNS OK or ERROR.

SEE ALSO **routeLib**

routeDelete( )

NAME *routeDelete*( ) – delete a route

SYNOPSIS

```
STATUS routeDelete
    (
    char * destination, /* inet addr or name of route destination */
    char * gateway      /* inet addr or name of gateway to destination */
    )
```

DESCRIPTION This routine deletes a specified route from the network routing tables.

RETURNS OK or ERROR.

SEE ALSO **routeLib**, *routeAdd*( )

2

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

routeProtoPrioritySet( )

NAME

routeProtoPrioritySet( ) – set the priority of routes added by the routing protocol

SYNOPSIS

```
STATUS routeProtoPrioritySet
    (
    int proto, /* protocol no, from m2Lib.h */
    int prio   /* priority, >= 0 , <= 200 */
    )
```

DESCRIPTION

This routine assigns a priority to a routing protocol. A route generated by the *proto* protocol is added to the routing table only if a protocol of higher priority does not already have that route installed in the table. Use *proto* to identify the protocol. See **m2Lib.h** for a listing of valid values for *proto*. Use *prio* to specify the priority level you want to assign to *proto*. The *prio* parameter may be any integer value greater or equal to 0 and less than or equal to 200. The higher values indicate higher priority. If you do not want VxWorks to prioritize protocols, do not call this routine.

Routes that are added with the ***routeAdd*( )** or ***mRouteAdd*( )** call are of type **M2_ipRouteProto_other**. These are static routes that are not affected by routing protocols such as RIP and OSPF. To change the priority of routes added in this way pass the value **M2_ipRoute_Proto_other** in the first argument of this routine.

RETURNS OK if priority set successfully else 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 (that is, "up")
0x2	– destination is a gateway
0x4	– host specific routing entry
0x8	– host or net unreachable
0x10	– created dynamically (by redirect)
0x20	– modified dynamically (by redirect)
0x40	– message confirmed
0x80	– subnet mask present
0x100	– generate new routes on use
0x200	– external daemon resolves name
0x400	– generated by ARP
0x800	– manually added (static)
0x1000	– just discard packets (during updates)
0x2000	– modified by management protocol
0x4000	– protocol specific routing flag
0x8000	– protocol specific routing flag

In the above display example, the entry in the ROUTE NET TABLE has a flag value of 1, which indicates that this route is "up" and usable and network specific (the 0x4 bit is turned off). The entry in the ROUTE HOST TABLE has a flag value of 5 (0x1 OR'ed with 0x4), which indicates that this route is "up" and usable and host-specific.

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 the configuration macro **INCLUDE_RPC** is defined.

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** when the configuration macro **INCLUDE_RT11FS** is defined.

RETURNS
OK, or ERROR if memory is insufficient.

SEE ALSO
rt11FsLib

2

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 [task[,addr[,addr1]]]
```

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

CAVEAT

When a task is continued, ***s( )*** does not distinguish between a 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*

---

sa1100DevInit( )

NAME

***sa1100DevInit*( )** – initialise an SA1100 channel

SYNOPSIS

```
void sa1100DevInit
    (
    SA1100_CHAN * pChan /* ptr to SA1100_CHAN describing this channel */
    )
```

DESCRIPTION

This routine initialises some **SIO_CHAN** function pointers and then resets the chip to a quiescent state. Before this routine is called, the BSP must already have initialised all the device addresses, etc. in the **SA1100_CHAN** structure.

RETURNS

N/A

SEE ALSO

sa1100Sio

sa1100Int( )

NAME

sa1100Int( ) – handle an interrupt

SYNOPSIS

```
void sa1100Int
    (
    SA1100_CHAN * pChan /* ptr to SA1100_CHAN describing this channel */
    )
```

DESCRIPTION

This routine handles interrupts from the UART.

RETURNS

N/A

SEE ALSO

sa1100Sio

saIoWrite( )

NAME

saIoWrite( ) – send a packet to the master agent's message queue

SYNOPSIS

```
STATUS saIoWrite
    (
    PTR_T       ipchandle, /* Subagent's identifier */
    EBUFFER_T * pBuf,      /* Encoded buffer */
    INT_32_T    code       /* Message type */
    )
```

DESCRIPTION

This routine is called either from *snmpSaHandlerAsync*( ) or from the registration routines. *ipchandle* contains an identifier to the sub agents's message queue except for the case when the message is a response to **IPC_AYT**. In this case, it contains the identifier to the local queue at the master agent. The *pBuf* parameter points to the message being sent. The *code* parameter takes a value that indicates how the master agent should process the message. Value *code* values are **CALL_QUERY_HANDLER**, **CALL_REG_HANDLER**, and **IPC_AYT**. For more on how these values influence message processing in the master agent, see the description of *snmpMonitorSpawn*( ).

RETURNS

OK or ERROR.

SEE ALSO

saIoLib

saIpcFree( )

NAME *saIpcFree*( ) – free the specified IPC mechanism

SYNOPSIS

```
void saIpcFree
    (
    PTR_T ipchandle /* pointer to IPC handle */
    )
```

DESCRIPTION Call this routine to free the IPC mechanism specified by *ipchandle*. You created this IPC mechanism with a call to ***snmpSaInit*( )**. If you rewrote ***snmpSaInit*( )** to use an IPC mechanism other than message queues, you must rewrite ***saIpcFree*( )** to match.

RETURNS N/A

SEE ALSO **saIoLib**

saMsgBuild( )

NAME *saMsgBuild*( ) – build and encode a message and send it to the master agent

SYNOPSIS

```
void saMsgBuild
    (
    VBL_T *       vblist,  /* pointer to varbind list */
    SA_HEADER_T * hdr_blk, /* pointer to header block */
    SA_DEMUX_T *  demuxer, /* pointer to demuxer */
    PTR_T         saId     /* IPC handle */
    )
```

DESCRIPTION The ***hdrBlkCreate*( )** routine calls ***saMsgBuild*( )** to build a message, encode it, and transmit it to the master agent. The message is built up from the information provided in the input parameters:

vblist
: Expects a pointer to the **VBL_T** structure containing the varbind list you want to include in the message.

hdr_blk
: Expects a pointer to the header for this message.

demuxer
: Expects a pointer to the demuxer information for this message. The demuxer

information consists of a string and an object ID. In a message dealing with a version 1 request, the string is the community string and the object ID is unused. In a message dealing with a version 2 request, the string is the local entity string from the context and the object ID is the local time object ID from the context.

said
: Expects a pointer to the IPC mechanism (a message queue ID) that the master agent can use to respond to this message.

To encode the message, this routine calls ***snmpSubEncode*( )**. To send the message to the master agent, this routine calls ***saIoWrite*( )**.

RETURNS
N/A

SEE ALSO
saIoLib

scanf( )

NAME
scanf( ) – read and convert characters from the standard input stream (ANSI)

SYNOPSIS

```
int scanf
    (
    char const * fmt /* format string */
    )
```

DESCRIPTION
This routine reads input from the standard input stream under the control of the string *fmt*. It is equivalent to ***fscanf*( )** with an *fp* argument of **stdin**.

INCLUDE FILES
stdio.h

RETURNS
The number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure; or EOF if an input failure occurs before any conversion.

SEE ALSO
ansiStdio, ***fscanf*( )**, ***sscanf*( )**

sched_get_priority_max( )

NAME *sched_get_priority_max*( ) – get the maximum priority (POSIX)

SYNOPSIS

```
int sched_get_priority_max
    (
    int policy /* scheduling policy */
    )
```

DESCRIPTION This routine returns the value of the highest possible task priority for a specified scheduling policy (**SCHED_FIFO** or **SCHED_RR**).

NOTE If the global variable **posixPriorityNumbering** is FALSE, the VxWorks native priority numbering scheme is used, in which higher priorities are indicated by smaller numbers. This is different than the priority numbering scheme specified by POSIX, in which higher priorities are indicated by larger numbers.

RETURNS Maximum priority value, or -1 (ERROR) on error.

ERRNO **EINVAL**
– invalid scheduling policy.

SEE ALSO **schedPxLib**

sched_get_priority_min( )

NAME *sched_get_priority_min*( ) – get the minimum priority (POSIX)

SYNOPSIS

```
int sched_get_priority_min
    (
    int policy /* scheduling policy */
    )
```

DESCRIPTION This routine returns the value of the lowest possible task priority for a specified scheduling policy (**SCHED_FIFO** or **SCHED_RR**).

NOTE If the global variable **posixPriorityNumbering** is FALSE, the VxWorks native priority numbering scheme is used, in which higher priorities are indicated by smaller numbers. This is different than the priority numbering scheme specified by POSIX, in which higher priorities are indicated by larger numbers.

RETURNS Minimum priority value, or -1 (ERROR) on error.

ERRNO **EINVAL**
– invalid scheduling policy.

SEE ALSO **schedPxLib**

sched_getparam( )

NAME *sched_getparam*( ) – get the scheduling parameters for a specified task (POSIX)

SYNOPSIS

```
int sched_getparam
    (
    pid_t                tid,  /* task ID */
    struct sched_param * param /* scheduling param to store priority */
    )
```

DESCRIPTION This routine gets the scheduling priority for a specified task, *tid*. If *tid* is 0, it gets the priority of the calling task. The task's priority is copied to the **sched_param** structure pointed to by *param*.

NOTE If the global variable **posixPriorityNumbering** is FALSE, the VxWorks native priority numbering scheme is used, in which higher priorities are indicated by smaller numbers. This is different than the priority numbering scheme specified by POSIX, in which higher priorities are indicated by larger numbers.

RETURNS 0 (OK) if successful, or -1 (ERROR) on error.

ERRNO **ESRCH**
– invalid task ID.

SEE ALSO **schedPxLib**

2

sched_getscheduler( )

NAME *sched_getscheduler*( ) – get the current scheduling policy (POSIX)

SYNOPSIS

```
int sched_getscheduler
    (
    pid_t tid /* task ID */
    )
```

DESCRIPTION This routine returns the currents scheduling policy (i.e., **SCHED_FIFO** or **SCHED_RR**).

RETURNS Current scheduling policy (**SCHED_FIFO** or **SCHED_RR**), or -1 (ERROR) on error.

ERRNO
ESRCH
– invalid task ID.

SEE ALSO **schedPxLib**

sched_rr_get_interval( )

NAME *sched_rr_get_interval*( ) – get the current time slice (POSIX)

SYNOPSIS

```
int sched_rr_get_interval
    (
    pid_t             tid,     /* task ID */
    struct timespec * interval /* struct to store time slice */
    )
```

DESCRIPTION This routine sets *interval* to the current time slice period if round-robin scheduling is currently enabled.

RETURNS 0 (OK) if successful, -1 (ERROR) on error.

ERRNO
EINVAL
– round-robin scheduling is not currently enabled.
ESRCH
– invalid task ID.

SEE ALSO **schedPxLib**

sched_setparam( )

NAME

sched_setparam( ) – set a task's priority (POSIX)

SYNOPSIS

```
int sched_setparam
    (
    pid_t                      tid,  /* task ID */
    const struct sched_param * param /* scheduling parameter */
    )
```

DESCRIPTION

This routine sets the priority of a specified task, *tid*. If *tid* is 0, it sets the priority of the calling task. Valid priority numbers are 0 through 255.

The *param* argument is a structure whose member **sched_priority** is the integer priority value. For example, the following program fragment sets the calling task's priority to 13 using POSIX interfaces:

```
#include "sched.h"
 ...
struct sched_param AppSchedPrio;
 ...
AppSchedPrio.sched_priority = 13;
if ( sched_setparam (0, &AppSchedPrio) != OK )
    {
    ... /* recovery attempt or abort message */
    }
 ...
```

NOTE

If the global variable **posixPriorityNumbering** is FALSE, the VxWorks native priority numbering scheme is used, in which higher priorities are indicated by smaller numbers. This is different than the priority numbering scheme specified by POSIX, in which higher priorities are indicated by larger numbers.

RETURNS

0 (OK) if successful, or -1 (ERROR) on error.

ERRNO

EINVAL
– scheduling priority is outside valid range.
ESRCH
– task ID is invalid.

SEE ALSO

schedPxLib

2

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**

2

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 un-filled part of the buffer to be lost when the buffer is invalidated after the command.

To optimize the performance of the driver in supporting different caching policies, the routine uses the **CACHE_USER_FLUSH** macro when flushing the cache. In the absence of a **CACHE_USER_CLEAR** macro, the following steps are taken:

1. If there is a non-NULL flush routine in the **cacheUserFuncs** structure, the cache is cleared.
2. If there is a non-NULL invalidate routine, the cache is invalidated.
3. Otherwise nothing is done; the cache is assumed to be coherent without any software intervention.

Finally, since flushing (clearing) cache line entries for a large data buffer can be time-consuming, if the data buffer is larger than a preset (run-time configurable) size, the entire cache is flushed.

RETURNS

N/A

SEE ALSO **scsi2Lib**

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 the page format bit (0-1) */
    int             saveParams,  /* value of the save parameters bit (0-1) */
    char *          buffer,       /* ptr to output data buffer */
    int             bufLength     /* length of buffer in bytes */
    )
```

DESCRIPTION

This routine issues a **MODE_SELECT** command to a specified SCSI device.

RETURNS

OK, or ERROR if the command fails.

SEE ALSO

scsiLib

scsiModeSense( )

NAME

scsiModeSense( ) – issue a **MODE_SENSE** command to a SCSI device

SYNOPSIS

```
STATUS scsiModeSense
    (
    SCSI_PHYS_DEV * pScsiPhysDev, /* ptr to SCSI physical device */
    int             pageControl,  /* value of the page control field (0-3) */
    int             pageCode,     /* value of the page code field (0-0x3f) */
    char *          buffer,       /* ptr to input data buffer */
    int             bufLength     /* length of buffer in bytes */
    )
```

DESCRIPTION

This routine issues a **MODE_SENSE** command to a specified SCSI device.

RETURNS

OK, or ERROR if the command fails.

SEE ALSO

scsiLib

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 */
    )
```

DESCRIPTIONNOTE This function is intended for use only by SCSI controller drivers.

RETURNS OK, or ERROR if the message is not supported.

SEE ALSO **scsi2Lib**

scsiPhysDevCreate( )

NAME *scsiPhysDevCreate*( ) – create a SCSI physical device structure

SYNOPSIS

```
SCSI_PHYS_DEV * scsiPhysDevCreate
    (
    SCSI_CTRL * pScsiCtrl, /* ptr to SCSI controller info */
    int  devBusId,         /* device's SCSI bus ID */
    int  devLUN,           /* device's logical unit number */
    int  reqSenseLength,   /* length of REQUEST SENSE data dev returns */
    int  devType,          /* type of SCSI device */
    BOOL removable,        /* whether medium is removable */
    int  numBlocks,        /* number of blocks on device */
    int  blockSize         /* size of a block in bytes */
    )
```

DESCRIPTION This routine enables access to a SCSI device and must be the first routine invoked. It must be called once for each physical device on the SCSI bus.

If *reqSenseLength* is NULL (0), one or more **REQUEST_SENSE** commands are issued to the device to determine the number of bytes of sense data it typically returns. Note that if the device returns variable amounts of sense data depending on its state, you must consult the device manual to determine the maximum amount of sense data that can be returned.

If *devType* is NONE (-1), an INQUIRY command is issued to determine the device type; as an added benefit, it acquires the device's make and model number. The ***scsiShow*( )** routine displays this information. Common values of *devType* can be found in **scsiLib.h** or in the SCSI specification.

If *numBlocks* or *blockSize* are specified as NULL (0), a **READ_CAPACITY** command is issued to determine those values. This occurs only for device types supporting **READ_CAPACITY**.

RETURNS A pointer to the created **SCSI_PHYS_DEV** structure, or NULL if the routine is unable to create the physical-device structure.

SEE ALSO **scsiLib**

---

scsiPhysDevDelete( )

NAME ***scsiPhysDevDelete*( )** – delete a SCSI physical-device structure

SYNOPSIS

```
STATUS scsiPhysDevDelete
    (
    SCSI_PHYS_DEV * pScsiPhysDev /* ptr to SCSI physical device info */
    )
```

DESCRIPTION This routine deletes a specified SCSI physical-device structure.

RETURNS OK, or ERROR if **pScsiPhysDev** is NULL or **SCSI_BLK_DEVs** have been created on the device.

SEE ALSO **scsiLib**

---

scsiPhysDevIdGet( )

NAME ***scsiPhysDevIdGet*( )** – return a pointer to a **SCSI_PHYS_DEV** structure

SYNOPSIS

```
SCSI_PHYS_DEV * scsiPhysDevIdGet
    (
    SCSI_CTRL * pScsiCtrl, /* ptr to SCSI controller info */
    int         devBusId,  /* device's SCSI bus ID */
    int         devLUN     /* device's logical unit number */
    )
```

DESCRIPTION This routine returns a pointer to the **SCSI_PHYS_DEV** structure of the SCSI physical device located at a specified bus ID (*devBusId*) and logical unit number (*devLUN*) and attached to a specified SCSI controller (*pScsiCtrl*).

RETURNS A pointer to the **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 bytes or blocks from a SCSI tape device

SYNOPSIS

```
int scsiRdTape
    (
    SCSI_SEQ_DEV * pScsiSeqDev, /* ptr to SCSI sequential device info */
    UINT           count,       /* total bytes or blocks to be read */
    char *         buffer,      /* ptr to input data buffer */
    BOOL           fixedSize    /* if variable size blocks */
    )
```

DESCRIPTION This routine reads the specified number of bytes or blocks from a specified physical device. If the boolean *fixedSize* is true, then *numBytes*represents the number of blocks of size *blockSize*, defined in the **pScsiPhysDev** structure. If variable block sizes are used (*fixedSize* = FALSE), then *numBytes* represents the actual number of bytes to be read.

RETURNS Number of bytes or blocks actually read, 0 if EOF, or ERROR.

SEE ALSO **scsiSeqLib**

---

scsiReadCapacity( )

NAME *scsiReadCapacity*( ) – issue a **READ_CAPACITY** command to a SCSI device

SYNOPSIS

```
STATUS scsiReadCapacity
    (
    SCSI_PHYS_DEV * pScsiPhysDev, /* ptr to SCSI physical device */
    int *           pLastLBA,     /* where to return last logical block */
    int *           pBlkLength    /* where to return block length */
    )
```

DESCRIPTION This routine issues a **READ_CAPACITY** command to a specified SCSI device.

RETURNS OK, or ERROR if the command fails.

SEE ALSO **scsiLib**

scsiRelease( )

NAME *scsiRelease*( ) – issue a RELEASE command to a SCSI device

SYNOPSIS

```
STATUS scsiRelease
    (
    SCSI_PHYS_DEV * pScsiPhysDev /* ptr to SCSI physical device */
    )
```

DESCRIPTION This routine issues a RELEASE command to a specified SCSI device.

RETURNS OK, or ERROR if the command fails.

SEE ALSO **scsiDirectLib**

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**

2

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

2

scsiSeqDevCreate( )

NAME *scsiSeqDevCreate*( ) – create a SCSI sequential device

SYNOPSIS

```
SEQ_DEV *scsiSeqDevCreate
    (
    SCSI_PHYS_DEV * pScsiPhysDev /* ptr to SCSI physical device info */
    )
```

DESCRIPTION This routine creates a SCSI sequential device and saves a pointer to this **SEQ_DEV** in the SCSI physical device. The following functions are initialized in this structure:

sd_seqRd	–	*scsiRdTape*( )
sd_seqWrt	–	*scsiWrtTape*( )
sd_ioctl	–	*scsiIoctl*( ) (in **scsiLib**)
sd_seqWrtFileMarks	–	*scsiWrtFileMarks*( )
sd_statusChk	–	*scsiSeqStatusCheck*( )
sd_reset	–	(not used)
sd_rewind	–	*scsiRewind*( )
sd_reserve	–	*scsiReserve*( )
sd_release	–	*scsiRelease*( )
sd_readBlkLim	–	*scsiSeqReadBlockLimits*( )
sd_load	–	*scsiLoadUnit*( )
sd_space	–	*scsiSpace*( )
sd_erase	–	*scsiErase*( )

Only one **SEQ_DEV** per **SCSI_PHYS_DEV** is allowed, unlike **BLK_DEVs** where an entire list is maintained. Therefore, this routine can be called only once per creation of a sequential device.

RETURNS A pointer to the **SEQ_DEV** structure, or NULL if the command fails.

SEE ALSO **scsiSeqLib**

scsiSeqIoctl( )

NAME

scsiSeqIoctl( ) – perform an I/O control function for sequential access devices

SYNOPSIS

```
int scsiSeqIoctl
    (
    SCSI_SEQ_DEV * pScsiSeqDev, /* ptr to SCSI sequential device */
    int            function,    /* ioctl function code */
    int            arg          /* argument to pass to called function */
    )
```

DESCRIPTION

This routine issues **scsiSeqLib** commands to perform sequential device-specific I/O control operations.

RETURNS

OK or ERROR.

ERRNO

S_scsiLib_INVALID_BLOCK_SIZE

SEE ALSO

scsiSeqLib

scsiSeqReadBlockLimits( )

NAME

scsiSeqReadBlockLimits( ) – issue a **READ_BLOCK_LIMITS** command to a SCSI device

SYNOPSIS

```
STATUS scsiSeqReadBlockLimits
    (
    SCSI_SEQ_DEV * pScsiSeqDev,     /* ptr to SCSI sequential device */
    int *          pMaxBlockLength, /* where to return max block length */
    UINT16 *       pMinBlockLength  /* where to return min 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.

ERRNO

S_scsiLib_ILLEGAL_REQUEST

SEE ALSO

scsiSeqLib

scsiStartStopUnit( )

NAME

scsiStartStopUnit( ) – issue a **START_STOP_UNIT** command to a SCSI device

SYNOPSIS

```
STATUS scsiStartStopUnit
    (
    SCSI_PHYS_DEV * pScsiPhysDev, /* ptr to SCSI physical device */
    BOOL            start         /* TRUE == start, FALSE == stop */
    )
```

DESCRIPTION

This routine issues a **START_STOP_UNIT** command to a specified SCSI device.

RETURNS OK, or ERROR if the command fails.

SEE ALSO **scsiDirectLib**

scsiSyncXferNegotiate( )

NAME *scsiSyncXferNegotiate*( ) – initiate or continue negotiating transfer parameters

SYNOPSIS

```
void scsiSyncXferNegotiate
    (
    SCSI_CTRL *           pScsiCtrl,   /* ptr to SCSI controller info */
    SCSI_TARGET *         pScsiTarget, /* ptr to SCSI target info */
    SCSI_SYNC_XFER_EVENT eventType    /* tells what has just happened */
    )
```

DESCRIPTION This routine manages negotiation by means of a finite-state machine which is driven by "significant events" such as incoming and outgoing messages. Each SCSI target has its own independent state machine.

NOTE If the controller does not support synchronous transfer or if the target's maximum REQ/ACK offset is zero, attempts to initiate a round of negotiation are ignored.

This function is intended for use only by SCSI controller drivers.

RETURNS N/A

SEE ALSO **scsi2Lib**

scsiTapeModeSelect( )

NAME *scsiTapeModeSelect*( ) – issue a **MODE_SELECT** command to a SCSI tape device

SYNOPSIS

```
STATUS scsiTapeModeSelect
    (
    SCSI_PHYS_DEV *pScsiPhysDev,/* ptr to SCSI physical device            */
    int pageFormat,             /* value of the page format bit (0-1)     */
    int saveParams,             /* value of the save parameters bit (0-1) */
    char *buffer,               /* ptr to output data buffer              */
    int bufLength               /* length of buffer in bytes              */
    )
```

DESCRIPTION This routine issues a **MODE_SELECT** command to a specified SCSI device.

RETURNS OK, or ERROR if the command fails.

SEE ALSO **scsiSeqLib**

scsiTapeModeSense( )

NAME *scsiTapeModeSense*( ) – issue a **MODE_SENSE** command to a SCSI tape device

SYNOPSIS

```
STATUS scsiTapeModeSense
    (
    SCSI_PHYS_DEV *pScsiPhysDev,/* ptr to SCSI physical device           */
    int pageControl,            /* value of the page control field (0-3) */
    int pageCode,               /* value of the page code field (0-0x3f) */
    char *buffer,               /* ptr to input data buffer              */
    int bufLength               /* length of buffer in bytes             */
    )
```

DESCRIPTION This routine issues a **MODE_SENSE** command to a specified SCSI tape device.

RETURNS OK, or ERROR if the command fails.

SEE ALSO **scsiSeqLib**

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 parentheses):

SCSI_SET_OPT_TIMEOUT	**selTimeOut**	select timeout period, microseconds
SCSI_SET_OPT_MESSAGES	**messages**	FALSE to disable SCSI messages
SCSI_SET_OPT_DISCONNECT	**disconnect**	FALSE to disable discon/recon
SCSI_SET_OPT_XFER_PARAMS	**maxOffset,**	max sync xfer offset, 0>async
	minPeriod	min sync xfer period, x 4 nsec.
SCSI_SET_OPT_TAG_PARAMS	**tagType,**	default tag type (**SCSI_TAG_***)
	maxTags	max cmd tags available
SCSI_SET_OPT_WIDE_PARAMS	**xferWidth**	data transfer width 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.

NOTE If the controller does not support wide transfers or the target's transfer width is zero, attempts to initiate a round of negotiation are ignored; this is because zero is the default narrow transfer.

This function is intended for use only by SCSI controller drivers.

RETURNS N/A

SEE ALSO **scsi2Lib**

scsiWrtFileMarks( )

NAME *scsiWrtFileMarks*( ) – write file marks to a SCSI sequential device

SYNOPSIS

```
STATUS scsiWrtFileMarks
    (
    SCSI_SEQ_DEV * pScsiSeqDev, /* ptr to SCSI sequential device info */
    int            numMarks,    /* number of file marks to write */
    BOOL           shortMark    /* TRUE to write short file mark */
    )
```

DESCRIPTION This routine writes file marks to a specified physical device.

RETURNS OK, or ERROR if the file mark cannot be written.

SEE ALSO **scsiSeqLib**

scsiWrtSecs( )

NAME

scsiWrtSecs( ) – write sector(s) to a SCSI block device

SYNOPSIS

```
STATUS scsiWrtSecs
    (
    SCSI_BLK_DEV * pScsiBlkDev, /* ptr to SCSI block device info */
    int            sector,      /* sector number to be written */
    int            numSecs,     /* total sectors to be written */
    char *         buffer       /* ptr to input data buffer */
    )
```

DESCRIPTION

This routine writes the specified physical sector(s) to a specified physical device.

RETURNS

OK, or ERROR if the sector(s) cannot be written.

SEE ALSO

scsiLib

scsiWrtTape( )

NAME

scsiWrtTape( ) – write data to a SCSI tape device

SYNOPSIS

```
STATUS scsiWrtTape
    (
    SCSI_SEQ_DEV * pScsiSeqDev, /* ptr to SCSI sequential device info */
    int            numBytes,    /* total bytes or blocks to be written */
    char *         buffer,      /* ptr to input data buffer */
    BOOL           fixedSize    /* if variable size blocks */
    )
```

DESCRIPTION

This routine writes data to the current block on a specified physical device. If the boolean *fixedSize* is true, then *numBytes*represents the number of blocks of size *blockSize*, defined in the **pScsiPhysDev** structure. If variable block sizes are used (*fixedSize* = FALSE), then *numBytes* represents the actual number of bytes to be written. If *numBytes* is greater than the **maxBytesLimit** field defined in the **pScsiPhysDev** structure, then more than one SCSI transaction is used to transfer the data.

RETURNS

OK, or ERROR if the data cannot be written or zero bytes are written.

SEE ALSO

scsiSeqLib

select( )

NAME *select*( ) – pend on a set of file descriptors

SYNOPSIS

```
int select
    (
    int              width,      /* number of bits to examine from 0 */
    fd_set *         pReadFds,   /* read fds */
    fd_set *         pWriteFds,  /* write fds */
    fd_set *         pExceptFds, /* exception fds (unsupported) */
    struct timeval * pTimeOut    /* max time to wait, NULL = forever */
    )
```

DESCRIPTION This routine permits a task to pend until one of a set of file descriptors becomes ready. Three parameters -- *pReadFds*, *pWriteFds*, and *pExceptFds* -- point to file descriptor sets in which each bit corresponds to a particular file descriptor. Bits set in the read file descriptor set (*pReadFds*) will cause ***select*( )** to pend until data is available on any of the corresponding file descriptors, while bits set in the write file descriptor set (*pWriteFds*) will cause ***select*( )** to pend until any of the corresponding file descriptors become writable. (The *pExceptFds* parameter is currently unused, but is provided for UNIX call compatibility.)

The following macros are available for setting the appropriate bits in the file descriptor set structure:

```
FD_SET(fd, &fdset)
FD_CLR(fd, &fdset)
FD_ZERO(&fdset)
```

If either *pReadFds* or *pWriteFds* is NULL, they are ignored. The *width* parameter defines how many bits will be examined in the file descriptor sets, and should be set to either the maximum file descriptor value in use plus one, or simply to **FD_SETSIZE**. When ***select*( )** returns, it zeros out the file descriptor sets, and sets only the bits that correspond to file descriptors that are ready. The **FD_ISSET** macro may be used to determine which bits are set.

If *pTimeOut* is NULL, ***select*( )** will block indefinitely. If *pTimeOut* is not NULL, but points to a **timeval** structure with an effective time of zero, the file descriptors in the file descriptor sets will be polled and the results returned immediately. If the effective time value is greater than zero, ***select*( )** will return after the specified time has elapsed, even if none of the file descriptors are ready.

Applications can use ***select*( )** with pipes and serial devices, in addition to sockets. Also, ***select*( )** now examines write file descriptors in addition to read file descriptors; however, exception file descriptors remain unsupported.

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      /* readers (SELREAD) or writers (SELWRITE) */
    )
```

DESCRIPTION

This routine wakes up all tasks pended in *select*( ) that are waiting for a device; it is called by a driver when the device becomes ready. The *type* parameter specifies the task to be awakened, either reader tasks (SELREAD) or writer tasks (SELWRITE).

RETURNS

N/A

SEE ALSO

selectLib

selWakeupListInit( )

NAME

selWakeupListInit( ) – initialize a *select*( ) wake-up list

SYNOPSIS

```
void selWakeupListInit
    (
    SEL_WAKEUP_LIST * pWakeupList /* wake-up list to initialize */
    )
```

DESCRIPTION

This routine should be called in a device's create routine to initialize the **SEL_WAKEUP_LIST** structure.

RETURNS

N/A

SEE ALSO

selectLib

selWakeupListLen( )

NAME *selWakeupListLen*( ) – get the number of nodes in a *select*( ) wake-up list

SYNOPSIS
```
int selWakeupListLen
    (
    SEL_WAKEUP_LIST * pWakeupList /* list of tasks to wake up */
    )
```

DESCRIPTION This routine returns the number of nodes in a specified **SEL_WAKEUP_LIST**. It can be used by a driver to determine if any tasks are currently pended in *select*( ) on this device, and whether these tasks need to be activated with *selWakeupAll*( ).

RETURNS The number of nodes currently in a *select*( ) wake-up list, or ERROR.

SEE ALSO **selectLib**

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**.

AVAILABILITY This routine is distributed as a component of the unbundled shared memory support option, VxMP.

RETURNS The semaphore ID, or NULL if memory cannot be allocated from the shared semaphore dedicated memory partition.

ERRNO **S_memLib_NOT_ENOUGH_MEMORY**, **S_semLib_INVALID_QUEUE_TYPE**, **S_semLib_INVALID_STATE**, **S_smObjLib_LOCK_TIMEOUT**

SEE ALSO **semSmLib**, **semLib**, **semBLib**, **smObjLib**, **semShow**

---

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**.

AVAILABILITY

This routine is distributed as a component of the unbundled shared memory support option, VxMP.

RETURNS

The semaphore ID, or NULL if memory cannot be allocated from the shared semaphore dedicated memory partition.

ERRNO

S_memLib_NOT_ENOUGH_MEMORY, **S_semLib_INVALID_QUEUE_TYPE**, **S_smObjLib_LOCK_TIMEOUT**

SEE ALSO

semSmLib, **semLib**, **semCLib**, **smObjLib**, **semShow**

---

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.

ERRNO **S_intLib_NOT_ISR_CALLABLE**, **S_objLib_OBJ_ID_ERROR**, **S_smObjLib_NO_OBJECT_DESTROY**

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.

ERRNO **S_objLib_OBJ_ID_ERROR**

SEE ALSO **semLib**, **semBLib**, **semCLib**, **semMLib**, **semSmLib**

semGive( )

NAME *semGive*( ) – give a semaphore

SYNOPSIS

```
STATUS semGive
    (
    SEM_ID semId /* semaphore ID to give */
    )
```

DESCRIPTION This routine performs the give operation on a specified semaphore. Depending on the type of semaphore, the state of the semaphore and of the pending tasks may be affected. 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.

ERRNO **S_intLib_NOT_ISR_CALLABLE**, **S_objLib_OBJ_ID_ERROR**, **S_semLib_INVALID_OPERATION**

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. It is called automatically when the this show facility is configured into VxWorks using either of the following methods:

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

– If you use the Tornado project facility, select **INCLUDE_POSIX_SEM_SHOW**.

RETURNS OK, or ERROR if an error occurs installing the file pointer show routine.

SEE ALSO **semPxShow**

semShow( )

NAME *semShow*( ) – show information about a semaphore

SYNOPSIS

```
STATUS semShow
    (
    SEM_ID semId, /* semaphore to display */
    int    level  /* 0 = summary, 1 = details */
    )
```

DESCRIPTION This routine displays the state and optionally the pended tasks of a semaphore.

A summary of the state of the semaphore is displayed as follows:

```
Semaphore Id        : 0x585f2
Semaphore Type      : BINARY
Task Queuing        : PRIORITY
Pended Tasks        : 1
State               : EMPTY {Count if COUNTING, Owner if MUTEX}
```

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 the semaphore show facility is configured into VxWorks using either of the following methods:

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

– If you use the Tornado project facility, select **INCLUDE_SEM_SHOW**.

RETURNS N/A

SEE ALSO **semShow**

semTake( )

NAME *semTake***( )** – take a semaphore

SYNOPSIS

```
STATUS semTake
    (
    SEM_ID semId,  /* semaphore ID to take */
    int    timeout /* timeout in ticks */
    )
```

DESCRIPTION This routine performs the take operation on a specified semaphore. Depending on the type of semaphore, the state of the semaphore and the calling task may be affected. The behavior of ***semTake*( )** is discussed fully in the library description of the specific semaphore type being used.

A timeout in ticks may be specified. If a task times out, ***semTake*( )** will return ERROR. Timeouts of **WAIT_FOREVER** (-1) and **NO_WAIT** (0) indicate to wait indefinitely or not to wait at all.

When ***semTake*( )** returns due to timeout, it sets the errno to **S_objLib_OBJ_TIMEOUT** (defined in **objLib.h**).

The ***semTake*( )** routine is not callable from interrupt service routines.

RETURNS OK, or ERROR if the semaphore ID is invalid or the task timed out.

ERRNO **S_intLib_NOT_ISR_CALLABLE**, **S_objLib_OBJ_ID_ERROR**, **S_objLib_OBJ_UNAVAILABLE**

SEE ALSO **semLib**, **semBLib**, **semCLib**, **semMLib**, **semSmLib**

2

sem_close( )

NAME
sem_close( ) – close a named semaphore (POSIX)

SYNOPSIS

```
int sem_close
    (
    sem_t * sem /* semaphore descriptor */
    )
```

DESCRIPTION
This routine is called to indicate that the calling task is finished with the specified named semaphore, *sem*. Do not call this routine with an unnamed semaphore (i.e., one created by ***sem_init*( )**); the effects are undefined. The ***sem_close*( )** call deallocates any system resources allocated by the system for use by this task for this semaphore.

If the semaphore has not been removed with a call to ***sem_unlink*( )**, then ***sem_close*( )** has no effect on the state of the semaphore. However, if the semaphore has been unlinked, the semaphore vanishes when the last task closes it.

WARNING
Take care to avoid risking the deletion of a semaphore that another task has already locked. Applications should only close semaphores that the closing task has opened.

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

ERRNO
EINVAL
– invalid semaphore descriptor.

SEE ALSO
semPxLib, ***sem_unlink*( )**, ***sem_open*( )**, ***sem_init*( )**

sem_destroy( )

NAME
sem_destroy( ) – destroy an unnamed semaphore (POSIX)

SYNOPSIS

```
int sem_destroy
    (
    sem_t * sem /* semaphore descriptor */
    )
```

DESCRIPTION
This routine is used to destroy the unnamed semaphore indicated by *sem*.

The ***sem_destroy*( )** call can only destroy a semaphore created by ***sem_init*( )**. Calling ***sem_destroy*( )** with a named semaphore will cause a **EINVAL** error. Subsequent use of the *sem* semaphore will cause an **EINVAL** error in the calling function.

If one or more tasks is blocked on the semaphore, the semaphore is not destroyed.

WARNING Take care when deleting semaphores, particularly those used for mutual exclusion, to avoid deleting a semaphore out from under a task that has already locked that semaphore. Applications should adopt the protocol of only deleting semaphores that the deleting task has successfully locked.

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

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 */
    )
```

DESCRIPTION

This routine establishes a connection between a named semaphore and a task. Following a call to ***sem_open*( )** with a semaphore name *name*, the task may reference the semaphore associated with *name* using the address returned by this call. This semaphore may be used in subsequent calls to ***sem_wait( )***, ***sem_trywait( )***, and ***sem_post( )***. The semaphore remains usable until the semaphore is closed by a successful call to ***sem_close( )***.

The *oflag* argument controls whether the semaphore is created or merely accessed by the call to ***sem_open( )***. The following flag bits may be set in *oflag*:

O_CREAT
Use this flag to create a semaphore if it does not already exist. If **O_CREAT** is set and the semaphore already exists, **O_CREAT** has no effect except as noted below under **O_EXCL**. Otherwise, ***sem_open( )*** creats a semaphore. **O_CREAT** requires a third and fourth argument: *mode*, which is of type mode_t, and *value*, which is of type unsigned int. *mode* has no effect in this implementation. The semaphore is created with an initial value of *value*. Valid initial values for semaphores must be less than or equal to **SEM_VALUE_MAX**.

O_EXCL
If **O_EXCL** and **O_CREAT** are set, ***sem_open( )*** will fail if the semaphore name exists. If **O_EXCL** is set and **O_CREAT** is not set, the named semaphore is not created.

To determine whether a named semaphore already exists in the system, call ***sem_open( )*** with the flags **O_CREAT | O_EXCL**. If the ***sem_open( )*** call fails, the semaphore exists.

If a task makes multiple calls to ***sem_open( )*** with the same value for *name*, then the same semaphore address is returned for each such call, provided that there have been no calls to ***sem_unlink( )*** for this semaphore.

References to copies of the semaphore will produce undefined results.

NOTE

The current implementation has the following limitations:

– A semaphore cannot be closed with calls to ***_exit( )*** or ***exec( )***.
– A semaphore cannot be implemented as a file.
– Semaphore names will not appear in the file system.

RETURNS

A pointer to sem_t, or -1 (ERROR) if unsuccessful.

ERRNO

EEXIST
– **O_CREAT | O_EXCL** are set and the semaphore already exists.
EINVAL
– *value* exceeds **SEM_VALUE_MAX** or the semaphore name is invalid.
ENAMETOOLONG
– the semaphore name is too long.
ENOENT
– the named semaphore does not exist and **O_CREAT** is not set.
ENOSPC
– the semaphore could not be initialized due to resource constraints.

SEE ALSO

semPxLib, ***sem_unlink( )***

sem_post( )

NAME *sem_post*( ) – unlock (give) a semaphore (POSIX)

SYNOPSIS
```
int sem_post
    (
    sem_t * sem /* semaphore descriptor */
    )
```

DESCRIPTION This routine unlocks the semaphore referenced by *sem* by performing the semaphore unlock operation on that semaphore.

If the semaphore value resulting from the operation is positive, then no tasks were blocked waiting for the semaphore to become unlocked; the semaphore value is simply incremented.

If the value of the semaphore resulting from this semaphore is zero, then one of the tasks blocked waiting for the semaphore will return successfully from its call to ***sem_wait*( )**.

NOTE The **_POSIX_PRIORITY_SCHEDULING** functionality is not yet supported.

Note that the POSIX terms *unlock* and *post* correspond to the term *give* used in other VxWorks semaphore documentation.

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

ERRNO **EINVAL**
– invalid semaphore descriptor.

SEE ALSO **semPxLib**, *sem_wait*( ), *sem_trywait*( )

sem_trywait( )

NAME *sem_trywait*( ) – lock (take) a semaphore, returning error if unavailable (POSIX)

SYNOPSIS
```
int sem_trywait
    (
    sem_t * sem /* semaphore descriptor */
    )
```

DESCRIPTION

This routine locks the semaphore referenced by *sem* only if the semaphore is currently not locked; that is, if the semaphore value is currently positive. Otherwise, it does not lock the semaphore. In either case, this call returns immediately without blocking.

Upon return, the state of the semaphore is always locked (either as a result of this call or by a previous ***sem_wait*( )** or ***sem_trywait*( )**). The semaphore will remain locked until ***sem_post*( )** is executed and returns successfully.

Deadlock detection is not implemented.

Note that the POSIX term *lock* corresponds to the term *take* used in other VxWorks semaphore documentation.

RETURNS

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

ERRNO

EAGAIN – semaphore is already locked.
EINVAL – invalid semaphore descriptor.

SEE ALSO

semPxLib, ***sem_wait*( )**, ***sem_post*( )**

sem_unlink( )

NAME

***sem_unlink*( )** – remove a named semaphore (POSIX)

SYNOPSIS

```
int sem_unlink
    (
    const char * name /* semaphore name */
    )
```

DESCRIPTION

This routine removes the string *name* from the semaphore name table, and marks the corresponding semaphore for destruction. An unlinked semaphore is destroyed when the last task closes it with ***sem_close*( )**. After a particular name is removed from the table, calls to ***sem_open*( )** using the same name cannot connect to the same semaphore, even if other tasks are still using it. Instead, such calls refer to a new semaphore with the same name.

RETURNS

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

ERRNO

ENAMETOOLONG
– semaphore name too long.
ENOENT
– named semaphore does not exist.

SEE ALSO

semPxLib, ***sem_open*( )**, ***sem_close*( )**

sem_wait( )

NAME

sem_wait( ) – lock (take) a semaphore, blocking if not available (POSIX)

SYNOPSIS

```
int sem_wait
    (
    sem_t * sem /* semaphore descriptor */
    )
```

DESCRIPTION

This routine locks the semaphore referenced by *sem* by performing the semaphore lock operation on that semaphore. If the semaphore value is currently zero, the calling task will not return from the call to ***sem_wait*( )** until it either locks the semaphore or the call is interrupted by a signal.

On return, the state of the semaphore is locked and will remain locked until ***sem_post*( )** is executed and returns successfully.

Deadlock detection is not implemented.

Note that the POSIX term *lock* corresponds to the term *take* used in other VxWorks documentation regarding semaphores.

RETURNS

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

ERRNO

EINVAL
– invalid semaphore descriptor, or semaphore destroyed while task waiting.

SEE ALSO

semPxLib, *sem_trywait*( ), *sem_post*( )

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.

For BSD 4.4 sockets a copy of the *mp*>msg_iov array will be made. This requires a cluster from the network stack system pool of size *mp*>msg_iovlen * sizeof (struct iovec) or 8 bytes.

RETURNS The number of bytes sent, or ERROR if the call fails.

SEE ALSO **sockLib**, ***sendto*( )**

sendto( )

NAME *sendto*( ) – send a message to a socket

SYNOPSIS

```
int sendto
    (
    int                 s,      /* socket to send data to */
    caddr_t             buf,    /* pointer to data buffer */
    int                 bufLen, /* length of buffer */
    int                 flags,  /* flags to underlying protocols */
    struct sockaddr *   to,     /* recipient's address */
    int                 tolen   /* length of to sockaddr */
    )
```

DESCRIPTION This routine sends a message to the datagram socket named by *to*. The socket *s* is received by the receiver as the sending socket.

The maximum length of *buf* is subject to the limits on UDP buffer size. See the discussion of **SO_SNDBUF** in the ***setsockopt*( )** manual entry.

You can OR the following values into the *flags* parameter with this operation:

MSG_OOB (0x1)
Out-of-band data.

MSG_DONTROUTE (0x4)
Send without using routing tables.

RETURNS The number of bytes sent, or ERROR if the call fails.

SEE ALSO **sockLib**, *setsockopt*( )

set_new_handler( )

NAME *set_new_handler*( ) – set new_handler to user-defined function (C++)

SYNOPSIS

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

DESCRIPTION This function is used to define the function that will be called when operator new cannot allocate memory.

The new_handler acts for all threads in the system; you cannot set a different handler for different tasks.

RETURNS A pointer to the previous value of new_handler.

INCLUDE FILES **new**

SEE ALSO **cplusLib**

set_terminate( )

NAME *set_terminate*( ) – set terminate to user-defined function (C++)

SYNOPSIS

```
extern void (*set_terminate (void(* terminate_handler)()))  ()
```

DESCRIPTION This function is used to define the terminate_handler which will be called when an uncaught exception is raised.

The terminate_handler acts for all threads in the system; you cannot set a different handler for different tasks.

RETURNS The previous terminate_handler.

INCLUDE FILES **exception**

SEE ALSO **cplusLib**

setbuf( )

NAME *setbuf*( ) – specify the buffering for a stream (ANSI)

SYNOPSIS

```
void setbuf
    (
    FILE * fp, /* stream to set buffering for */
    char * buf /* buffer to use */
    )
```

DESCRIPTION Except that it returns no value, this routine is equivalent to ***setvbuf*( )** invoked with the *mode* _IOFBF (full buffering) and *size* BUFSIZ, or (if *buf* is a null pointer), with the *mode* _IONBF (no buffering).

INCLUDE FILES **stdio.h**

RETURNS N/A

SEE ALSO **ansiStdio**, *setvbuf*( )

setbuffer( )

NAME *setbuffer*( ) – specify buffering for a stream

SYNOPSIS

```
void setbuffer
    (
    FILE * fp,  /* stream to set buffering for */
    char * buf, /* buffer to use */
    int    size /* buffer size */
    )
```

DESCRIPTION This routine specifies a buffer *buf* to be used for a stream in place of the automatically allocated buffer. If *buf* is NULL, the stream is unbuffered. This routine should be called only after the stream has been associated with an open file and before any other operation is performed on the stream.

This routine is provided for compatibility with earlier VxWorks releases.

INCLUDE FILES **stdio.h**

RETURNS N/A

SEE ALSO **ansiStdio**, *setvbuf*( )

sethostname( )

NAME *sethostname*( ) – set the symbolic name of this machine

SYNOPSIS

```
int sethostname
    (
    char * name,   /* machine name */
    int    nameLen /* length of name */
    )
```

DESCRIPTION This routine sets the target machine's symbolic name, which can be used for identification.

RETURNS

OK or ERROR.

SEE ALSO

hostLib

setjmp( )

NAME

setjmp( ) – save the calling environment in a **jmp_buf** argument (ANSI)

SYNOPSIS

```
int setjmp
    (
    jmp_buf env
    )
```

DESCRIPTION

This routine saves the calling environment in *env*, in order to permit a ***longjmp*( )** call to restore that environment (thus performing a non-local goto).

Constraints on Calling Environment

The ***setjmp*( )** routine may only be used in the following contexts:

– as the entire controlling expression of a selection or iteration statement;

– as one operand of a relational or equality operator, in the controlling expression of a selection or iteration statement;

– 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 is included for ANSI compatibility. Only the default is implemented. At program start-up, the equivalent of the following is executed:

```
    setlocale (LC_ALL, "C");
```

This specifies the program’s entire locale and the minimal environment for C translation.

INCLUDE FILES **locale.h**, **string.h**, **stdlib.h**

RETURNS A pointer to the string "C".

SEE ALSO **ansiLocale**

setsockopt( )

NAME *setsockopt*( ) – set socket options

SYNOPSIS

```
STATUS setsockopt
    (
    int    s,       /* target socket */
    int    level,   /* protocol level of option */
    int    optname, /* option name */
    char * optval,  /* pointer to option value */
    int    optlen   /* option length */
    )
```

DESCRIPTION This routine sets the options associated with a socket. To manipulate options at the "socket" level, *level* should be **SOL_SOCKET**. Any other levels should use the appropriate protocol number.

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 in response to the shutdown of a connection, 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.

SO_DEBUG -- Debugging the underlying protocol

Specify the **SO_DEBUG** option to let the underlying protocol module record debug information.

```
setsockopt (sock, SOL_SOCKET, SO_KEEPALIVE, &optval, sizeof (optval));
```

The value at *optval* for this option is an integer (type **int**), either 1 (on) or 0 (off).

OPTION FOR DATAGRAM SOCKETS

The following section discusses an option for datagram (UDP) sockets.

SO_BROADCAST -- Sending to Multiple Destinations

Specify the **SO_BROADCAST** option when an application needs to send data to more than one destination:

```
setsockopt (sock, SOL_SOCKET, SO_BROADCAST, &optval, sizeof (optval));
```

The value at *optval* is an integer (type *int*), either 1 (on) or 0 (off).

OPTIONS FOR DATAGRAM AND RAW SOCKETS

The following section discusses options for multicasting on UDP and RAW sockets.

IP_ADD_MEMBERSHIP -- Join a Multicast Group

Specify the **IP_ADD_MEMBERSHIP** option when a process needs to join multicast group:

```
setsockopt (sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, (char *)&ipMreq,
            sizeof (ipMreq));
```

The value of *ipMreq* is an **ip_mreq** structure. **ipMreq.imr_multiaddr.s**_addr is the internet multicast address **ipMreq.imr_interface.s**_addr is the internet unicast address of the interface through which the multicast packet needs to pass.

IP_DROP_MEMBERSHIP -- Leave a Multicast Group

Specify the **IP_DROP_MEMBERSHIP** option when a process needs to leave a previously joined multicast group:

2

```
setsockopt (sock, IPPROTO_IP, IP_DROP_MEMBERSHIP, (char *)&ipMreq,
            sizeof (ipMreq));
```

The value of *ipMreq* is an **ip_mreq** structure. **ipMreq.imr_multiaddr.s**_addr is the internet multicast address. **ipMreq.imr_interface.s**_addr is the internet unicast address of the interface to which the multicast address was bound.

IP_MULTICAST_IF -- Select a Default Interface for Outgoing Multicasts

Specify the **IP_MULTICAST_IF** option when an application needs to specify an outgoing network interface through which all multicast packets are sent:

```
setsockopt (sock, IPPROTO_IP, IP_MULTICAST_IF, (char *)&ifAddr,
            sizeof (mCastAddr));
```

The value of *ifAddr* is an **in_addr** structure. **ifAddr.s**_addr is the internet network interface address.

IP_MULTICAST_TTL -- Select a Default TTL

Specify the **IP_MULTICAST_TTL** option when an application needs to select a default TTL (time to live) for outgoing multicast packets:

```
setsockopt (sock, IPPROTO_IP, IP_MULTICAST_TTL, &optval, sizeof(optval));
```

The value at *optval* is an integer (type *int*), time to live value.

optval(TTL)	Application	Scope
0		same interface
1		same subnet
31	local event video	
32		same site
63	local event audio	
64		same region
95	IETF channel 2 video	
127	IETF channel 1 video	
128		same continent
159	IETF channel 2 audio	
191	IETF channel 1 audio	
223	IETF channel 2 low-rate audio	
255	IETF channel 1 low-rate audio	
	unrestricted in scope	

IP_MULTICAST_LOOP -- Enable or Disable Loopback

Enable or disable loopback of outgoing multicasts.

```
setsockopt (sock, IPPROTO_IP, IP_MULTICAST_LOOP, &optval, sizeof(optval));
```

The value at *optval* is an integer (type *int*), either 1(on) or 0 (off).

OPTIONS FOR BOTH STREAM AND DATAGRAM SOCKETS

The following options can be used with either stream or datagram sockets.

SO_REUSEADDR -- Reusing a Socket Address

Specify **SO_REUSEADDR** 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 **SO_SNDBUF** to adjust the maximum size of the socket-level send buffer:

```
setsockopt (sock, SOL_SOCKET, SO_SNDBUF, &optval, sizeof (optval));
```

The value at *optval* is an integer (type **int**) that specifies the size of the socket-level send buffer to be allocated.

When stream or datagram sockets are created, each transport protocol reserves a set amount of space at the socket level for use when the sockets are attached to a protocol. For TCP, the default size of the send buffer is 8192 bytes. For UDP, the default size is 9216 bytes. Socket-level buffers are allocated dynamically from the mbuf pool.

The effect of setting the maximum size of buffers (for both **SO_SNDBUF** and **SO_RCVBUF**, described below) is not actually to allocate the mbufs from the mbuf pool, 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 **SO_RCVBUF** to adjust the maximum size of the socket-level receive buffer:

```
setsockopt (sock, SOL_SOCKET, SO_RCVBUF, &optval, sizeof (optval));
```

The value at *optval* is an integer (type **int**) that specifies the size of the socket-level receive buffer to be allocated.

When stream or datagram sockets are created, each transport protocol reserves a set amount of space at the socket level for use when the sockets are attached to a protocol.

For TCP, the default size is 8192 bytes. UDP reserves 41600 bytes, enough space for up to forty incoming datagrams (1 Kbyte each).

See the **SO_SNDBUF** discussion above for a discussion of the impact of buffer size on application performance.

SO_OOBINLINE -- Placing Urgent Data in the Normal Data Stream

Specify the **SO_OOBINLINE** option to place urgent data within the normal receive data stream:

```
setsockopt (sock, SOL_SOCKET, SO_OOBINLINE, &optval, sizeof (optval));
```

TCP provides an expedited data service 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

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 called with a single parameter indicating whether this is an interactive shell to be used from a terminal or a socket, or a shell that executes a script.

Normally, the shell is spawned in interactive mode by the root task, ***usrRoot*( )**, when VxWorks starts up. After that, ***shell*( )** is called only to execute scripts, or when the shell is restarted after an abort.

The shell gets its input from standard input and sends output to standard output. Both standard input and standard output are initially assigned to the console, but are redirected by ***telnetdTask*( )** and ***rlogindTask*( )**.

The shell is not reentrant, since **yacc** does not generate a reentrant parser. Therefore, there can be only a single shell executing at one time.

RETURNS N/A

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

shellHistory( )

NAME *shellHistory*( ) – display or set the size of shell history

SYNOPSIS

```
void shellHistory
    (
    int size /* 0 = display, >0 = set history to new size */
    )
```

DESCRIPTION This routine displays shell history, or resets the default number of commands displayed by shell history to *size*. By default, history size is 20 commands. Shell history is actually maintained by **ledLib**.

RETURNS N/A

SEE ALSO **shellLib**, **ledLib**, ***h*( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

shellInit( )

NAME *shellInit*( ) – start the shell

SYNOPSIS

```
STATUS shellInit
    (
    int stackSize, /* shell stack (0 = previous/default value) */
    int arg        /* argument to shell task */
    )
```

DESCRIPTION This routine starts the shell task. If the configuration macro **INCLUDE_SHELL** is defined, *shellInit*( ) is called by the root task, *usrRoot*( ), in **usrConfig.c**.

RETURNS OK or ERROR.

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

shellLock( )

NAME *shellLock*( ) – lock access to the shell

SYNOPSIS

```
BOOL shellLock
    (
    BOOL request /* TRUE = lock, FALSE = unlock */
    )
```

DESCRIPTION This routine locks or unlocks access to the shell. When locked, cooperating tasks, such as ***telnetdTask*( )** and ***rlogindTask*( )**, will not take the shell.

RETURNS TRUE if *request* is "lock" and the routine successfully locks the shell, otherwise FALSE. TRUE if *request* is "unlock" and the routine successfully unlocks the shell, otherwise FALSE.

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

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,      /* socket to shut down */
    int how     /* 0 = receives disallowed */
                /* 1 = sends disallowed */
                /* 2 = sends and receives disallowed */
    )
```

DESCRIPTION
This routine shuts down all, or part, of a connection-based socket *s*. If the value of *how* is 0, receives are disallowed. If *how* is 1, sends are disallowed. If *how* is 2, both sends and receives are disallowed.

RETURNS
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
    (
    intsigno,
    void(*pHandler) ()
    ))  ()
```

DESCRIPTION

This routine chooses one of three ways in which receipt of the signal number *signo* is to be subsequently handled. If the value of *pHandler* is **SIG_DFL**, default handling for that signal will occur. If the value of *pHandler* is **SIG_IGN**, the signal will be ignored. Otherwise, *pHandler*must point to a function to be called when that signal occurs.

RETURNS

The value of the previous signal handler, or **SIG_ERR**.

SEE ALSO

sigLib

sigpending( )

NAME *sigpending*( ) – retrieve the set of pending signals blocked from delivery (POSIX)

SYNOPSIS

```
int sigpending
    (
    sigset_t * pSet /* location to store pending signal set */
    )
```

DESCRIPTION This routine stores the set of signals that are blocked from delivery and that are pending for the calling process in the space pointed to by *pSet*.

RETURNS OK (0), or ERROR (-1) if the signal TCB cannot be allocated.

ERRNO ENOMEM

SEE ALSO **sigLib**

sigprocmask( )

NAME *sigprocmask*( ) – examine and/or change the signal mask (POSIX)

SYNOPSIS

```
int sigprocmask
    (
    int               how,  /* how signal mask will be changed */
    const sigset_t * pSet, /* location of new signal mask */
    sigset_t *        pOset /* location to store old signal mask */
    )
```

DESCRIPTION This routine allows the calling process to examine and/or change its signal mask. If the value of *pSet* is not NULL, it points to a set of signals to be used to change the currently blocked set.

The value of *how* indicates the manner in which the set is changed and consists of one of the following, defined in **signal.h**:

SIG_BLOCK
the resulting set is the union of the current set and the signal set pointed to by *pSet*.

SIG_UNBLOCK
the resulting set is the intersection of the current set and the complement of the signal set pointed to by *pSet*.

SIG_SETMASK
the resulting set is the signal set pointed to by *pSset*.

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

ERRNO **EINVAL**

SEE ALSO **sigLib**, ***sigsetmask*( )**, ***sigblock*( )**

---

sigqueue( )

NAME ***sigqueue*( )** – send a queued signal to a task

SYNOPSIS

```
int sigqueue
    (
    int                tid,
    int                signo,
    const union sigval value
    )
```

DESCRIPTION The function ***sigqueue*( )** sends the signal specified by *signo* with the signal-parameter value specified by *value* to the process specified by *tid*.

RETURNS OK (0), or ERROR (-1) if the task ID or signal number is invalid, or if there are no queued-signal buffers available.

ERRNO **EINVAL EAGAIN**

SEE ALSO **sigLib**

---

sigqueueInit( )

NAME ***sigqueueInit*( )** – initialize the queued signal facilities

SYNOPSIS

```
int sigqueueInit
    (
    int nQueues
    )
```

DESCRIPTION

This routine initializes the queued signal facilities. It must be called before any call to ***sigqueue*( )**. It is usually called from the system start-up routine ***usrInit*( )** in usrConfig, after ***sysInit*( )** is called.

It allocates *nQueues* buffers to be used by ***sigqueue*( )**. A buffer is used by each call to ***sigqueue*( )** and freed when the signal is delivered (thus if a signal is block, the buffer is unavailable until the signal is unblocked.)

RETURNS

OK, or ERROR if memory could not be allocated.

SEE ALSO

sigLib

---

sigsetmask( )

NAME

***sigsetmask*( )** – set the signal mask

SYNOPSIS

```
int sigsetmask
    (
    int mask /* new signal mask */
    )
```

DESCRIPTION

This routine sets the calling task's signal mask to a specified value. A one (1) in the bit mask indicates that the specified signal is blocked from delivery. Use the macro **SIGMASK** to construct the mask for a specified signal number.

RETURNS

The previous value of the signal mask.

SEE ALSO

sigLib, ***sigprocmask*( )**

---

sigsuspend( )

NAME

***sigsuspend*( )** – suspend the task until delivery of a signal (POSIX)

SYNOPSIS

```
int sigsuspend
    (
    const sigset_t * pSet /* signal mask while suspended */
    )
```

DESCRIPTION This routine suspends the task until delivery of a signal. While suspended, *pSet* is used as the set of masked signals.

NOTE Since the ***sigsuspend*( )** function suspends thread execution indefinitely, there is no successful completion return value.

RETURNS -1, always.

ERRNO **EINTR**

SEE ALSO **sigLib**

---

sigtimedwait( )

NAME ***sigtimedwait*( )** – wait for a signal

SYNOPSIS

```
int sigtimedwait
    (
    const sigset_t *        pSet,    /* the signal mask while suspended */
    struct siginfo *        pInfo,   /* return value */
    const struct timespec * pTimeout
    )
```

DESCRIPTION The function ***sigtimedwait*( )** selects the pending signal from the set specified by *pSet*. If multiple signals in *pSet* are pending, it will remove and return the lowest numbered one. If no signal in *pSet* is pending at the time of the call, the task will be suspend until one of the signals in *pSet*become pending, it is interrupted by an unblocked caught signal, or until the time interval specified by *pTimeout* has expired. If *pTimeout* is NULL, then the timeout interval is forever.

If the *pInfo* argument is non-NULL, the selected signal number is stored in the **si_signo** member, and the cause of the signal is stored in the **si_code** member. If the signal is a queued signal, the value is stored in the **si_value** member of *pInfo*; otherwise the content of **si_value** is undefined.

The following values are defined in **signal.h** for **si_code**:

SI_USER
the signal was sent by the ***kill*( )** function.

SI_QUEUE
the signal was sent by the ***sigqueue*( )** function.

SI_TIMER
the signal was generated by the expiration of a timer set by ***timer_settime*( )**.

SI_ASYNCIO
the signal was generated by the completion of an asynchronous I/O request.

SI_MESGQ
the signal was generated by the arrival of a message on an empty message queue.

The function ***sigtimedwait*( )** provides a synchronous mechanism for tasks to wait for 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

2

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 CLSIP options parameters *compressEnable* and *compressAllow* determine support for TCP/IP header compression. If *compressAllow* is TRUE (1), then CSLIP will be enabled only if a CSLIP type packet is received by this device. If *compressEnable* is TRUE (1), then CSLIP compression will be 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.

The *pAnchor* parameter is the local address by which the local CPU may access the shared memory anchor.

The *maxInputPkts* parameter specifies the maximum number of incoming shared memory packets which may be queued to this CPU at one time.

The *intType*, *intArg1*, *intArg2*, and *intArg3* parameters allow a CPU to announce the method by which it is to be notified of input packets which have been queued to it.

The *ticksPerBeat* parameter specifies the 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.

If *numLoan* is non-zero, it specifies the number of shared memory packets available to be loaned out.

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 is the global address of shared memory added to the partition. The memory area pointed to by *pPool* must be in the same address space as the shared memory anchor and shared memory pool.

poolSize is the size in bytes of shared memory added to the partition.

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS OK, or ERROR if access to the shared memory system partition fails.

ERRNO **S_smObjLib_LOCK_TIMEOUT**

SEE ALSO **smMemLib**

smMemCalloc( )

NAME *smMemCalloc*( ) – allocate memory for 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 routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS A pointer to the block, or NULL if the memory cannot be allocated.

ERRNO **S_memLib_NOT_ENOUGH_MEMORY**
S_smObjLib_LOCK_TIMEOUT

SEE ALSO **smMemLib**

smMemFindMax( )

NAME *smMemFindMax*( ) – find 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 routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS The size (in bytes) of the largest available block, or ERROR if the attempt to access the partition fails.

ERRNO **S_smObjLib_LOCK_TIMEOUT**

SEE ALSO **smMemLib**

---

smMemFree( )

NAME *smMemFree*( ) – free a shared memory system partition block of memory (VxMP Opt.)

SYNOPSIS

```
STATUS smMemFree
    (
    void * ptr /* pointer to block of memory to be freed */
    )
```

DESCRIPTION This routine takes a block of memory previously allocated with *smMemMalloc*( ) or *smMemCalloc*( ) and returns it to the free shared memory system pool.

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

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS OK, or ERROR if the block is invalid.

ERRNO **S_memLib_BLOCK_ERROR**
S_smObjLib_LOCK_TIMEOUT

SEE ALSO **smMemLib**, *smMemMalloc*( ), *smMemCalloc*( )

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 routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS

A pointer to the block, or NULL if the memory cannot be allocated.

ERRNO

S_memLib_NOT_ENOUGH_MEMORY
S_smObjLib_LOCK_TIMEOUT

SEE ALSO

smMemLib

smMemOptionsSet( )

NAME

smMemOptionsSet( ) – set 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 library manual entry for **smMemLib**.

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS OK or ERROR.

ERRNO **S_smObjLib_LOCK_TIMEOUT**

SEE ALSO **smMemLib**

---

smMemRealloc( )

NAME *smMemRealloc*( ) – reallocate 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 routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS A pointer to the new block of memory, or NULL if the reallocation cannot be completed.

ERRNO **S_memLib_NOT_ENOUGH_MEMORY**
S_memLib_BLOCK_ERROR
S_smObjLib_LOCK_TIMEOUT

SEE ALSO **smMemLib**

smMemShow( )

NAME

smMemShow( ) – show the shared memory system partition blocks and statistics (VxMP Opt.)

SYNOPSIS

```
void smMemShow
    (
    int type /* 0 = statistics, 1 = statistics & list */
    )
```

DESCRIPTION

This routine displays the total amount of free space in the shared memory system partition, including the number of blocks, the average block size, and the maximum block size. It also shows the number of blocks currently allocated, and the average allocated block size.

If *type* is 1, it displays a list of all the blocks in the free list of the shared memory system partition.

WARNING

This routine locks access to the shared memory system partition while displaying the information. This can compromise the access time to the partition from other CPUs in the system. Generally, this routine is used for debugging purposes only.

EXAMPLE

```
-> smMemShow 1
FREE LIST:
  num    addr       size
  --- ---------- ----------
    1   0x4ffef0        264
    2   0x4fef18       1700
SUMMARY:
    status        bytes    blocks   ave block max block
--------------- --------- -------- ---------- ----------
        current
           free      1964        2        982       1700
          alloc      2356        1       2356          -
     cumulative
          alloc      2620        2       1310          -
value = 0 = 0x0
```

AVAILABILITY

This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS

N/A

SEE ALSO

smMemShow, **windsh**, *Tornado User's Guide: Shell*

smNameAdd( )

NAME

smNameAdd( ) – add a name to the shared memory name database (VxMP Opt.)

SYNOPSIS

```
STATUS smNameAdd
    (
    char * name,  /* name string to enter in database */
    void * value, /* value associated with name */
    int    type   /* type associated with name */
    )
```

DESCRIPTION

This routine adds a name of specified object type and value to the shared memory objects name database.

The *name* parameter is an arbitrary null-terminated string with a maximum of 20 characters, including EOS.

By convention, *type* values of less than 0x1000 are reserved by VxWorks; all other values are user definable. The following types are predefined in **smNameLib.h** :

T_SM_SEM_B	0	shared binary semaphore
T_SM_SEM_C	1	shared counting semaphore
T_SM_MSG_Q	2	shared message queue
T_SM_PART_ID	3	shared memory Partition
T_SM_BLOCK	4	shared memory allocated block

A name can be entered only once in the database, but there can be more than one name associated with an object ID.

AVAILABILITY

This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS

OK, or ERROR if there is insufficient memory for *name* to be allocated, if *name* is already in the database, or if the database is already full.

ERRNO

S_smNameLib_NOT_INITIALIZED
S_smNameLib_NAME_TOO_LONG
S_smNameLib_NAME_ALREADY_EXIST
S_smNameLib_DATABASE_FULL
S_smObjLib_LOCK_TIMEOUT

SEE ALSO

smNameLib, **smNameShow**

smNameFind( )

NAME

smNameFind( ) – look up a shared memory object by name (VxMP Opt.)

SYNOPSIS

```
STATUS smNameFind
    (
    char *   name,    /* name to search for */
    void * * pValue,  /* pointer where to return value */
    int *    pType,   /* pointer where to return object type */
    int      waitType /* NO_WAIT or WAIT_FOREVER */
    )
```

DESCRIPTION

This routine searches the shared memory objects name database for an object matching a specified *name*. If the object is found, its value and type are copied to the addresses pointed to by *pValue* and *pType*. The value of *waitType* can be one of the following:

NO_WAIT (0)
The call returns immediately, even if *name* is not in the database.

WAIT_FOREVER (-1)
The call returns only when *name* is available in the database. If *name*is not already in, the database is scanned periodically as the routine waits for *name* to be entered.

AVAILABILITY

This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS

OK, or ERROR if the object is not found, if *name* is too long, or the wait type is invalid.

ERRNO

S_smNameLib_NOT_INITIALIZED
S_smNameLib_NAME_TOO_LONG
S_smNameLib_NAME_NOT_FOUND
S_smNameLib_INVALID_WAIT_TYPE
S_smObjLib_LOCK_TIMEOUT

SEE ALSO

smNameLib, **smNameShow**

smNameFindByValue( )

NAME

smNameFindByValue( ) – look up a shared memory object by value (VxMP Opt.)

SYNOPSIS

```
STATUS smNameFindByValue
    (
    void * value,   /* value to search for */
    char * name,    /* pointer where to return name */
    int *  pType,   /* pointer where to return object type */
    int    waitType /* NO_WAIT or WAIT_FOREVER */
    )
```

DESCRIPTION

This routine searches the shared memory name database for an object matching a specified value. If the object is found, its name and type are copied to the addresses pointed to by *name* and *pType*. The value of *waitType* can be one of the following:

NO_WAIT (0)
The call returns immediately, even if the object value is not in the database.

WAIT_FOREVER (-1)
The call returns only when the object value is available in the database.

AVAILABILITY

This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS

OK, or ERROR if *value* is not found or if the wait type is invalid.

ERRNO

S_smNameLib_NOT_INITIALIZED
S_smNameLib_VALUE_NOT_FOUND
S_smNameLib_INVALID_WAIT_TYPE
S_smObjLib_LOCK_TIMEOUT

SEE ALSO

smNameLib, **smNameShow**

smNameRemove( )

NAME

smNameRemove( ) – remove an object from the shared memory objects name database (VxMP Opt.)

SYNOPSIS

```
STATUS smNameRemove
    (
    char * name /* name of object to remove */
    )
```

DESCRIPTION

This routine removes an object called *name* from the shared memory objects name database.

AVAILABILITY

This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS

OK, or ERROR if the object name is not in the database or if *name* is too long.

ERRNO

S_smNameLib_NOT_INITIALIZED
S_smNameLib_NAME_TOO_LONG
S_smNameLib_NAME_NOT_FOUND
S_smObjLib_LOCK_TIMEOUT

SEE ALSO

smNameLib, **smNameShow**

smNameShow( )

NAME

smNameShow( ) – show the contents of the shared memory objects name database (VxMP Opt.)

SYNOPSIS

```
STATUS smNameShow
    (
    int level /* information level */
    )
```

DESCRIPTION

This routine displays the names, values, and types of objects stored in the shared memory objects name database. Predefined types are shown, using their ASCII representations; all other types are printed in hexadecimal.

2

The *level* parameter defines the level of database information displayed. If *level* is 0, only statistics on the database contents are displayed. If *level* is greater than 0, then both statistics and database contents are displayed.

WARNING This routine locks access to the shared memory objects name database while displaying its contents. This can compromise the access time to the name database from other CPUs in the system. Generally, this routine is used for debugging purposes only.

EXAMPLE

```
-> smNameShow
Names in Database  Max : 30  Current : 6  Free : 24
-> smNameShow 1
Names in Database  Max : 30  Current : 6  Free : 24
Name                Value         Type
---------------- ----------- -------------
inputImage        0x802340    SM_MEM_BLOCK
ouputImage        0x806340    SM_MEM_BLOCK
imagePool         0x802001    SM_MEM_PART
imageInSem        0x8e0001    SM_SEM_B
imageOutSem       0x8e0101    SM_SEM_C
actionQ           0x8e0201    SM_MSG_Q
userObject        0x8e0400    0x1b0
```

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS OK, or ERROR if the name facility is not initialized.

ERRNO **S_smNameLib_NOT_INITIALIZED**
S_smObjLib_LOCK_TIMEOUT

SEE ALSO **smNameShow**, **smNameLib**

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. The *unit* parameter specifies the backplane unit number.

The *pAnchor* parameter is the local address by which the local CPU may access the shared memory anchor.

The *maxInputPkts* parameter specifies the maximum number of incoming shared memory packets which may be queued to this CPU at one time.

The *intType*, *intArg1*, *intArg2*, and *intArg3* parameters 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**

2

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 bytes, 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 when the configuration macro **INCLUDE_SM_OBJ** is defined.

This routine will complete the attach process only if and when the shared memory has been initialized by the master CPU. If the shared memory is not recognized as active within the timeout period (10 minutes), this routine returns ERROR.

The ***smObjAttach*( )** routine connects the shared memory objects handler to the shared memory interrupt. Note that this interrupt may be shared between the shared memory network driver and the shared memory objects facility when both are used at the same time.

WARNING Once a CPU has attached itself to the shared memory objects facility, it cannot be detached. Since the shared memory network driver and the shared memory objects facility use the same low-level attaching mechanism, a CPU cannot be detached from a shared memory network driver if the CPU also uses shared memory objects.

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS OK, or ERROR if the shared memory objects facility is not active or the number of CPUs exceeds the maximum.

ERRNO **S_smLib_INVALID_CPU_NUMBER**

SEE ALSO **smObjLib**, ***smObjSetup*( )**, ***smObjInit*( )**

---

smObjGlobalToLocal( )

NAME ***smObjGlobalToLocal*( )** – convert a global address to a local address (VxMP Opt.)

SYNOPSIS

```
void * smObjGlobalToLocal
    (
    void * globalAdrs /* global address to convert */
    )
```

DESCRIPTION This routine converts a global shared memory address *globalAdrs* to its corresponding local value. This routine does not verify that *globalAdrs*is really a valid global shared memory address.

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS The local shared memory address pointed to by *globalAdrs*.

SEE ALSO **smObjLib**

*smObjInit***( )**

NAME ***smObjInit*( )** – initialize a shared memory objects descriptor (VxMP Opt.)

SYNOPSIS

```
void smObjInit
    (
    SM_OBJ_DESC * pSmObjDesc,   /* ptr to shared memory descriptor */
    SM_ANCHOR * anchorLocalAdrs,/* shared memory anchor local adrs */
    int         ticksPerBeat,   /* cpu ticks per heartbeat */
    int         smObjMaxTries,  /* max no. of tries to obtain spinLock */
    int         intType,        /* interrupt method */
    int         intArg1,        /* interrupt argument #1 */
    int         intArg2,        /* interrupt argument #2 */
    int         intArg3         /* interrupt argument #3 */
    )
```

DESCRIPTION This routine initializes a shared memory descriptor. The descriptor must already be allocated in the CPU's local memory. Once the descriptor has been initialized by this routine, the CPU may attach itself to the shared memory area by calling ***smObjAttach*( )**.

This routine is called automatically when the configuration macro **INCLUDE_SM_OBJ** is defined.

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 routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS N/A

SEE ALSO **smObjLib**, *smObjSetup*( ), *smObjAttach*( )

---

smObjLibInit( )

NAME ***smObjLibInit*( )** – install the shared memory objects facility (VxMP Opt.)

SYNOPSIS
```
STATUS smObjLibInit (void)
```

DESCRIPTION This routine installs the shared memory objects facility. It is called automatically when the configuration macro **INCLUDE_SM_OBJ** is defined.

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS OK, or ERROR if the shared memory objects facility has already been installed.

SEE ALSO **smObjLib**

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 routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS

The global shared memory address pointed to by *localAdrs*.

SEE ALSO

smObjLib

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, when the configuration macro **INCLUDE_SM_OBJ** is defined.

Any CPU on the system backplane can use the shared memory objects facility; however, the facility must first be initialized on the master CPU. Then before other CPUs are attached to the shared memory area by ***smObjAttach*( )**, each must initialize its own shared memory objects descriptor using ***smObjInit*( )**. This mechanism is similar to the one used by the shared memory network driver.

The *smObjParams* parameter is a pointer to a structure containing the values used to describe the shared memory objects setup. This structure is defined as follows in **smObjLib.h**:

```
typedef struct sm_obj_params    /* setup parameters */
    {
    BOOL        allocatedPool;  /* TRUE if shared memory pool is malloced */
    SM_ANCHOR * pAnchor;        /* shared memory anchor                   */
    char *      smObjFreeAdrs;  /* start address of shared memory pool    */
    int         smObjMemSize;   /* memory size reserved for shared memory */
    int         maxCpus;        /* max number of CPUs in the system       */
    int         maxTasks;       /* max number of tasks using smObj        */
    int         maxSems;        /* max number of shared semaphores        */
    int         maxMsgQueues;   /* max number of shared message queues    */
    int         maxMemParts;    /* max number of shared memory partitions */
    int         maxNames;       /* max number of names of shared objects  */
    } SM_OBJ_PARAMS;
```

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS OK, or ERROR if the shared memory pool cannot hold all the requested objects or the number of CPUs exceeds the maximum.

ERRNO **S_smObjLib_TOO_MANY_CPU**
S_smObjLib_SHARED_MEM_TOO_SMALL

SEE ALSO **smObjLib**, *smObjInit*( ), *smObjAttach*( )

---

smObjShow( )

NAME *smObjShow*( ) – display the current status of shared memory objects (VxMP Opt.)

SYNOPSIS

```
STATUS smObjShow ()
```

DESCRIPTION This routine displays useful information about the current status of shared memory objects facilities.

WARNING The information returned by this routine is not static and may be obsolete by the time it is examined. This information is generally used for debugging purposes only.

EXAMPLE

```
-> smObjShow
Shared Mem Anchor Local Addr: 0x600.
```

```
Shared Mem Hdr Local Addr:    0xb1514.
Attached CPU :                5
Max Tries to Take Lock:       1
Shared Object Type    Current    Maximum  Available
-------------------- ---------- --------- ----------
Tasks                         1        20         19
Binary Semaphores             8        30         20
Counting Semaphores           2        30         20
Messages Queues               3        10          7
Memory Partitions             1         4          3
Names in Database            16       100         84
```

AVAILABILITY This routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS OK, or ERROR if no shared memory objects are initialized.

ERRNO **S_smObjLib_NOT_INITIALIZED**
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 routine is distributed as a component of the unbundled shared memory objects support option, VxMP.

RETURNS N/A

SEE ALSO **smObjLib**

sn83932EndLoad( )

NAME *sn83932EndLoad*( ) – initialize the driver and device

SYNOPSIS

```
END_OBJ * sn83932EndLoad
    (
    char * initString /* String to be parse by the driver. */
    )
```

DESCRIPTION This routine initializes the driver and the device to the operational state. All of the device specific parameters are passed in the *initString* parameter. This string must be of the format:

unit_number:*device_reg_addr*:*ivec*

These parameters are all individually described in the sn83932End man page.

RETURNS An END object pointer or NULL on error.

SEE ALSO **sn83932End**

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**

snmpMasterCleanup( )

NAME

snmpMasterCleanup( ) – free up resources after a query times out

SYNOPSIS

```
void snmpMasterCleanup
    (
    UINT_16_T reqid,  /* request Id to track state block */
    UINT_16_T options /* as mentioned above */
    )
```

DESCRIPTION

If you use *snmpMasterHandlerAsync*( ), the master agent calls this routine if the IPC layer determines that a timeout period for a query response has been exceeded. The *reqid* parameter is the same as the *requestId* value passed to the send routine. It is used to track the correct state block. The *options* parameter passes in a set of flags that control what actions the cleanup routine. Currently, there are three flags: **SA_CLEANUP_INACTIVE**, **SA_CLEANUP_TIMEOUT**, and **SA_CLEANUP_CONTINUE**. The continue and timeout flags should always be set. The inactive flag indicates that any objects associated with the subagent should be removed. Set this flag when the IPC layer determines that the subagent has stopped rather than timed out.

RETURNS

N/A

SEE ALSO

subagentLib

snmpMasterHandlerAsync( )

NAME

snmpMasterHandlerAsync( ) – process messages from the subagent asynchronously

SYNOPSIS

```
void snmpMasterHandlerAsync
    (
    OCTET_T *      pMsg,      /* pointer to the message */
    ALENGTH_T      msgl,      /* length of the message */
    IPCCOMP_T *    pIpcComp,  /* completion routine */
    IPCSEND_AS_T * pIpcSend,  /* send routine */
    IPCRCV_T *     pIpcRcv,   /* receive routine */
    IPCFREE_T *    pIpcFree,  /* free routine */
    IPCAYT_T *     pIpcAyt,   /* status check routine */
    PTR_T          ipchandle, /* ipchandle for the IPC scheme used */
    PTR_T          user_priv  /* MIB tree identifier */
    )
```

DESCRIPTION

This function provides support for an asynchronous communication scheme between the master agent and its subagents. The shipped version of WindNet SNMP does not call this function. Instead, it calls ***snmpMasterHandlerWR( )***, a function that supports a synchronous communication scheme. If you want master agents and subagents to use an asynchronous communication scheme, you must rewrite ***snmpQueMonitor( )*** to call ***snmpMasterHandlerAsync( )*** instead of ***snmpMasterHandlerWR( )***. In addition, because ***snmpMasterHandlerAsync( )*** does not return a function value, you will need to remove the ***snmpQueMonitor( )*** code that responded to the ***snmpMasterHandlerWR( )*** function value. The functionality handled by the removed code should instead be implemented in the function referenced by the *ipcComp* parameter. Use the parameters as follows:

pMsg
: Expects a pointer to an **EBUFFER_T** structure containing the data part of the message from the subagent. The message shows up on the queue as an **SA_MESSAGE_T** structure. The message expected by this parameter is contained in the **mesg** member of this structure. To extract this pointer, use **EbufferStart** macro defined in defined in **buffer.h**.

msgl
: Expects the length of the message referenced in *pMsg*. To retrieve this length value, use the **EBufferUsed** macro defined in **buffer.h**.

pIpcComp
: Expects a pointer to the completion function, which must be of the form:

```
void masterIpcComp
   (
   OCTET_T       opcode,    /* this specifies what needs to be done */
   EBUFFER_T *   ebuf,      /* reply message to be sent */
   VBL_T *       vblist,    /* list of varbinds that the message contained */
   PTR_T         ipchandle  /* subagent address */
   )
```

The master agent executes this function upon completing processing for an unsolicited control message from a subagent (primarily registration requests, although a trap from the subagent will eventually find its way to this function). Your ***masterIcpComp( )*** should be able handle things such as letting the subagent know the completion status of message it sent to the master agent.

For a registration routine, it must send the message in *ebuf* back to the subagent. This message contains the group ID of the MIB variables added to the master agent's MIB tree. The subagent needs this ID to make a deregistration request.

If you decide to support traps from subagents, this function must be able to forward the varbind list in *vblist* to the SNMP manager. In addition, it is your responsibility to acquire any values not specified in *vblist* and include it in the message you send the to the SNMP manager. Use the *opcode* to know when you are handling the completion processing for a registration request, a deregistration request, or a trap from a subagent.

For an example of an IPC completion routine, see ***masterIpcComp*( )** defined in **masterIoLib.c**.

pIpcSend

Expects a pointer to the function that method routines should use to send messages to the subagent. This function must be of the form:

```
INT_32_T masterIpcSend
   (
   EBUFFER_T *         pBuf,      /* message to be sent */
   PTR_T               ipchandle  /* address of subagent */
   UINT_16_T           reqid      /* ID for request sent */
   )
```

To make the communication between the master agent and subagent asynchronous, this send routine should send the message to the subagent and return. Eventually, a response shows up on the master agent's local queue, or the query times out. How you process a query response or a query time out is almost entirely up to you.

To process a query response, you must call ***snmpMasterQueryHandler*( )**. This function will handle the details of integrating the message from the subagent into a message to the SNMP manager.

To clean up after a send that times out, you must call ***snmpMasterCleanup*( )**. The specifics of the mechanism you use are up to you, but you will likely need to integrate the mechanism with your ***masterIpcSend*( )** routine. That is because this function gets the request ID that you will need for clean up. The request ID is a number generated internally to the SNMP master agent. It passes this value into your ***masterIpcSend*( )** using the *reqid* parameter. To clean up after a send that times out, you submit the *reqid* in a call to ***snmpMasterCleanup*( )**.

For an example of an ***masterIpcSend*( )**, see the ***masterIpcSend*( )** defined in **masterIoLib.c**.

pIpcRcv

This parameter is not used by ***snmpMasterHandlerAsync*( )** and so should be null. It is included to maintain parallelism with ***snmpMasterHandlerWR*( )**.

pIpcFree

Expects a pointer to a function of the form:

```
void masterIpcFree ( PTR_T ipchandle )
```

The master agent uses this function to free any resources it might have allocated to maintain the IPC link with the subagent. The master agent calls this function when a subagent deregisters.

pIpcAyt

Expects a pointer to the function the master agent can use to test the connection with the subagent. This function must be of the form:

```
INT_32_T masterIpcAyt ( PTR_T ipchandle )
```

For an example of such a function, see the ***masterIpcAyt*( )** defined in **masterIoLib.c**.

ipchandle
Expects a pointer to the IPC handle used to access the subagent that sent this message. In the shipped implementation, this is a pointer to a message queue.

user_priv
Expects a pointer to the MIB tree from which registration and deregistration requests want to add or delete objects or instances. If this pointer is NULL, the default MIB tree specified by mib_root_node is used.

RETURNS N/A

SEE ALSO **subagentLib**

---

snmpMasterHandlerWR( )

NAME ***snmpMasterHandlerWR*( )** – synchronous version of ***snmpMasterHandlerAsync*( )**

SYNOPSIS

```
INT_32_T snmpMasterHandlerWR
    (
    OCTET_T *   pMsg,      /* pointer to the message */
    ALENGTH_T   msgl,      /* length of the message */
    IPCSEND_T * pIpcSend,  /* send routine */
    IPCRCV_T *  pIpcRcv,   /* receive routine */
    IPCFREE_T * pIpcFree,  /* free routine */
    IPCAYT_T *  pIpcAyt,   /* status Check Routine */
    PTR_T       ipchandle, /* ipchandle for the IPC scheme used */
    EBUFFER_T * pBuf,      /* buffer to place reply in */
    VBL_T *     pVblist,   /* place to put varbinds */
    PTR_T       user_priv  /* MIB tree identifier */
    )
```

DESCRIPTION This function is called to process the control messages received from subagents when the communication method between master and subagent is synchronous.

To process a registration request, this function extracts the objects from the message and adds them as a group to the master agent's MIB tree. The actual get, test, and set methods for these objects reside in the subagent. To set up local methods for these routines, ***snmpMasterHandlerAsync*( )** uses the function referenced in *pIpcSend* and *pIpcRcv*.

The methods local to the master agent use *pIpcSend* to send queries to the subagent which locally executes the actual method routine for the object. The subagent then transmits the results back to the master agent's public queue. When the function monitoring this queue sees the query response, it transfers the message to the master agent's local queue where the *pIpcRcv* function is waiting for the response.

To process a deregistration request, this function extracts a group ID from the message and removes that group of objects from the master agent's MIB tree. It also executes the function in *pIpcFree* to free any resources allocated locally to maintain the IPC link with the deregistered subagent.

The ***snmpMasterHandlerWR( )*** routine returns information using the output parameters *pBuf* and *pVblist* and its function return value. If the returned function value indicates success, the master agent sends the message returned in *pBuf* to the subagent that sent the registration or deregistration request. If the returned value of this function indicates failure, the master agent silently drops the packet.

This function as has the ability to return an opcode value, although this functionality is unused in the shipped version of WindNet SNMP. In fact, if ***snmpMasterHandlerWR( )*** were to return an opcode, the current implementation of the master agent would silently drop the packet. The possibility of returning an opcode is supported to make it possible for you to create subagents that send traps. In this case, ***snmpMasterHandlerWR( )*** would return an opcode and a varbind list using the *pVblist* parameter. You could then rewrite ***snmpQueMonitor( )***, the master agent function that calls ***snmpMasterHandlerWR( )***, so that it responds appropriately to the returned opcode and forwards the contents of *pVblist* to the SNMP manager.

Use the ***snmpMasterHandlerWR( )*** parameters as follows:

pMsg

Expects a pointer to an **EBUFFER_T** structure containing the data part of the message from the subagent. The message shows up on the queue as an **SA_MESSAGE_T** structure. The message expected by this parameter is contained in the **mesg** member of the **SA_MESSAGE_T** structure. To extract this pointer, you can use the **EbufferStart** macro defined in defined in **buffer.h**.

msgl

Expects the length of the message referenced in *pMsg*. To retrieve this length value, use the **EBufferUsed** macro defined in **buffer.h**.

pIpcSend

Expects a pointer to the function that method routines should use to send messages to the subagent. This function must be of the form:

```
INT_32_T masterIpcSend
   (
   EBUFFER_T *        pBuf,      /* message to be sent */
   PTR_T              ipchandle  /* address of subagent */
   )
```

If ***snmpMasterHandlerWR( )*** is processing a registration request from the subagent, it associates this function pointer with the group of objects it adds to the master agent's MIB tree. The methods for those objects call this routine to send a message to the subagent to make a test, get, or set query against those variables. After using this function to send the message, the master agent then calls the function referenced in *pIpcRcv*. The *pIpcRcv* function waits on a local queue for a response from the subagent. For an example of an ***masterIpcSend*( )** routine, see the ***masterIpcSend*( )** defined in **masterIoLib.c**.

ipcRcv

Expects a pointer to a function of the form:

```
INT_32_T masterIpcRcv
   (
      EBUFFER_T *        pBuf,        /* buffer to receive message */
      PTR_T              ipchandle    /* address of subagent */
   )
```

If ***snmpMasterHandlerWR*( )** is processing a registration request from the subagent, it associates this function pointer with the group of objects it adds to the master agent's MIB tree. The methods for those objects call this routine to wait on a local queue for a response from the subagent. For an example of an ***masterIpcRcv*( )**, see the ***masterIpcRcv*( )** defined in **masterIoLib.c**.

ipcFree

Expects a pointer to a function of the form:

```
void masterIpcFree ( PTR_T ipchandle )
```

The master agent uses this function to free any resources it allocated to maintain the IPC link with the subagent. The master agent calls this function when a subagent deregisters.

pIpcAyt

Expects a pointer to the function the master agent can use to test the connection with the subagent. This function must be of the form:

```
INT_32_T masterIpcAyt ( PTR_T ipchandle )
```

For an example of such a function, see the ***masterIpcAyt*( )** defined in **masterIoLib.c**.

ipchandle

Expects a pointer to the IPC handle used to access the subagent that sent this message. In the shipped implementation, this is a pointer to a message queue.

pBuf

Expects a pointer to a previously allocated **EBUFFER_T**. This is an output parameter that ***snmpMasterHandlerWR*( )** uses this to return a reply packet, if one is generated. For example, if ***snmpMasterHandlerWR*( )** successfully processes a registration request, it writes a message to the **EBUFFER_T** at *pBuf*. This message contains the group ID for the objects just added to the master agent's MIB tree. When control returns from ***snmpMasterHandlerWR*( )**, you must transmit this message back to the subagent, which will store the group ID for use in a deregistration request. In the current implementation, ***snmpQueMonitor*( )** already handles this for you.

pVblist

Expects a pointer to a previously allocated **VBL_T**. The intended use of this parameter is to provide an output vehicle for the varbind list received in a trap message from a subagent. Because of the application-dependent nature of traps, the shipped implementation of ***snmpQueMonitor*( )** just drops the packet. However, if you want to support traps from your subagents, you can modify ***snmpQueMonitor*( )** to check the returned value of ***snmpMasterHandlerWR*( )** to watch for a trap message. You can then use ***snmpIoTrapSend*( )** to forward the trap message in *pVblist* to the SNMP manager.

user_priv

Expects a pointer to the MIB tree from which registration and deregistration requests want to add or delete objects or instances. If this pointer is NULL, the default MIB tree specified by **mib_root_node** is used.

If the message is trap request, it is the responsibility of the user code to acquire any values not specified in the trap message and to send the trap to the manager.

RETURNS

The opcode from the decoded packet or 0 or -1. An returned value of 0 indicates an error for which you should just drop the packet. A return value of -1 indicates success.

If this function returns an opcode, a value from 1 to 127, the shipped implementation just drops the packet. However, to support traps from the subagent, you could modify ***snmpQueMonitor*( )** to note a returned value of **SA_TRAP_REQUEST** and then forward the varbind list in *pVblist* to the SNMP manager.

SEE ALSO

subagentLib

snmpMasterQueryHandler( )

NAME

***snmpMasterQueryHandler*( )** – handles replies from the subagent

SYNOPSIS

```
UINT_16_T snmpMasterQueryHandler
    (
    OCTET_T * pMsg, /* pointer to the packet */
    ALENGTH_T msgl, /* length of packet */
    int       flag  /* should be 1 */
    )
```

DESCRIPTION

This routine is for use with ***snmpMasterHandlerAsync*( )**. It handles the replies to queries generated by the method routines. It decodes the message and tries to integrate the response with an outstanding packet. The *pMsg* and *msgl*parameters are pointers to the message and the length respectively. The *flag* parameter specifies whether the continuation routines should be run. This should always be set to 1.

RETURNS
The request ID if routine could decode the packet or 0 in case of error.

SEE ALSO
subagentLib

snmpMonitorSpawn( )

NAME
snmpMonitorSpawn( ) – spawn **tMonQue** to run *snmpQueMonitor*( )

SYNOPSIS
```
void snmpMonitorSpawn (void)
```

DESCRIPTION
This function spawns the **tMonQue** task to run *snmpQueMonitor*( ) a function that waits on the message queue that subagents use to leave messages for the master agent. The *snmpQueMonitor*( ) waits forever on the master agent's message queue. When message comes in, it is interpreted using an **SA_MESSAGE_T** structure, which is defined in **ipcLib.h** as:

```
typedef struct SA_MESSAGE_S
   {
   int             msgType;
   MSG_Q_ID        saId;
   EBUFFER_T       mesg;
   } SA_MESSAGE_T;
```

A switch internal to *snmpQueMonitor*( ) handles the message according to the value of the **msgType** member.

If the message type is **CALL_QUERY_HANDLER**, the message is a response to a query from the master agent. The buffer referenced in the **mesg** is then transferred to the local message queue monitored by **tSnmpd**, where a *masterIpcRcv*( ) routine is waiting for a query response from a subagent.

If the message type is **CALL_REG_HANDLER**, the message is a control message such as a registration request, a deregistration request, or a trap. To respond to such requests, *snmpQueMonitor*( ) passes the buffer in **mesg**on to *snmpMasterHandlerWR*( ).

If the message in the buffer passed to *snmpMasterHandlerWR*( ) is not correctly formed, the returned function value indicates failure and *snmpQueMonitor*( ) drops the packet.

If the buffer passed to *snmpMasterHandlerWR*( ) is a correctly formed registration request, *snmpMasterHandlerWR*( ) adds the specified objects to the master agent's MIB tree. If the buffer contains a correctly formed deregistration request, *snmpMasterHandlerWR*( ) removes the specified objects from the master agent's MIB tree. In both cases the returned value of *snmpMasterHandlerWR*( ) indicates success and its *rbuf* parameter contains a message that *snmpQueMonitor*( ) forwards to the subagent that sent the message.

In the case of a successful registration request, the message sent to the subagent contains a group ID for the objects just added to the master agent's MIB tree. When the subagent deregisters itself, it includes this ID in its deregistration message to the master agent. It also uses this group ID when it must register instances of the object just registered.

If the buffer passed to ***snmpMasterHandlerWR*( )** contains a trap, the returned function value is **SA_TRAP_REQUEST**, the value extracted from the **opcode2** member of the header associated with the message. The message itself (minus the header) is a varbind list. It is returned using the *vbl* parameter. The current implementation of ***snmpQueMonitor*( )** just drops this message. However, you can rewrite ***snmpQueMonitor*( )** to make a ***snmpIoTrapSend*( )** that forwards the varbind list to the SNMP manager. Likewise, you can implement appropriate responses to other **opcode2** values. Currently, **subagent.h** defines symbolic constants for opcodes 1 through 12 (with opcode 11, **SA_TRAP_REQUEST**, reserved for trap requests). If necessary you are free to use the remaining opcodes for message types specific to your implementation.

If your transport needs require that you rewrite **masterIoLib** to use an IPC other than message queues, you might need to modify this function, which is called from ***snmpIoMain*( )** just before a call to ***snmpIoBody*( )**. For example, if you use sockets as your IPC between the SNMP master agent and its subagents, **tSnmpd** could monitor the socket connection with the SNMP manager as well as the socket connections with the SNMP subagents.

ASYNCHRONOUS COMMUNICATION

The shipped version of ***snmpQueMonitor*( )** uses ***snmpMasterHandlerWR*( )** and thus processes messages asynchronously. However, if necessary, you can rewrite ***snmpQueMonitor*( )** to call ***snmpMasterHandlerAsync*( )** instead. For more information on ***snmpMasterHandlerAsync*( )**, see its reference entry.

RETURNS N/A

SEE ALSO **masterIoLib**

---

snmpSaHandlerAsync( )

NAME ***snmpSaHandlerAsync*( )** – asynchronous message processing routine for the subagent

SYNOPSIS

```
void snmpSaHandlerAsync
    (
    OCTET_T *           pMsg,      /* message from the master-agent */
    ALENGTH_T           msglength, /* length of message in octets */
    PTR_T               root,      /* root of mib tree */
    SA_IO_COMPLETE_T *  pIoComp,   /* IO completion routine */
```

```
SA_ERR_COMPLETE_T * pErrComp,  /* error completion routine */
SA_REG_COMPLETE_T * pRegComp,  /* registration complete routine */
PTR_T               cookie     /* cookie */
)
```

2

DESCRIPTION

It decodes the message in *pMsg* and responds appropriately, which can include testing, getting, and setting variables. After the message is processed, ***snmpSaHandlerAsync*( )** then calls whichever completion routine is appropriate.

pMsg
Expects pointer to an octet string containing the message from the master agent.

msglength
Expects the length of the message.

root
Expects a pointer to the root of the subagent's MIB tree. If *root* is NULL, the default **mib_root_node** is used.

pIoComp
Expects a pointer to the function ***snmpSaHandlerAsync*( )** should call after it has processed the message from the master agent. This routine should be able to send a response to the master agent, if necessary. This function must handle the building, encoding, transmission of the response to the master agent. This function must be of the form:

```
void SA_IO_COMPLETE_T(PTR_T pktp, SA_HEADER_T *hdr_blk, PTR_T cookie)
```

When the subagent calls this routine, it uses the *pktp* parameter to pass in a pointer to the data to be sent to the master agent. It uses the *hdr_blk* parameter to pass in a pointer to the header to be included with the packet. It uses the *cookie* parameter to pass in the *cookie*specified in the call to ***snmpSaHandlerAsync*( )**. You can use this *cookie*to carry information specific to your environment and application.

pErrComp
Expects a pointer to the function ***snmpSaHandlerAsync*( )** should call if it cannot generate an appropriate response to a message from the master agent. This function must be of the form:

```
void SA_ERR_COMPLETE_T(int error_code, PTR_T cookie)
```

The *error_code* passes in one of the following error codes:

SA_GEN_ERROR
SA_UNKNOWN_VERSION
SA_UNKNOWN_OPCODE1
SA_UNKNOWN_OPCODE2
SA_UNKNOWN_ENCODING
SA_DECODE_FAILURE
SA_ENCODE_FAILURE
SA_UNKNOWN_NODE

SA_UNKNOWN_TAG
SA_UNKNOWN_GRP
SA_SHORT_MSG
SA_IPC_ERROR
SA_LOCK_ERROR
SA_NODE_ERROR
SA_MEMORY_ERROR
SA_UNSUPPORTED_TYPE
SA_NO_SAVED_PACKET

The *cookie* parameter passes in the *cookie* specified in the call to ***snmpSaHandlerAsync*( )**. You can use this *cookie* to carry information specific to your environment and application.

pRegComp

Expects a pointer to the function ***snmpSaHandlerAsync*( )** should call in response to a registration completion message from the master agent. If successful, this message should contain a group ID for the MIB variables that the registration request added to the master agent's MIB tree. The subagent needs this ID when it comes time to deregister and remove those variables from the master agent's MIB tree. This function must be of the form:

```
void SA_REG_COMPLETE_T ( INT_32_T ecode, SA_HEADER_T *hdr_blk,
                         VBL_T *vblp, PTR_T cookie )
```

This completion routine expects an error code in *ecode*, a header block in *hdr_blk*, a list of nodes at *vblp*, and the *cookie* passed into the ***snmpSaHandlerAsync*( )**.

cookie

Expects a pointer that you can use to pass data unchanged to the functions you specified in the *pIoComp*, *pErrComp*, and *pRegComp* functions.

RETURNS N/A

SEE ALSO **subagentLib**

---

snmpSaHandlerCleanup( )

NAME ***snmpSaHandlerCleanup*( )** – cleanup routine for subagent

SYNOPSIS

```
void snmpSaHandlerCleanup
    (
    PTR_T         pPkt, /* pointer to the packet */
    SA_HEADER_T * pHdr  /* header block */
    )
```

DESCRIPTION This routine is called by the IO completion routine if it detects an error. It either frees or arranges to free any resources that might have been allocated for processing a query from the master agent. The information at *pPkt* and *pHdr* is passed unchanged into the completion routine.

RETURNS N/A

SEE ALSO **subagentLib**

---

snmpSaHandlerContinue( )

NAME *snmpSaHandlerContinue*( ) – subagent continuation function

SYNOPSIS

```
void snmpSaHandlerContinue
    (
    SNMP_PKT_T * pPkt /* pointer to the SNMP packet */
    )
```

DESCRIPTION This routine is similar to ***snmpdContinue*( )**. Method routines that do not complete their tasks before returning should arrange to have this routine called when the task is finished. This routine should not be called if you call ***snmpSaHandlerWR*( )**. The *pPkt* parameter expects a pointer to the packet. If **SNMP_CONTINUE_REENTRANT** is installed, this routine will attempt to release the per-packet write lock.

RETURNS N/A

SEE ALSO **subagentLib**

---

snmpSaHandlerFinish( )

NAME *snmpSaHandlerFinish*( ) – encode packet for subagent IO completion

SYNOPSIS

```
INT_32_T snmpSaHandlerFinish
    (
    PTR_T         pkt,  /* pointer to the packet */
    SA_HEADER_T * pHdr, /* header block */
    EBUFFER_T *   pBuf  /* buffer to place the result in */
    )
```

DESCRIPTION
This routine encodes the packet at *pkt* and the header block at *pHdr*. If *pBuf* is empty, this routine tries to allocate space. If it cannot or if the space provided is too small, an error is returned.

RETURNS
0 on success, or a non-zero value on failure.

SEE ALSO
subagentLib

snmpSaHandlerWR( )

NAME
***snmpSaHandlerWR*( )** – provide ***snmpSaHandlerAsync*( )** functionality synchronously

SYNOPSIS
```
INT_32_T snmpSaHandlerWR
    (
    OCTET_T *     pMsg,    /* message from the master-agent */
    ALENGTH_T     msgl,    /* kength of message in octets */
    EBUFFER_T *   pBuf,    /* buffer to hold reply packet */
    SA_HEADER_T * pHdr,    /* place for header structure */
    VBL_T *       pVblist, /* place for vblist */
    PTR_T         root     /* root of mib tree */
    )
```

DESCRIPTION
This routine puts a synchronous shell around ***snmpSaHandlerAsync*( )**. Like ***snmpSaHandlerAsync*( )**, this function can decode a message from the master agent. If the message is a query against a variable in the subagent's MIB tree, ***snmpSaHandlerWR*( )** processes the request and generates a response. However, ***snmpSaHandlerWR*( )** does not handle the completion processing for the message that would have been handled by the *pIoComp*, *pErrComp*, and *pRegComp* routines specified as input to ***snmpSaHandlerAsync*( )**.

Instead, it uses its returned function value to indicate that status of the message processing and uses *pBuf*, *pHdr*, and *pVblist* as output parameters if that status requires additional processing on your part. For example, if the message was a successfully processed query, the response data is included in *pVblist* and a header is included in *pHdr*, but that response is not yet encoded in a packet or transmitted back to the master agent. In ***snmpSaHandlerAsyn*( )**, all that would normally be handled in the *pIoComp* routine. Effectively, you must now call your *pIoComp*routine explicitly.

pMsg
Expects a pointer to the message, an octet string, from the master agent.

msgl
Expects the length of the message starting at *pMsg*.

pBuf
: Expects a pointer to a previously allocate **EBUFFER_T** into which this function can write a response, if any. In some cases (if **opcode1**is **SA_QUERY_REQUEST**), instead of indicating an error in the returned value of ***snmpSaHandlerWR*( )**, the error is encoded into this message. This is done for errors more appropriately handled by the SNMP manager.

pHdr
: Expects a pointer to a previously allocated **SA_HEADER_T** structure into which this function can writer header block information, if necessary. If **hdr_blk.sa_error** is non-zero, other members might not contain valid data.

pVblist
: Expects a pointer to a previously allocated **VBL_T** structure into which this function can write the list of nodes found in the original message from the master agent.

root
: Expects a pointer to the root of the subagent's MIB tree. If *root*is NULL, the default **mib_root_node** is used.

RETURNS

0 on success, or a positive value indicating an error. For return code values, see **subagent.h**. Using these values as a switch, you should call one of the functions you would have specified for *pIoComp*, *pErrComp*, or *pRegComp* in a call to ***snmpSaHandlerAsync*( )**.

SEE ALSO

subagentLib

---

snmpSaInit( )

NAME

***snmpSaInit*( )** – initialize the subagent

SYNOPSIS

```
PTR_T snmpSaInit
    (
    PTR_T              saId,     /* ipchandle for socket/queue */
    PTR_T              sa_root,  /* pointer to mib root node */
    SA_REG_COMPLETE_T saRegComp /* registration complete routine */
    )
```

DESCRIPTION

Call this routine to initialize an SNMP subagent. Internally, this routine creates an IPC mechanism for receiving messages from the master agent and then spawns a task to run ***snmpSaMonitor*( )**, a function that monitors the IPC mechanism created by ***snmpSaInit*( )**. As input, ***snmpSaInit*( )** takes the parameters: *saId*, *sa_root*, and *saRegComp*.

saId

Expects a null. In most functions in this library, an *saId* parameter is a pointer to the IPC mechanism used to pass messages to the subagent. However, the IPC mechanism is first created internally to this function. Thus, this *saId* parameter is not actually used for input nor is it an output parameter. It is included for parallelism with other functions in this library.

sa_root

This parameter provides a pointer to the MIB tree for this subagent.

saRegComp

Use this routine to pass in a pointer to the function that ***snmpSaHandlerAsync*( )** should execute in response to a registration status message from the master agent. If the registration was successful, the response contains a group ID for the MIB variables registered with the master agent. You will need this group ID when it comes time to deregister this SNMP subagent, or when you need to register instances of the object just registered.

Although this function sets up the IPC mechanism and spawns the task that is effectively the SNMP subagent, this routine does not actually register the subagent with the master agent. The details of how and when one does that are entirely dependent upon the nature of the system you are designing. Thus, no generic registration utility is provided. For more information on sending a registration request to the master agent, see the description of ***hdrBlkBuild*( )**.

RETURNS A pointer to the IPC mechanism created within this function, or NULL on failure.

SEE ALSO **saIoLib**

snmpSubEncode( )

NAME ***snmpSubEncode*( )** – encode a packet for transmission to master agent or subagent

SYNOPSIS

```
INT_32_T snmpSubEncode
    (
    VBL_T *       pVblist,  /* varbindlist to be encoded */
    SA_HEADER_T * pHdr,     /* header block structure */
    SA_DEMUX_T *  pDemuxer, /* demuxer structure */
    EBUFFER_T *   pBuf      /* buffer to place result in */
    )
```

DESCRIPTION This routine encodes a memory-resident varbind list. The result is a buffer containing a message ready for transmission. Most of the arguments are values to be encoded into the buffer.

pVblist

Expects a pointer to a **VBL_T** structure containing the list of the varbinds to be encoded in the message. In a control message, the varbinds identify the nodes or instances to be added or removed from the master agents MIB tree. In a query message, the varbinds identify the variables to be gotten or set. In a trap message sent from a subagent to its master agent, the varbinds specify the objects to be sent in a trap message to the SNMP manager. A trap message from a subagent follows the SNMPv2 trap style. Thus, the first object in the list must always be **sysUpTime**. The second object must be a **snmpTrapOID.0** whose value is the administratively assigned name of the notification.

pHdr

Expects a pointer to a **SA_HEADER_T** structure containing all the items that go into the message header.

pDemuxer

Expects a pointer to an **SA_DEMUX_T** structure containing all the information the subagent might need to demux the packet. That is, to determine the time and space contexts for this request. In a v1 request, the string part of the demuxer is the community string and the object ID is unused. In a v2 request, the string is the local entity string from the context and the Object ID is the local time ID from the context.

pBuf

Expects a pointer to an **EBUFFER_T** structure into which ***snmpSubEncode*( )** can write the encoded packet. If *pBuf* references a previously allocated **EBUFFER_T** structure, ***snmpSubEncode*( )** uses that space. Otherwise, ***snmpSubEncode*( )** tries to the necessary space.

RETURNS 0, if successful (that is, the structure at pBuf is ready for transmission); 1, if there is an illegal or unknown argument; 2, if there is insufficient buffer space at pBuf or space cannot be allocated.

SEE ALSO **subagentLib**

***sntpcTimeGet*( )**

NAME ***sntpcTimeGet*( )** – retrieve the current time from a remote source

SYNOPSIS

```
STATUS sntpcTimeGet
    (
    char *            pServerAddr, /* server IP address or hostname */
    u_int             timeout,     /* timeout interval in ticks */
    struct timespec * pCurrTime    /* storage for retrieved time value */
    )
```

DESCRIPTION

This routine stores the current time as reported by an SNTP/NTP server in the location indicated by *pCurrTime*. The reported time is first converted to the elapsed time since January 1, 1970, 00:00, GMT, which is the base value used by UNIX systems. If *pServerAddr* is NULL, the routine listens for messages sent by an SNTP/NTP server in broadcast mode. Otherwise, this routine sends a request to the specified SNTP/NTP server and extracts the reported time from the reply. In either case, an error is returned if no message is received within the interval specified by *timeout*. Typically, SNTP/NTP servers operating in broadcast mode send update messages every 64 to 1024 seconds. An infinite timeout value is specified by **WAIT_FOREVER**.

RETURNS

OK, or ERROR if unsuccessful.

ERRNO

S_sntpcLib_INVALID_PARAMETER
S_sntpcLib_INVALID_ADDRESS

SEE ALSO

sntpcLib

---

sntpsClockSet( )

NAME

sntpsClockSet( ) – assign a routine to access the reference clock

SYNOPSIS

```
STATUS sntpsClockSet
    (
    FUNCPTR pClockHookRtn /* new interface to reference clock */
    )
```

DESCRIPTION

This routine installs a hook routine that is called to access the reference clock used by the SNTP server. This hook routine must use the following interface:

```
    STATUS sntpsClockHook (int request, void *pBuffer);
```

The hook routine should copy one of three settings used by the server to construct outgoing NTP messages into *pBuffer* according to the value of the *request* parameter. If the requested setting is available, the installed routine should return OK (or ERROR otherwise).

This routine calls the given hook routine with the *request* parameter set to **SNTPS_ID** to get the 32-bit reference identifier in the format specified in RFC 1769. It also calls the hook routine with *request* set to **SNTPS_RESOLUTION** to retrieve a 32-bit value containing the clock resolution in nanoseconds. That value will be used to determine the 8-bit signed integer indicating the clock precision (according to the format specified in RFC 1769). Other library routines will set the *request* parameter to **SNTPS_TIME** to retrieve the current 64-bit NTP timestamp from *pBuffer* in host byte order. The routine

***sntpsNsecToFraction*( )** will convert a value in nanoseconds to the format required for the NTP fractional part.

RETURNS OK or ERROR.

ERRNO N/A

SEE ALSO **sntpsLib**

sntpsConfigSet( )

NAME *sntpsConfigSet*( ) – change SNTP server broadcast settings

SYNOPSIS

```
STATUS sntpsConfigSet
    (
    int    setting, /* configuration option to change */
    void * pValue   /* new value for parameter */
    )
```

DESCRIPTION This routine alters the configuration of the SNTP server when operating in broadcast mode. A *setting* value of **SNTPS_DELAY** interprets the contents of *pValue* as the new 16-bit broadcast interval. When *setting* equals **SNTPS_ADDRESS**, *pValue* should provide the string representation of an IP broadcast or multicast address (for example, "224.0.1.1"). Any changed settings will take effect after the current broadcast interval is completed and the corresponding NTP message is sent.

RETURNS OK or ERROR.

ERRNO **S_sntpsLib_INVALID_PARAMETER**

SEE ALSO **sntpsLib**

*sntpsNsecToFraction***( )**

NAME

*sntpsNsecToFraction***( )** – convert portions of a second to NTP format

SYNOPSIS

```
ULONG sntpsNsecToFraction
    (
    ULONG nsecs /* nanoseconds to convert to binary fraction */
    )
```

DESCRIPTION

This routine is provided for convenience in fulfilling an **SNTPS_TIME** request to the clock hook. It converts a value in nanoseconds to the fractional part of the NTP timestamp format. The routine is not designed to convert non-normalized values greater than or equal to one second. Although the NTP time format provides a precision of about 200 pico-seconds, rounding errors in the conversion process decrease the accuracy as the input value increases. In the worst case, only the 24 most significant bits are valid, which reduces the precision to tenths of a micro-second.

RETURNS

Value for NTP fractional part in host-byte order.

ERRNO

N/A

SEE ALSO

sntpsLib

*so***( )**

NAME

*so***( )** – single-step, but step over a subroutine

SYNOPSIS

```
STATUS so
    (
    int task /* task to step; 0 = use default */
    )
```

DESCRIPTION

This routine single-steps a task that is stopped at a breakpoint. However, if the next instruction is a JSR or BSR, *so***( )** breaks at the instruction following the subroutine call instead. To execute, enter:

```
-> so [task]
```

If *task* is omitted or zero, the last task referenced is assumed.

SEE ALSO

dbgLib, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

socket( )

NAME

socket( ) – open a socket

SYNOPSIS

```
int socket
    (
    int domain,  /* address family (for example, AF_INET) */
    int type,    /* SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW */
    int protocol /* socket protocol (usually 0) */
    )
```

DESCRIPTION

This routine opens a socket and returns a socket descriptor. The socket descriptor is passed to the other socket routines to identify the socket. The socket descriptor is a standard I/O system file descriptor (fd) and can be used with the ***close*( )**, ***read*( )**, ***write*( )**, and ***ioctl*( )** routines.

Available socket types include:

SOCK_STREAM
Specifies a connection-based (stream) socket.

SOCK_DGRAM
Specifies a datagram (UDP) socket.

SOCK_RAW
Specifies a raw socket.

RETURNS

A socket descriptor, or ERROR.

SEE ALSO

sockLib

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 */
    )
```

DESCRIPTION This routine copies a formatted string to a specified buffer, which is null-terminated. Its function and syntax are otherwise identical to ***printf*( )**.

RETURNS The number of characters copied to *buffer*, not including the NULL terminator.

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

---

spy( )

NAME *spy*( ) – begin periodic task activity reports

SYNOPSIS

```
void spy
    (
    int freq,       /* reporting freq in sec, 0 = default of 5 */
    int ticksPerSec /* interrupt clock freq, 0 = default of 100 */
    )
```

DESCRIPTION This routine collects task activity data and periodically runs ***spyReport*( )**. Data is gathered *ticksPerSec* times per second, and a report is made every *freq* seconds. If *freq* is zero, it defaults to 5 seconds. If *ticksPerSec* is omitted or zero, it defaults to 100.

This routine spawns ***spyTask*( )** to do the actual reporting.

It is not necessary to call ***spyClkStart*( )** before running ***spy*( )**.

RETURNS N/A

SEE ALSO **usrLib**, **spyLib**, ***spyClkStart*( )**, ***spyTask*( )**, *VxWorks Programmer's Guide: Target Shell*

---

spyClkStart( )

NAME *spyClkStart*( ) – start collecting task activity data

SYNOPSIS

```
STATUS spyClkStart
    (
    int intsPerSec /* timer interrupt freq, 0 = default of 100 */
    )
```

DESCRIPTION This routine begins data collection by enabling the auxiliary clock interrupts at a frequency of *intsPerSec* interrupts per second. If *intsPerSec* is omitted or zero, the frequency will be 100. Data from previous collections is cleared.

RETURNS OK, or ERROR if the CPU has no auxiliary clock, or if task create and delete hooks cannot be installed.

SEE ALSO **usrLib**, **spyLib**, ***sysAuxClkConnect*( )**, *VxWorks Programmer's Guide: Target Shell*

---

spyClkStop( )

NAME *spyClkStop*( ) – stop collecting task activity data

SYNOPSIS `void spyClkStop (void)`

DESCRIPTION This routine disables the auxiliary clock interrupts. Data collected remains valid until the next ***spyClkStart*( )** call.

RETURNS N/A

SEE ALSO **usrLib**, **spyLib**, ***spyClkStart*( )**, *VxWorks Programmer's Guide: Target Shell*

---

spyHelp( )

NAME *spyHelp*( ) – display task monitoring help menu

SYNOPSIS `void spyHelp (void)`

DESCRIPTION This routine displays a summary of **spyLib** utilities:

```
spyHelp                       Print this list
spyClkStart [ticksPerSec]     Start task activity monitor running
                                at ticksPerSec ticks per second
spyClkStop                    Stop collecting data
spyReport                     Prints display of task activity
                                statistics
spyStop                       Stop collecting data and reports
spy     [freq[,ticksPerSec]]  Start spyClkStart and do a report
                                every freq seconds
ticksPerSec defaults to 100.  freq defaults to 5 seconds.
```

RETURNS N/A

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

spyLibInit( )

NAME *spyLibInit*( ) – initialize task cpu utilization tool package

SYNOPSIS

```
void spyLibInit (void)
```

DESCRIPTION This routine initializes the task cpu utilization tool package. If the configuration macro **INCLUDE_SPY** is defined, it is called by the root task, ***usrRoot*( )**, in **usrConfig.c**.

RETURNS N/A

SEE ALSO **spyLib**, **usrLib**

spyReport( )

NAME *spyReport*( ) – display task activity data

SYNOPSIS

```
void spyReport (void)
```

DESCRIPTION This routine reports on data gathered at interrupt level for the amount of CPU time utilized by each task, the amount of time spent at interrupt level, the amount of time spent in the kernel, and the amount of idle time. Time is displayed in ticks and as a percentage, and the data is shown since both the last call to ***spyClkStart*( )** and the last ***spyReport*( )**. If no interrupts have occurred since the last ***spyReport*( )**, nothing is displayed.

RETURNS N/A

SEE ALSO **usrLib**, **spyLib**, ***spyClkStart*( )**, *VxWorks Programmer's Guide: Target Shell*

spyStop( )

NAME *spyStop*( ) – stop spying and reporting

SYNOPSIS

```
void spyStop (void)
```

DESCRIPTION This routine calls *spyClkStop*( ). Any periodic reporting by *spyTask*( ) is terminated.

RETURNS N/A

SEE ALSO **usrLib**, **spyLib**, *spyClkStop*( ), *spyTask*( ), *VxWorks Programmer's Guide: Target Shell*

spyTask( )

NAME *spyTask*( ) – run periodic task activity reports

SYNOPSIS

```
void spyTask
    (
    int freq /* reporting frequency, in seconds */
    )
```

DESCRIPTION This routine is spawned as a task by *spy*( ) to provide periodic task activity reports. It prints a report, delays for the specified number of seconds, and repeats.

RETURNS N/A

SEE ALSO **usrLib**, **spyLib**, *spy*( ), *VxWorks Programmer's Guide: Target Shell*

sqrt( )

NAME *sqrt*( ) – compute a non-negative square root (ANSI)

SYNOPSIS

```
double sqrt
    (
    double x /* value to compute the square root of */
    )
```

DESCRIPTION This routine computes the non-negative square root of *x* in double precision. A domain error occurs if the argument is negative.

INCLUDE FILES **math.h**

RETURNS The double-precision square root of *x* or 0 if *x* is negative.

ERRNO **EDOM**

SEE ALSO **ansiMath**, **mathALib**

---

sqrtf( )

NAME *sqrtf*( ) – compute a non-negative square root (ANSI)

SYNOPSIS

```
float sqrtf
    (
    float x /* value to compute the square root of */
    )
```

DESCRIPTION This routine returns the non-negative square root of *x* in single precision.

INCLUDE FILES **math.h**

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

SEE ALSO **mathALib**

---

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
*sramDrv***( )** – 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 */
    )
```

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 composed of white-space character(s) is executed by reading input up to the first non-white-space character (which remains unread), or until no more characters can be read.

A directive that is an ordinary multibyte character is executed by reading the next characters of the stream. If one of the characters differs from one comprising the directive, the directive fails, and the differing and subsequent characters remain unread.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

Input white-space characters (as specified by the ***isspace*( )** function) are skipped, unless the specification includes a **[**, **c**, or **n** specifier.

An input item is read from the stream, unless the specification includes an **n** specifier. An input item is defined as the longest matching sequence of input characters, unless that exceeds a specified field width, in which case it is the initial subsequence of that length in the sequence. The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails: this condition is a matching failure, unless an error prevented input from the stream, in which case it is an input failure.

Except in the case of a **%** specifier, the input item is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a *****, the result of the conversion is placed in the object pointed to by the first argument following the *fmt* argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

The following conversion specifiers are valid:

d

Matches an optionally signed decimal integer whose format is the same as expected for the subject sequence of the ***strtol*( )** function with the value 10 for the *base* argument. The corresponding argument should be a pointer to **int**.

i

Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the ***strtol*( )** function with the value 0 for the *base* argument. The corresponding argument should be a pointer to **int**.

o

Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the ***strtoul*( )** function with the value 8 for the *base* argument. The corresponding argument should be a pointer to **unsigned int**.

u

Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the ***strtoul*( )** function with the value 10 for the *base* argument. The corresponding argument should be a pointer to **unsigned int**.

x

Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the ***strtoul*( )** function with the value 16 for the *base* argument. The corresponding argument should be a pointer to **unsigned int**.

e, **f**, **g**

Match an optionally signed floating-point number, whose format is the same as expected for the subject string of the ***strtod*( )** function. The corresponding argument should be a pointer to **float**.

s

Matches a sequence of non-white-space characters. The corresponding argument should be a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which will be added automatically.

[

Matches a non-empty sequence of characters from a set of expected characters (the

scanset). The corresponding argument should be a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which is added automatically. The conversion specifier includes all subsequent character in the format string, up to and including the matching right bracket (**]**). The characters between the brackets (the **scanlist**) comprise the scanset, unless the character after the left bracket is a circumflex (**^**) in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with "[]" or "[^]", the right bracket character is in the scanlist and the next right bracket character is the matching right bracket that ends the specification; otherwise the first right bracket character is the one that ends the specification.

c

Matches a sequence of characters of the number specified by the field width (1 if no field width is present in the directive). The corresponding argument should be a pointer to the initial character of an array large enough to accept the sequence. No null character is added.

p

Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the %p conversion of the ***fprintf*( )** function. The corresponding argument should be a pointer to a pointer to **void**. VxWorks defines its pointer input field to be consistent with pointers written by the ***fprintf*( )** function ("0x" hexadecimal notation). If the input item is a value converted earlier during the same program execution, the pointer that results should compare equal to that value; otherwise the behavior of the %p conversion is undefined.

n

No input is consumed. The corresponding argument should be a pointer to **int** into which the number of characters read from the input stream so far by this call to ***sscanf*( )** is written. Execution of a %n directive does not increment the assignment count returned when ***sscanf*( )** completes execution.

%

Matches a single **%**; no conversion or assignment occurs. The complete conversion specification is %%.

If a conversion specification is invalid, the behavior is undefined.

The conversion specifiers **E**, **G**, and **X** are also valid and behave the same as **e**, **g**, and **x**, respectively.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (if any) is terminated with an input failure.

If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream. Trailing white space (including new-line characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the %n directive.

INCLUDE FILES **fioLib.h**

RETURNS The number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure; or EOF if an input failure occurs before any conversion.

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

---

st16552DevInit( )

NAME *st16552DevInit*( ) – initialise an ST16552 channel

SYNOPSIS

```
void st16552DevInit
    (
    ST16552_CHAN * pChan
    )
```

DESCRIPTION This routine initialises some **SIO_CHAN** function pointers and then resets the chip in a quiescent state. Before this routine is called, the BSP must already have initialised all the device addresses, etc. in the **ST16552_CHAN** structure.

RETURNS N/A

SEE ALSO **st16552Sio**

---

st16552Int( )

NAME *st16552Int*( ) – interrupt level processing

SYNOPSIS

```
void st16552Int
    (
    ST16552_CHAN * pChan /* ptr to struct describing channel */
    )
```

DESCRIPTION This routine handles interrupts from the UART.

RETURNS N/A

SEE ALSO **st16552Sio**

st16552IntEx( )

NAME *st16552IntEx*( ) – miscellaneous interrupt processing

SYNOPSIS

```
void st16552IntEx
    (
    ST16552_CHAN * pChan /* ptr to struct describing channel */
    )
```

DESCRIPTION This routine handles miscellaneous interrupts on the UART.

RETURNS N/A

SEE ALSO **st16552Sio**

st16552IntRd( )

NAME *st16552IntRd*( ) – handle a receiver interrupt

SYNOPSIS

```
void st16552IntRd
    (
    ST16552_CHAN * pChan /* ptr to struct describing channel */
    )
```

DESCRIPTION This routine handles read interrupts from the UART.

RETURNS N/A

SEE ALSO **st16552Sio**

st16552IntWr( )

NAME

st16552IntWr( ) – handle a transmitter interrupt

SYNOPSIS

```
void st16552IntWr
    (
    ST16552_CHAN * pChan /* ptr to struct describing channel */
    )
```

DESCRIPTION

This routine handles write interrupts from the UART.

RETURNS

N/A

SEE ALSO

st16552Sio

st16552MuxInt( )

NAME

st16552MuxInt( ) – multiplexed interrupt level processing

SYNOPSIS

```
void st16552MuxInt
    (
    ST16552_MUX * pMux /* ptr to struct describing multiplexed chans */
    )
```

DESCRIPTION

This routine handles multiplexed interrupts from the DUART. It assumes that channels 0 and 1 are connected so that they produce the same interrupt.

RETURNS

N/A

SEE ALSO

st16552Sio

2

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. It is called automatically when this show facility is configured into VxWorks using either of the following methods:

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

– If you use the Tornado project facility, select **INCLUDE_STDIO_SHOW**.

RETURNS OK, or ERROR if an error occurs installing the file pointer show routine.

SEE ALSO **ansiStdio**

strcat( )

NAME *strcat*( ) – concatenate one string to another (ANSI)

SYNOPSIS

```
char * strcat
    (
    char *       destination, /* string to be appended to */
    const char * append       /* string to append to destination */
    )
```

DESCRIPTION This routine appends a copy of string *append* to the end of string *destination*. The resulting string is null-terminated.

INCLUDE FILES **string.h**

RETURNS A pointer to *destination*.

SEE ALSO **ansiString**

strchr( )

NAME *strchr*( ) – find the first occurrence of a character in a string (ANSI)

SYNOPSIS

```
char * strchr
    (
    const char * s, /* string in which to search */
    int          c  /* character to find in string */
    )
```

DESCRIPTION This routine finds the first occurrence of character *c*in string *s*. The terminating null is considered to be part of the string.

INCLUDE FILES **string.h**

RETURNS The address of the located character, or NULL if the character is not found.

SEE ALSO **ansiString**

2

strcmp( )

NAME *strcmp*( ) – compare two strings lexicographically (ANSI)

SYNOPSIS

```
int strcmp
    (
    const char * s1, /* string to compare */
    const char * s2  /* string to compare s1 to */
    )
```

DESCRIPTION This routine compares string *s1* to string *s2* lexicographically.

INCLUDE FILES **string.h**

RETURNS An integer greater than, equal to, or less than 0, according to whether *s1* is lexicographically greater than, equal to, or less than *s2*, respectively.

SEE ALSO **ansiString**

strcoll( )

NAME *strcoll*( ) – compare two strings as appropriate to **LC_COLLATE** (ANSI)

SYNOPSIS

```
int strcoll
    (
    const char * s1, /* string 1 */
    const char * s2  /* string 2 */
    )
```

DESCRIPTION This routine compares two strings, both interpreted as appropriate to the **LC_COLLATE** category of the current locale.

INCLUDE FILES **string.h**

RETURNS An integer greater than, equal to, or less than zero, according to whether string *s1* is greater than, equal to, or less than string *s2* when both are interpreted as appropriate to the current locale.

SEE ALSO **ansiString**

strcpy( )

NAME *strcpy*( ) – copy one string to another (ANSI)

SYNOPSIS

```
char * strcpy
    (
    char *       s1, /* string to copy to */
    const char * s2  /* string to copy from */
    )
```

DESCRIPTION This routine copies string *s2* (including EOS) to string *s1*.

INCLUDE FILES **string.h**

RETURNS A pointer to *s1*.

SEE ALSO **ansiString**

strcspn( )

NAME *strcspn*( ) – return the string length up to the first character from a given set (ANSI)

SYNOPSIS

```
size_t strcspn
    (
    const char * s1, /* string to search */
    const char * s2  /* set of characters to look for in s1 */
    )
```

DESCRIPTION This routine computes the length of the maximum initial segment of string *s1* that consists entirely of characters not included in string *s2*.

INCLUDE FILES **string.h**

RETURNS The length of the string segment.

SEE ALSO **ansiString**, *strpbrk*( ), *strspn*( )

strerror( )

NAME

strerror( ) – map an error number to an error string (ANSI)

SYNOPSIS

```
char * strerror
    (
    int errcode /* error code */
    )
```

DESCRIPTION

This routine maps the error number in *errcode* to an error message string. It returns a pointer to a static buffer that holds the error string.

This routine is not reentrant. For a reentrant version, see ***strerror_r*( )**.

INCLUDE

string.h

RETURNS

A pointer to the buffer that holds the error string.

SEE ALSO

ansiString, *strerror_r*( )

strerror_r( )

NAME

strerror_r( ) – map an error number to an error string (POSIX)

SYNOPSIS

```
STATUS strerror_r
    (
    int    errcode, /* error 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*( )

strftime( )

NAME

strftime( ) – convert broken-down time into a formatted string (ANSI)

SYNOPSIS

```
size_t strftime
    (
    char *              s,      /* string array */
    size_t              n,      /* maximum size of array */
    const char *        format, /* format of output string */
    const struct tm *   tptr    /* broken-down time */
    )
```

DESCRIPTION

This routine formats the broken-down time in *tptr* based on the conversion specified in the string *format*, and places the result in the string *s*.

The format is a multibyte character sequence, beginning and ending in its initial state. The *format* string consists of zero or more conversion specifiers and ordinary multibyte characters. A conversion specifier consists of a % character followed by a character that determines the behavior of the conversion. All ordinary multibyte characters (including the terminating NULL character) are copied unchanged to the array. If copying takes place between objects that overlap, the behavior is undefined. No more than *n* characters are placed into the array.

Each conversion specifier is replaced by appropriate characters as described in the following list. The appropriate characters are determined by the **LC_TIME** category of the current locale and by the values contained in the structure pointed to by *tptr*.

%a the locale's abbreviated weekday name.

%A the locale's full weekday name.

%b the locale's abbreviated month name.

%B the locale's full month name.

%c the locale's appropriate date and time representation.

%d the day of the month as decimal number (01-31).

%H the hour (24-hour clock) as a decimal number (00-23).

%I the hour (12-hour clock) as a decimal number (01-12).

%j the day of the year as decimal number (001-366).

%m the month as a decimal number (01-12).

%M the minute as a decimal number (00-59).

%P the locale's equivalent of the AM/PM designations associated with a 12-hour clock.

%S	the second as a decimal number (00-59).
%U	the week number of the year (first Sunday as 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**

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*( )

2

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 subject sequence is empty or does not have the expected form, no conversion is performed; the value of *s* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

INCLUDE FILES

stdlib.h

RETURNS

The converted value, if any. If no conversion could be performed, it returns zero. If the correct value is outside the range of representable values, it returns plus or minus **HUGE_VAL** (according to the sign of the value), and stores the value of the macro **ERANGE** in **errno**. If the correct value would cause underflow, it returns zero and stores the value of the macro **ERANGE** in **errno**.

SEE ALSO

ansiStdlib

---

strtok( )

NAME

strtok( ) – break down a string into tokens (ANSI)

SYNOPSIS

```
char * strtok
    (
    char *       string,   /* string */
    const char * separator /* separator indicator */
    )
```

DESCRIPTION

A sequence of calls to this routine breaks the string *string* into a sequence of tokens, each of which is delimited by a character from the string *separator*. The first call in the sequence has *string* as its first argument, and is followed by calls with a null pointer as their first argument. The separator string may be different from call to call.

The first call in the sequence searches *string* for the first character that is not contained in the current separator string. If the character is not found, there are no tokens in *string* and ***strtok*( )** returns a null pointer. If the character is found, it is the start of the first token.

***strtok*( )** then searches from there for a character that is contained in the current separator string. If the character is not found, the current token expands to the end of the string pointed to by *string*, and subsequent searches for a token will return a null pointer. If the character is found, it is overwritten by a null character, which terminates the current token. ***strtok*( )** saves a pointer to the following character, from which the next search for a token will start. (Note that because the separator character is overwritten by a null character, the input string is modified as a result of this call.)

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

The implementation behaves as if ***strtok*( )** is called by no library functions.

REENTRANCY This routine is not reentrant; the reentrant form is ***strtok_r*( )**.

INCLUDE FILES **string.h**

RETURNS A pointer to the first character of a token, or a NULL pointer if there is no token.

SEE ALSO **ansiString**, ***strtok_r*( )**

---

strtok_r( )

NAME ***strtok_r*( )** – break down a string into tokens (reentrant) (POSIX)

SYNOPSIS

```
char * strtok_r
    (
    char *       string,     /* string to break into tokens */
    const char * separators, /* the separators */
    char * *     ppLast      /* pointer to serve as string index */
    )
```

DESCRIPTION This routine considers the null-terminated string *string* as a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *separators*. The argument *ppLast* points to a user-provided pointer which in turn points to the position within *string*at which scanning should begin.

In the first call to this routine, *string* points to a null-terminated string; *separators* points to a null-terminated string of separator characters; and *ppLast* points to a NULL pointer. The function returns a pointer to the first character of the first token, writes a null character into *string* immediately following the returned token, and updates the pointer to which *ppLast* points so that it points to the first character following the null written into *string*. (Note that because the separator character is overwritten by a null character, the input string is modified as a result of this call.)

In subsequent calls *string* must be a NULL pointer and *ppLast* must be unchanged so that subsequent calls will move through the string *string*, returning successive tokens until no tokens remain. The separator string *separators* may be different from call to call. When no token remains in *string*, a NULL pointer is returned.

INCLUDE FILES **string.h**

RETURNS A pointer to the first character of a token, or a NULL pointer if there is no token.

SEE ALSO **ansiString**, ***strtok*( )**

strtol( )

NAME ***strtol*( )** – convert a string to a long integer (ANSI)

SYNOPSIS

```
long strtol
    (
    const char * nptr,   /* string to convert */
    char * *     endptr, /* ptr to final string */
    int          base    /* radix */
    )
```

DESCRIPTION This routine converts the initial portion of a string *nptr* to **long int**representation. First, it decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by ***isspace*( )**); a subject sequence resembling an integer represented in some radix determined by the value of *base*; and a final string of one or more unrecognized characters, including the terminating NULL character of the input string. Then, it attempts to convert the subject sequence to an integer number, and returns the result.

If the value of *base* is zero, the expected form of the subject sequence is that of an integer constant, optionally preceded by a plus or minus sign, but not including an integer suffix. If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base* optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from a (or A) through to z (or Z) are ascribed the values 10 to 35; only letters whose ascribed values are less than *base* are 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 defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of *base* is zero, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it is used as the *base* for conversion, ascribing to each latter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer.

In other than the "C" locale, additional implementation-defined subject sequence forms may be accepted. VxWorks supports only the "C" locale; it assumes that the upper- and lower-case alphabets and digits are each contiguous.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer.

INCLUDE FILES **stdlib.h**

RETURNS The converted value, if any. If no conversion could be performed, it returns zero. If the correct value is outside the range of representable values, it returns **LONG_MAX** or **LONG_MIN** (according to the sign of the value), and stores the value of the macro **ERANGE** in **errno**.

SEE ALSO **ansiStdlib**

***strtoul*( )**

NAME ***strtoul*( )** – convert a string to an unsigned long integer (ANSI)

SYNOPSIS

```
ulong_t strtoul
    (
    const char * nptr,   /* string to convert */
    char * *     endptr, /* ptr to final string */
    int          base    /* radix */
    )
```

DESCRIPTION

This routine converts the initial portion of a string *nptr* to **unsigned long int** representation. First, it decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by ***isspace*( )**); a subject sequence resembling an unsigned integer represented in some radix determined by the value *base*; and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then, it attempts to convert the subject sequence to an unsigned integer, and returns the result.

If the value of *base* is zero, the expected form of the subject sequence is that of an integer constant, optionally preceded by a plus or minus sign, but not including an integer suffix. If the value of *base* is between 2 and 36, the expected form of the subject sequence is a 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 defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of *base* is zero, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it is used as the *base* for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

In other than the "C" locale, additional implementation-defined subject sequence forms may be accepted. VxWorks supports only the "C" locale; it assumes that the upper- and lower-case alphabets and digits are each contiguous.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

INCLUDE FILES

stdlib.h

RETURNS

The converted value, if any. If no conversion could be performed it returns zero. If the correct value is outside the range of representable values, it returns **ULONG_MAX**, and stores the value of the macro **ERANGE** in *errno*.

SEE ALSO

ansiStdlib

strxfrm( )

NAME

strxfrm( ) – transform up to *n* characters of *s2* into *s1* (ANSI)

SYNOPSIS

```
size_t strxfrm
    (
    char *       s1, /* string out */
    const char * s2, /* string in */
    size_t       n   /* size of buffer */
    )
```

DESCRIPTION

This routine transforms string *s2* and places the resulting string in *s1*. The transformation is such that if ***strcmp*( )** is applied to two transformed strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the ***strcoll*( )** function applied to the same two original strings. No more than *n* characters are placed into the resulting *s1*, including the terminating null character. If *n* is zero, *s1* is permitted to be a NULL pointer. If copying takes place between objects that overlap, the behavior is undefined.

INCLUDE FILES

string.h

RETURNS

The length of the transformed string, not including the terminating null character. If the value is *n* or more, the contents of *s1* are indeterminate.

SEE ALSO

ansiString, *strcmp*( ), *strcoll*( )

swab( )

NAME

swab( ) – swap bytes

SYNOPSIS

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

DESCRIPTION

This routine gets the specified number of bytes from *source*, exchanges the adjacent even and odd bytes, and puts them in *destination*. The buffers *source* and *destination* should not overlap.

NOTE

On some CPUs, ***swab*( )** will cause an exception if the buffers are unaligned. In such cases, use ***uswab*( )** for unaligned swaps.

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 the configuration macro **INCLUDE_SYM_TBL** is defined, *symLibInit*( ) is called by the root task, *usrRoot*( ), in **usrConfig.c**.

RETURNS OK, or ERROR if the library could not be initialized.

SEE ALSO **symLib**

symRemove( )

NAME *symRemove*( ) – remove a symbol from a symbol table

SYNOPSIS

```
STATUS symRemove
    (
    SYMTAB_ID symTblId, /* symbol tbl to remove symbol from */
    char *    name,     /* name of symbol to remove */
    SYM_TYPE  type      /* type of symbol to remove */
    )
```

DESCRIPTION This routine removes a symbol of matching name and type from a specified symbol table. The symbol is deallocated if found. Note that VxWorks symbols in a standalone VxWorks image (where the symbol table is linked in) cannot be removed.

RETURNS OK, or ERROR if the symbol is not found or could not be deallocated.

SEE ALSO **symLib**

2

symSyncLibInit( )

NAME *symSyncLibInit*( ) – initialize host/target symbol table synchronization

SYNOPSIS

```
void symSyncLibInit ()
```

DESCRIPTION This routine initializes host/target symbol table synchronization. To enable synchronization, it must be called before a target server is started. It is called automatically if the configuration macro **INCLUDE_SYM_TBL_SYNC** is defined.

RETURNS N/A

SEE ALSO **symSyncLib**

symSyncTimeoutSet( )

NAME *symSyncTimeoutSet*( ) – set WTX timeout

SYNOPSIS

```
UINT32 symSyncTimeoutSet
    (
    UINT32 timeout /* WTX timeout in milliseconds */
    )
```

DESCRIPTION This routine sets the WTX timeout between target server and synchronization task.

RETURNS If *timeout* is 0, the current timeout, otherwise the new timeout value in milliseconds.

SEE ALSO **symSyncLib**

symTblCreate( )

NAME *symTblCreate*( ) – create a symbol table

SYNOPSIS

```
SYMTAB_ID symTblCreate
    (
    int     hashSizeLog2, /* size of hash table as a power of 2 */
    BOOL    sameNameOk,   /* allow 2 symbols of same name & type */
```

```
    PART_ID symPartId     /* memory part ID for symbol allocation */
    )
```

DESCRIPTION This routine creates and initializes a symbol table with a hash table of a specified size. The size of the hash table is specified as a power of two. For example, if *hashSizeLog2* is 6, a 64-entry hash table is created.

If *sameNameOk* is FALSE, attempting to add a symbol with the same name and type as an already-existing symbol results in an error.

Memory for storing symbols as they are added to the symbol table will be allocated from the memory partition *symPartId*. The ID of the system memory partition is stored in the global variable **memSysPartId**, which is declared in **memLib.h**.

RETURNS Symbol table ID, or NULL if memory is insufficient.

SEE ALSO **symLib**

---

symTblDelete( )

NAME *symTblDelete*( ) – delete a symbol table

SYNOPSIS

```
STATUS symTblDelete
    (
    SYMTAB_ID symTblId /* ID of symbol table to delete */
    )
```

DESCRIPTION This routine deletes a specified symbol table. It deallocates all associated memory, including the hash table, and marks the table as invalid.

Deletion of a table that still contains symbols results in ERROR. Successful deletion includes the deletion of the internal hash table and the deallocation of memory associated with the table. The table is marked invalid to prohibit any future references.

RETURNS OK, or ERROR if the symbol table ID is invalid.

SEE ALSO **symLib**

sysAuxClkConnect( )

NAME

sysAuxClkConnect( ) – connect a routine to the auxiliary clock interrupt

SYNOPSIS

```
STATUS sysAuxClkConnect
    (
    FUNCPTR routine, /* routine called at each aux clock interrupt */
    int     arg      /* argument to auxiliary clock interrupt routine */
    )
```

DESCRIPTION

This routine specifies the interrupt service routine to be called at each auxiliary clock interrupt. It does not enable auxiliary clock interrupts.

NOTE

This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS

OK, or ERROR if the routine cannot be connected to the interrupt.

SEE ALSO

sysLib, *intConnect*( ), *sysAuxClkEnable*( ), and BSP-specific reference pages for this routine

sysAuxClkDisable( )

NAME

sysAuxClkDisable( ) – turn off auxiliary clock interrupts

SYNOPSIS

```
void sysAuxClkDisable (void)
```

DESCRIPTION

This routine disables auxiliary clock interrupts.

NOTE

This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS

N/A

SEE ALSO

sysLib, *sysAuxClkEnable*( ), and BSP-specific reference pages for this routine

sysAuxClkEnable( )

NAME *sysAuxClkEnable*( ) – turn on auxiliary clock interrupts

SYNOPSIS

```
void sysAuxClkEnable (void)
```

DESCRIPTION This routine enables auxiliary clock interrupts.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS N/A

SEE ALSO **sysLib**, *sysAuxClkConnect*( ), *sysAuxClkDisable*( ), *sysAuxClkRateSet*( ), and BSP-specific reference pages for this routine

sysAuxClkRateGet( )

NAME *sysAuxClkRateGet*( ) – get the auxiliary clock rate

SYNOPSIS

```
int sysAuxClkRateGet (void)
```

DESCRIPTION This routine returns the interrupt rate of the auxiliary clock.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS The number of ticks per second of the auxiliary clock.

SEE ALSO **sysLib**, *sysAuxClkEnable*( ), *sysAuxClkRateSet*( ), and BSP-specific reference pages for this routine

sysAuxClkRateSet( )

NAME *sysAuxClkRateSet*( ) – set the auxiliary clock rate

SYNOPSIS

```
STATUS sysAuxClkRateSet
    (
    int ticksPerSecond /* number of clock interrupts per second */
    )
```

DESCRIPTION This routine sets the interrupt rate of the auxiliary clock. It does not enable auxiliary clock interrupts.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS OK, or ERROR if the tick rate is invalid or the timer cannot be set.

SEE ALSO **sysLib**, *sysAuxClkEnable*( ), *sysAuxClkRateGet*( ), and BSP-specific reference pages for this routine

sysBspRev( )

NAME *sysBspRev*( ) – return the BSP version and revision number

SYNOPSIS

```
char * sysBspRev (void)
```

DESCRIPTION This routine returns a pointer to a BSP version and revision number, for example, 1.0/1. **BSP_REV** is concatenated to **BSP_VERSION** and returned.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS A pointer to the BSP version/revision string.

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine

sysBusIntAck( )

NAME *sysBusIntAck*( ) – acknowledge a bus interrupt

SYNOPSIS

```
int sysBusIntAck
    (
    int intLevel /* interrupt level to acknowledge */
    )
```

DESCRIPTION This routine acknowledges a specified VMEbus interrupt level.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.*

RETURNS NULL.

SEE ALSO **sysLib**, *sysBusIntGen*( ), and BSP-specific reference pages for this routine

sysBusIntGen( )

NAME *sysBusIntGen*( ) – generate a bus interrupt

SYNOPSIS

```
STATUS sysBusIntGen
    (
    int intLevel, /* bus interrupt level to generate */
    int vector    /* interrupt vector to generate (0-255) */
    )
```

DESCRIPTION This routine generates a bus interrupt for a specified level with a specified vector.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.*

RETURNS OK, or ERROR if *intLevel* is out of range or the board cannot generate a bus interrupt.

SEE ALSO **sysLib**, *sysBusIntAck*( ), and BSP-specific reference pages for this routine

2

sysBusTas( )

NAME *sysBusTas*( ) – test and set a location across the bus

SYNOPSIS

```
BOOL sysBusTas
    (
    char * adrs /* address to be tested and set */
    )
```

DESCRIPTION This routine performs a test-and-set instruction across the backplane.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

NOTE This routine is equivalent to *vxTas*( ).

RETURNS TRUE if the value had not been set but is now, or FALSE if the value was set already.

SEE ALSO **sysLib**, *vxTas*( ), and BSP-specific reference pages for this routine

sysBusToLocalAdrs( )

NAME *sysBusToLocalAdrs*( ) – convert a bus address to a local address

SYNOPSIS

```
STATUS sysBusToLocalAdrs
    (
    int    adrsSpace,  /* bus address space in which busAdrs resides */
    char * busAdrs,    /* bus address to convert */
    char * *pLocalAdrs /* where to return local address */
    )
```

DESCRIPTION This routine gets the local address that accesses a specified bus memory address.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS OK, or ERROR if the address space is unknown or the mapping is not possible.

SEE ALSO **sysLib**, *sysLocalToBusAdrs*( ), and BSP-specific reference pages for this routine

sysClkConnect( )

NAME *sysClkConnect*( ) – connect a routine to the system clock interrupt

SYNOPSIS

```
STATUS sysClkConnect
    (
    FUNCPTR routine, /* routine called at each system clock interrupt */
    int     arg      /* argument with which to call routine */
    )
```

DESCRIPTION This routine specifies the interrupt service routine to be called at each clock interrupt. Normally, it is called from ***usrRoot*( )** in **usrConfig.c** to connect ***usrClock*( )** to the system clock interrupt.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURN OK, or ERROR if the routine cannot be connected to the interrupt.

SEE ALSO **sysLib**, *intConnect*( ), *usrClock*( ), *sysClkEnable*( ), and BSP-specific reference pages for this routine

sysClkDisable( )

NAME *sysClkDisable*( ) – turn off system clock interrupts

SYNOPSIS

```
void sysClkDisable (void)
```

DESCRIPTION This routine disables system clock interrupts.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS N/A

SEE ALSO **sysLib**, *sysClkEnable*( ), and BSP-specific reference pages for this routine

sysClkEnable( )

NAME *sysClkEnable*( ) – turn on system clock interrupts

SYNOPSIS

```
void sysClkEnable (void)
```

DESCRIPTION This routine enables system clock interrupts.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS N/A

SEE ALSO **sysLib**, *sysClkConnect*( ), *sysClkDisable*( ), *sysClkRateSet*( ), and BSP-specific reference pages for this routine

sysClkRateGet( )

NAME *sysClkRateGet*( ) – get the system clock rate

SYNOPSIS

```
int sysClkRateGet (void)
```

DESCRIPTION This routine returns the system clock rate.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS The number of ticks per second of the system clock.

SEE ALSO **sysLib**, *sysClkEnable*( ), *sysClkRateSet*( ), and BSP-specific reference pages for this routine

sysClkRateSet( )

NAME *sysClkRateSet*( ) – set the system clock rate

SYNOPSIS

```
STATUS sysClkRateSet
    (
    int ticksPerSecond /* number of clock interrupts per second */
    )
```

DESCRIPTION This routine sets the interrupt rate of the system clock. It is called by ***usrRoot*( )** in **usrConfig.c**.

There may be interactions between this routine and the POSIX **clockLib**routines. Refer to the **clockLib** reference entry.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS OK, or ERROR if the tick rate is invalid or the timer cannot be set.

SEE ALSO **sysLib**, *sysClkEnable*( ), *sysClkRateGet*( ), **clockLib**, and BSP-specific reference pages for this routine

sysHwInit( )

NAME *sysHwInit*( ) – initialize the system hardware

SYNOPSIS

```
void sysHwInit (void)
```

DESCRIPTION This routine initializes various features of the board. It is called from ***usrInit*( )** in **usrConfig.c**.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

NOTE This routine should not be called directly by the user application.

RETURNS N/A

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine

sysIntDisable( )

NAME
sysIntDisable( ) – disable a bus interrupt level

SYNOPSIS
```
STATUS sysIntDisable
    (
    int intLevel /* interrupt level to disable */
    )
```

DESCRIPTION
This routine disables a specified bus interrupt level.

NOTE
This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.*

RETURNS
OK, or ERROR if *intLevel* is out of range.

SEE ALSO
sysLib, *sysIntEnable*( ), and BSP-specific reference pages for this routine

sysIntEnable( )

NAME
sysIntEnable( ) – enable a bus interrupt level

SYNOPSIS
```
STATUS sysIntEnable
    (
    int intLevel /* interrupt level to enable (1-7) */
    )
```

DESCRIPTION
This routine enables a specified bus interrupt level.

NOTE
This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.*

RETURNS
OK, or ERROR if *intLevel* is out of range.

SEE ALSO
sysLib, *sysIntDisable*( ), and BSP-specific reference pages for this routine

sysLocalToBusAdrs( )

NAME *sysLocalToBusAdrs*( ) – convert a local address to a bus address

SYNOPSIS

```
STATUS sysLocalToBusAdrs
    (
    int    adrsSpace, /* bus address space in which busAdrs resides */
    char * localAdrs, /* local address to convert */
    char * *pBusAdrs  /* where to return bus address */
    )
```

DESCRIPTION This routine gets the bus address that accesses a specified local memory address.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS OK, or ERROR if the address space is unknown or not mapped.

SEE ALSO **sysLib**, *sysBusToLocalAdrs*( ), and BSP-specific reference pages for this routine

sysMailboxConnect( )

NAME *sysMailboxConnect*( ) – connect a routine to the mailbox interrupt

SYNOPSIS

```
STATUS sysMailboxConnect
    (
    FUNCPTR routine, /* routine called at each mailbox interrupt */
    int     arg      /* argument with which to call routine */
    )
```

DESCRIPTION This routine specifies the interrupt service routine to be called at each mailbox interrupt.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS OK, or ERROR if the routine cannot be connected to the interrupt.

SEE ALSO **sysLib**, *intConnect*( ), *sysMailboxEnable*( ), and BSP-specific reference pages for this routine

sysMailboxEnable( )

NAME *sysMailboxEnable*( ) – enable the mailbox interrupt

SYNOPSIS

```
STATUS sysMailboxEnable
    (
    char * mailboxAdrs /* address of mailbox (ignored) */
    )
```

DESCRIPTION This routine enables the mailbox interrupt.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS OK, always.

SEE ALSO **sysLib**, *sysMailboxConnect*( ), and BSP-specific reference pages for this routine

sysMemTop( )

NAME *sysMemTop*( ) – get the address of the top of logical memory

SYNOPSIS

```
char *sysMemTop (void)
```

DESCRIPTION This routine returns the address of the top of memory.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS The address of the top of memory.

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine

sysModel( )

NAME
sysModel( ) – return the model name of the CPU board

SYNOPSIS
```
char *sysModel (void)
```

DESCRIPTION
This routine returns the model name of the CPU board.

NOTE
This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS
A pointer to a string containing the board name.

SEE ALSO
sysLib, and BSP-specific reference pages for this routine

sysNvRamGet( )

NAME
sysNvRamGet( ) – get the contents of non-volatile RAM

SYNOPSIS
```
STATUS sysNvRamGet
    (
    char * string, /* where to copy non-volatile RAM */
    int    strLen, /* maximum number of bytes to copy */
    int    offset  /* byte offset into non-volatile RAM */
    )
```

DESCRIPTION
This routine copies the contents of non-volatile memory into a specified string. The string will be terminated with an EOS.

NOTE
This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS
OK, or ERROR if access is outside the non-volatile RAM address range.

SEE ALSO
sysLib, *sysNvRamSet*( ), and BSP-specific reference pages for this routine

sysNvRamSet( )

NAME *sysNvRamSet*( ) – write to non-volatile RAM

SYNOPSIS

```
STATUS sysNvRamSet
    (
    char * string, /* string to be copied into non-volatile RAM */
    int    strLen, /* maximum number of bytes to copy */
    int    offset  /* byte offset into non-volatile RAM */
    )
```

DESCRIPTION This routine copies a specified string into non-volatile RAM.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS OK, or ERROR if access is outside the non-volatile RAM address range.

SEE ALSO **sysLib**, *sysNvRamGet*( ), and BSP-specific reference pages for this routine

sysPhysMemTop( )

NAME *sysPhysMemTop*( ) – get the address of the top of memory

SYNOPSIS

```
char * sysPhysMemTop (void)
```

DESCRIPTION This routine returns the address of the first missing byte of memory, which indicates the top of memory. Normally, the amount of physical memory is specified with the macro **LOCAL_MEM_SIZE**. BSPs that support run-time memory sizing do so only if the macro **LOCAL_MEM_AUTOSIZE** is defined. If not defined, then **LOCAL_MEM_SIZE** is assumed to be, and must be, the true size of physical memory.

NOTE: Do no adjust **LOCAL_MEM_SIZE** to reserve memory for application use. See *sysMemTop*( ) for more information on reserving memory.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS The address of the top of physical memory.

SEE ALSO **sysLib**, ***sysMemTop*( )**, and BSP-specific reference pages for this routine

sysProcNumGet( )

NAME *sysProcNumGet*( ) – get the processor number

SYNOPSIS

```
int sysProcNumGet (void)
```

DESCRIPTION This routine returns the processor number for the CPU board, which is set with *sysProcNumSet*( ).

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS The processor number for the CPU board.

SEE ALSO **sysLib**, ***sysProcNumSet*( )**, and BSP-specific reference pages for this routine

sysProcNumSet( )

NAME *sysProcNumSet*( ) – set the processor number

SYNOPSIS

```
void sysProcNumSet
    (
    int procNum /* processor number */
    )
```

DESCRIPTION This routine sets the processor number for the CPU board. Processor numbers should be unique on a single backplane.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS N/A

SEE ALSO **sysLib**, ***sysProcNumGet*( )**, and BSP-specific reference pages for this routine

sysScsiBusReset( )

NAME *sysScsiBusReset*( ) – assert the RST line on the SCSI bus (Western Digital WD33C93 only)

SYNOPSIS

```
void sysScsiBusReset
    (
    WD_33C93_SCSI_CTRL * pSbic /* ptr to SBIC info */
    )
```

DESCRIPTION This routine asserts the RST line on the SCSI bus, which causes all connected devices to return to a quiescent state.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS N/A

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine

sysScsiConfig( )

NAME *sysScsiConfig*( ) – system SCSI configuration

SYNOPSIS

```
STATUS sysScsiConfig (void)
```

DESCRIPTION This is an example SCSI configuration routine.

Most of the code found here is an example of how to declare a SCSI peripheral configuration. You must edit this routine to reflect the actual configuration of your SCSI bus. This example can also be found in **src/config/usrScsi.c**.

If you are just getting started, you can test your hardware configuration by defining **SCSI_AUTO_CONFIG**, which will probe the bus and display all devices found. No device should have the same SCSI bus ID as your VxWorks SCSI port (default = 7), or the same as any other device. Check for proper bus termination.

There are three configuration examples here. They demonstrate configuration of a SCSI hard disk (any type), an OMTI 3500 floppy disk, and a tape drive (any type).

Hard Disk The hard disk is divided into two 32-Mbyte partitions and a third partition with the remainder of the disk. The first partition is initialized as a dosFs device. The second and third partitions are initialized as rt11Fs devices, each with 256 directory entries.

It is recommended that the first partition (**BLK_DEV**) on a block device be a dosFs device, if the intention is eventually to boot VxWorks from the device. This will simplify the task considerably.

Floppy Disk

The floppy, since it is a removable medium device, is allowed to have only a single partition, and dosFs is the file system of choice for this device, since it facilitates media compatibility with IBM PC machines.

In contrast to the hard disk configuration, the floppy setup in this example is more intricate. Note that the ***scsiPhysDevCreate*( )** call is issued twice. The first time is merely to get a "handle" to pass to ***scsiModeSelect*( )**, since the default media type is sometimes inappropriate (in the case of generic SCSI-to-floppy cards). After the hardware is correctly configured, the handle is discarded via ***scsiPhysDevDelete*( )**, after which the peripheral is correctly configured by a second call to ***scsiPhysDevCreate*( )**. (Before the ***scsiModeSelect*( )** call, the configuration information was incorrect.) Note that after the ***scsiBlkDevCreate*( )** call, the correct values for *sectorsPerTrack* and *nHeads* must be set via ***scsiBlkDevInit*( )**. This is necessary for IBM PC compatibility.

Tape Drive

The tape configuration is also somewhat complex because certain device parameters need to turned off within VxWorks and the fixed-block size needs to be defined, assuming that the tape supports fixed blocks.

The last parameter to the ***dosFsDevInit*( )** call is a pointer to a **DOS_VOL_CONFIG** structure. By specifying NULL, you are asking ***dosFsDevInit*( )** to read this information off the disk in the drive. This may fail if no disk is present or if the disk has no valid dosFs directory. Should this be the case, you can use the ***dosFsMkfs*( )** command to create a new directory on a disk. This routine uses default parameters (see **dosFsLib**) that may not be suitable for your application, in which case you should use ***dosFsDevInit*( )** with a pointer to a valid **DOS_VOL_CONFIG** structure that you have created and initialized. If ***dosFsDevInit*( )** is used, a ***diskInit*( )** call should be made to write a new directory on the disk, if the disk is blank or disposable.

NOTE

The variable **pSbdFloppy** is global to allow the above calls to be made from the VxWorks shell, for example:

```
-> dosFsMkfs "/fd0/", pSbdFloppy
```

If a disk is new, use ***diskFormat*( )** to format it.

NOTE

This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS

OK or ERROR.

SEE ALSO

sysLib, and BSP-specific reference pages for this routine

sysScsiInit( )

NAME *sysScsiInit*( ) – initialize an on-board SCSI port

SYNOPSIS

```
STATUS sysScsiInit (void)
```

DESCRIPTION This routine creates and initializes a SCSI control structure, enabling use of the on-board SCSI port. It also connects the proper interrupt service routine to the desired vector, and enables the interrupt at the desired level.

If SCSI DMA is supported by the board and **INCLUDE_SCSI_DMA** is defined, the DMA is also initialized.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS OK, or ERROR if the control structure cannot be connected, the controller cannot be initialized, or the DMA's interrupt cannot be connected.

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine

sysSerialChanGet( )

NAME *sysSerialChanGet*( ) – get the **SIO_CHAN** device associated with a serial channel

SYNOPSIS

```
SIO_CHAN * sysSerialChanGet
    (
    int channel /* serial channel */
    )
```

DESCRIPTION This routine gets the **SIO_CHAN** device associated with a specified serial channel.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS A pointer to the **SIO_CHAN** structure for the channel, or ERROR if the channel is invalid.

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine

sysSerialHwInit( )

NAME *sysSerialHwInit*( ) – initialize the BSP serial devices to a quiesent state

SYNOPSIS

```
void sysSerialHwInit (void)
```

DESCRIPTION This routine initializes the BSP serial device descriptors and puts the devices in a quiesent state. It is called from ***sysHwInit*( )** with interrupts locked.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS N/A

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine

sysSerialHwInit2( )

NAME *sysSerialHwInit2*( ) – connect BSP serial device interrupts

SYNOPSIS

```
void sysSerialHwInit2 (void)
```

DESCRIPTION This routine connects the BSP serial device interrupts. It is called from ***sysHwInit2*( )**. Serial device interrupts could not be connected in ***sysSerialHwInit*( )** because the kernel memory allocator was not initialized at that point, and ***intConnect*( )** calls ***malloc*( )**.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this call is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS N/A

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine

sysSerialReset( )

NAME *sysSerialReset*( ) – reset all SIO devices to a quiet state

SYNOPSIS

```
void sysSerialReset (void)
```

DESCRIPTION This routine is called from ***sysToMonitor*( )** to reset all SIO device and prevent them from generating interrupts or performing DMA cycles.

NOTE This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS N/A

SEE ALSO **sysLib**, and BSP-specific reference pages for this routine

system( )

NAME *system*( ) – pass a string to a command processor (Unimplemented) (ANSI)

SYNOPSIS

```
int system
    (
    const char * string /* pointer to string */
    )
```

DESCRIPTION This function is not applicable to VxWorks.

INCLUDE FILES **stdlib.h**

RETURNS OK, always.

SEE ALSO **ansiStdlib**

sysToMonitor( )

NAME

sysToMonitor( ) – transfer control to the ROM monitor

SYNOPSIS

```
STATUS sysToMonitor
    (
    int startType /* parameter passed to ROM to tell it how to boot */
    )
```

DESCRIPTION

This routine transfers control to the ROM monitor. Normally, it is called only by ***reboot*( )**--which services **CTRL+X**--and by bus errors at interrupt level. However, in some circumstances, the user may wish to introduce a *startType* to enable special boot ROM facilities.

NOTE

This is a generic page for a BSP-specific routine; this description contains general information only. To determine if this routine is supported by your BSP, or for information specific to your BSP's version of this routine, see the reference pages for your BSP.

RETURNS

Does not return.

SEE ALSO

sysLib, and BSP-specific reference pages for this routine

tan( )

NAME

tan( ) – compute a tangent (ANSI)

SYNOPSIS

```
double tan
    (
    double x /* angle in radians */
    )
```

DESCRIPTION

This routine computes the tangent of x in double precision. The angle x is expressed in radians.

INCLUDE FILES

math.h

RETURNS

The double-precision tangent of x.

SEE ALSO

ansiMath, **mathALib**

tanf( )

NAME *tanf*( ) – compute a tangent (ANSI)

SYNOPSIS

```
float tanf
    (
    float x /* angle in radians */
    )
```

DESCRIPTION This routine returns the tangent of *x* in single precision. The angle *x* is expressed in radians.

INCLUDE FILES **math.h**

RETURNS The single-precision tangent of *x*.

SEE ALSO **mathALib**

tanh( )

NAME *tanh*( ) – compute a hyperbolic tangent (ANSI)

SYNOPSIS

```
double tanh
    (
    double x /* number whose hyperbolic tangent is required */
    )
```

DESCRIPTION This routine returns the hyperbolic tangent of *x* in double precision (IEEE double, 53 bits).

INCLUDE FILES **math.h**

RETURNS The double-precision hyperbolic tangent of *x*.

Special cases:
If *x* is NaN, ***tanh*( )** returns NaN.

SEE ALSO **ansiMath**, **mathALib**

tanhf( )

NAME *tanhf*( ) – compute a hyperbolic tangent (ANSI)

SYNOPSIS

```
float tanhf
    (
    float x /* number whose hyperbolic tangent is required */
    )
```

DESCRIPTION This routine returns the hyperbolic tangent of *x* in single precision.

INCLUDE FILES **math.h**

RETURNS The single-precision hyperbolic tangent of *x*.

SEE ALSO **mathALib**

tapeFsDevInit( )

NAME *tapeFsDevInit*( ) – associate a sequential device with tape volume functions

SYNOPSIS

```
TAPE_VOL_DESC *tapeFsDevInit
    (
    char *        volName,    /* volume name */
    SEQ_DEV *     pSeqDev,    /* pointer to sequential device info */
    TAPE_CONFIG * pTapeConfig /* pointer to tape config info */
    )
```

DESCRIPTION This routine takes a sequential device created by a device driver and defines it as a tape file system volume. As a result, when high-level I/O operations, such as *open*( ) and *write*( ), are performed on the device, the calls will be routed through **tapeFsLib**.

This routine associates **volName** with a device and installs it in the VxWorks I/O system-device table. The driver number used when the device is added to the table is that which was assigned to the tape library during *tapeFsInit*( ). (The driver number is kept in the global variable **tapeFsDrvNum**.)

The **SEQ_DEV** structure specified by **pSeqDev** contains configuration data describing the device and the addresses of the routines which are called to read blocks, write blocks, write file marks, reset the device, check device status, perform other I/O control functions (*ioctl*( )), reserve and release devices, load and unload devices, and rewind devices. These routines are not called until they are required by subsequent I/O operations. The **TAPE_CONFIG** structure is used to define configuration parameters for the **TAPE_VOL_DESC**. The configuration parameters are defined and described in **tapeFsLib.h**.

RETURNS A pointer to the volume descriptor (**TAPE_VOL_DESC**), or NULL if there is an error.

ERRNO **S_tapeFsLib_NO_SEQ_DEV, S_tapeFsLib_ILLEGAL_TAPE_CONFIG_PARM**

SEE ALSO **tapeFsLib**

---

tapeFsInit( )

NAME *tapeFsInit*( ) – initialize the tape volume library

SYNOPSIS

```
STATUS tapeFsInit ()
```

DESCRIPTION This routine initializes the tape volume library. It must be called exactly once, before any other routine in the library. Only one file descriptor per volume is assumed.

This routine also installs tape volume library routines in the VxWorks I/O system driver table. The driver number assigned to **tapeFsLib** is placed in the global variable **tapeFsDrvNum**. This number is later associated with system file descriptors opened to tapeFs devices.

To enable this initialization, simply call the routine ***tapeFsDevInit*( )**, which automatically calls ***tapeFsInit*( )** in order to initialize the tape file system.

RETURNS OK or ERROR.

SEE ALSO **tapeFsLib**

---

tapeFsReadyChange( )

NAME *tapeFsReadyChange*( ) – notify **tapeFsLib** of a change in ready status

SYNOPSIS

```
STATUS tapeFsReadyChange
    (
    TAPE_VOL_DESC * pTapeVol /* pointer to volume descriptor */
    )
```

DESCRIPTION
This routine sets the volume descriptor state to **TAPE_VD_READY_CHANGED**. It should be called whenever a driver senses that a device has come on-line or gone off-line (for example, that a tape has been inserted or removed).

After this routine has been called, the next attempt to use the volume results in an attempted remount.

RETURNS
OK if the read change status is set, or ERROR if the file descriptor is in use.

ERRNO
S_tapeFsLib_FILE_DESCRIPTOR_BUSY

SEE ALSO
tapeFsLib

tapeFsVolUnmount( )

NAME
tapeFsVolUnmount( ) – disable a tape device volume

SYNOPSIS
```
STATUS tapeFsVolUnmount
    (
    TAPE_VOL_DESC * pTapeVol /* pointer to volume descriptor */
    )
```

DESCRIPTION
This routine is called when I/O operations on a volume are to be discontinued. This is commonly done before changing removable tape. All buffered data for the volume is written to the device (if possible), any open file descriptors are marked obsolete, and the volume is marked not mounted.

Because this routine flushes data from memory to the physical device, it should not be used in situations where the tape-change is not recognized until after a new tape has been inserted. In these circumstances, use the ready-change mechanism. (See the manual entry for *tapeFsReadyChange*( ).)

This routine may also be called by issuing an *ioctl*( ) call using the **FIOUNMOUNT** function code.

RETURNS
OK, or ERROR if the routine cannot access the volume.

ERRNO
S_tapeFsLib_VOLUME_NOT_AVAILABLE, S_tapeFsLib_FILE_DESCRIPTOR_BUSY, S_tapeFsLib_SERVICE_NOT_AVAILABLE

SEE ALSO
tapeFsLib, *tapeFsReadyChange*( )

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.

ERRNO **S_intLib_NOT_ISR_CALLABLE**, **EINTR**

SEE ALSO **taskLib**

---

taskDelete( )

NAME ***taskDelete*( )** – delete a task

SYNOPSIS
```
STATUS taskDelete
    (
    int tid /* task ID of task to delete */
    )
```

DESCRIPTION This routine causes a specified task to cease to exist and deallocates the stack and **WIND_TCB** memory resources. Upon deletion, all routines specified by ***taskDeleteHookAdd*( )** will be called in the context of the deleting task. This routine is the companion routine to ***taskSpawn*( )**.

RETURNS OK, or ERROR if the task cannot be deleted.

ERRNO **S_intLib_NOT_ISR_CALLABLE**, **S_objLib_OBJ_DELETED** **S_objLib_OBJ_UNAVAILABLE**, **S_objLib_OBJ_ID_ERROR**

SEE ALSO **taskLib**, **excLib**, ***taskDeleteHookAdd*( )**, ***taskSpawn*( )**, *VxWorks Programmer's Guide: Basic OS*

taskDeleteForce( )

NAME

taskDeleteForce( ) – delete a task without restriction

SYNOPSIS

```
STATUS taskDeleteForce
    (
    int tid /* task ID of task to delete */
    )
```

DESCRIPTION

This routine deletes a task even if the task is protected from deletion. It is similar to *taskDelete*( ). Upon deletion, all routines specified by *taskDeleteHookAdd*( ) will be called in the context of the deleting task.

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.

ERRNO

S_intLib_NOT_ISR_CALLABLE, S_objLib_OBJ_DELETED, S_objLib_OBJ_UNAVAILABLE, S_objLib_OBJ_ID_ERROR

SEE ALSO

taskLib, *taskDeleteHookAdd*( ), *taskDelete*( )

taskDeleteHookAdd( )

NAME

taskDeleteHookAdd( ) – add a routine to be called at every task delete

SYNOPSIS

```
STATUS taskDeleteHookAdd
    (
    FUNCPTR deleteHook /* routine to be called when a task is deleted */
    )
```

DESCRIPTION

This routine adds a specified routine to a list of routines that will be called whenever a task is deleted. The routine should be declared as follows:

```
    void deleteHook
        (
```

```
    WIND_TCB *pTcb        /* pointer to deleted task's WIND_TCB */
    )
```

RETURNS OK, or ERROR if the table of task delete routines is full.

SEE ALSO **taskHookLib**, *taskDeleteHookDelete*( )

---

taskDeleteHookDelete( )

NAME *taskDeleteHookDelete*( ) – delete a previously added task delete routine

SYNOPSIS

```
STATUS taskDeleteHookDelete
    (
    FUNCPTR deleteHook /* routine to be deleted from list */
    )
```

DESCRIPTION This routine removes a specified routine from the list of routines to be called at each task delete.

RETURNS OK, or ERROR if the routine is not in the table of task delete routines.

SEE ALSO **taskHookLib**, *taskDeleteHookAdd*( )

---

taskDeleteHookShow( )

NAME *taskDeleteHookShow*( ) – show the list of task delete routines

SYNOPSIS

```
void taskDeleteHookShow (void)
```

DESCRIPTION This routine shows all the delete routines installed in the task delete hook table, in the order in which they were installed. Note that the delete routines will be run in reverse of the order in which they were installed.

RETURNS N/A

SEE ALSO **taskHookShow**, *taskDeleteHookAdd*( )

taskHookInit( )

NAME *taskHookInit*( ) – initialize task hook facilities

SYNOPSIS

```
void taskHookInit (void)
```

DESCRIPTION This routine is a NULL routine called to configure the task hook package into the system. It is called automatically if the configuration macro **INCLUDE_TASK_HOOKS** is defined.

RETURNS N/A

SEE ALSO **taskHookLib**

taskHookShowInit( )

NAME *taskHookShowInit*( ) – initialize the task hook show facility

SYNOPSIS

```
void taskHookShowInit (void)
```

DESCRIPTION This routine links the task hook show facility into the VxWorks system. It is called automatically when the task hook show facility is configured into VxWorks using either of the following methods:

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

– If you use the Tornado project facility, select **INCLUDE_TASK_HOOK_SHOW**.

RETURNS N/A

SEE ALSO **taskHookShow**

2

taskIdDefault( )

NAME *taskIdDefault*( ) – set the default task ID

SYNOPSIS

```
int taskIdDefault
    (
    int tid /* user supplied task ID; if 0, return default */
    )
```

DESCRIPTION This routine maintains a global default task ID. This ID is used by libraries that want to allow a task ID argument to take on a default value when one is not explicitly supplied.

If *tid* is not zero (i.e., the user did specify a task ID), the default ID is set to that value, and that value is returned. If *tid* is zero (i.e., the user did not specify a task ID), the default ID is not changed and its value is returned. Thus the value returned is always the last task ID the user specified.

RETURNS The most recent non-zero task ID.

SEE ALSO **taskInfo**, **dbgLib**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

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.

ERRNO **S_objLib_OBJ_ID_ERROR**

SEE ALSO **taskLib**

2

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.

ERRNO

S_intLib_NOT_ISR_CALLABLE, S_objLib_OBJ_ID_ERROR

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.

ERRNO **S_objLib_OBJ_ID_ERROR**, **S_intLib_NOT_ISR_CALLABLE**

SEE ALSO **taskLib**, ***taskUnlock*( )**, ***intLock*( )**, ***taskSafe*( )**, ***semTake*( )**

taskName( )

NAME *taskName*( ) – get the name associated with a task ID

SYNOPSIS

```
char *taskName
    (
    int tid /* ID of task whose name is to be found */
    )
```

DESCRIPTION This routine returns a pointer to the name of a task of a specified ID, if the task has a name. If the task has no name, it returns an empty string.

RETURNS A pointer to the task name, or NULL if the task ID is invalid.

SEE ALSO **taskInfo**

taskNameToId( )

NAME *taskNameToId*( ) – look up the task ID associated with a task name

SYNOPSIS

```
int taskNameToId
    (
    char * name /* task name to look up */
    )
```

DESCRIPTION This routine returns the ID of the task matching a specified name. Referencing a task in this way is inefficient, since it involves a search of the task list.

RETURNS The task ID, or ERROR if the task is not found.

ERRNO **S_taskLib_NAME_NOT_FOUND**

SEE ALSO **taskInfo**

2

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.

ERRNO **S_objLib_OBJ_ID_ERROR**

SEE ALSO **taskLib**, *taskPrioritySet*( )

taskPrioritySet( )

NAME *taskPrioritySet*( ) – change the priority of a task

SYNOPSIS

```
STATUS taskPrioritySet
    (
    int tid,        /* task ID */
    int newPriority /* new priority */
    )
```

DESCRIPTION This routine changes a task's priority to a specified priority. Priorities range from 0, the highest priority, to 255, the lowest priority.

RETURNS OK, or ERROR if the task ID is invalid.

ERRNO **S_taskLib_ILLEGAL_PRIORITY**, **S_objLib_OBJ_ID_ERROR**

SEE ALSO **taskLib**, *taskPriorityGet*( )

---

taskRegsGet( )

NAME *taskRegsGet*( ) – get a task's registers from the TCB

SYNOPSIS

```
STATUS taskRegsGet
    (
    int       tid,  /* task ID */
    REG_SET * pRegs /* put register contents here */
    )
```

DESCRIPTION This routine gathers task information kept in the TCB. It copies the contents of the task's registers to the register structure *pRegs*.

NOTE This routine only works well if the task is known to be in a stable, non-executing state. Self-examination, for instance, is not advisable, as results are unpredictable.

RETURNS OK, or ERROR if the task ID is invalid.

SEE ALSO **taskInfo**, *taskSuspend*( ), *taskRegsSet*( )

---

taskRegsSet( )

NAME *taskRegsSet*( ) – set a task's registers

SYNOPSIS

```
STATUS taskRegsSet
    (
    int       tid,  /* task ID */
    REG_SET * pRegs /* get register contents from here */
    )
```

DESCRIPTION This routine loads a specified register set *pRegs* into a specified task's TCB.

NOTE This routine only works well if the task is known not to be in the ready state. Suspending the task before changing the register set is recommended.

RETURNS OK, or ERROR if the task ID is invalid.

SEE ALSO **taskInfo**, ***taskSuspend*( )**, ***taskRegsGet*( )**

taskRegsShow( )

NAME ***taskRegsShow*( )** – display the contents of a task's registers

SYNOPSIS

```
void taskRegsShow
    (
    int tid /* task ID */
    )
```

DESCRIPTION This routine displays the register contents of a specified task on standard output.

EXAMPLE The following example displays the register of the shell task (68000 family):

```
-> taskRegsShow (taskNameToId ("tShell"))
d0     =        0   d1     =        0   d2     =    578fe   d3     =        1
d4     =   3e84e1   d5     =   3e8568   d6     =        0   d7     = ffffffff
a0     =        0   a1     =        0   a2     =    4f06c   a3     =    578d0
a4     =   3fffc4   a5     =        0   fp     =   3e844c   sp     =   3e842c
sr     =     3000   pc     =    4f0f2
```

RETURNS N/A

SEE ALSO **taskShow**

2

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.

ERRNO **S_intLib_NOT_ISR_CALLABLE, S_objLib_OBJ_DELETED, S_objLib_OBJ_UNAVAILABLE, S_objLib_OBJ_ID_ERROR, S_smObjLib_NOT_INITIALIZED, S_memLib_NOT_ENOUGH_MEMORY, S_memLib_BLOCK_ERROR**

SEE ALSO **taskLib**

taskResume( )

NAME *taskResume*( ) – resume a task

SYNOPSIS

```
STATUS taskResume
    (
    int tid /* task ID of task to resume */
    )
```

DESCRIPTION This routine resumes a specified task. Suspension is cleared, and the task operates in the remaining state.

RETURNS OK, or ERROR if the task cannot be resumed.

ERRNO **S_objLib_OBJ_ID_ERROR**

SEE ALSO **taskLib**

taskSafe( )

NAME *taskSafe*( ) – make the calling task safe from deletion

SYNOPSIS

```
STATUS taskSafe (void)
```

DESCRIPTION This routine protects the calling task from deletion. Tasks that attempt to delete a protected task will block until the task is made unsafe, using ***taskUnsafe*( )**. When a task becomes unsafe, the deleter will be unblocked and allowed to delete the task.

The ***taskSafe*( )** primitive utilizes a count to keep track of nested calls for task protection. When nesting occurs, the task becomes unsafe only after the outermost ***taskUnsafe*( )** is executed.

RETURNS OK.

SEE ALSO **taskLib**, ***taskUnsafe*( )**, *VxWorks Programmer's Guide: Basic OS*

taskShow( )

NAME *taskShow*( ) – display task information from TCBs

SYNOPSIS

```
STATUS taskShow
    (
    int tid,  /* task ID */
    int level /* 0 = summary, 1 = details, 2 = all tasks */
    )
```

DESCRIPTION This routine displays the contents of a task control block (TCB) for a specified task. If *level* is 1, it also displays task options and registers. If *level* is 2, it displays all tasks.

The TCB display contains the following fields:

Field	Meaning
NAME	Task name
ENTRY	Symbol name or address where task began execution
TID	Task ID
PRI	Priority
STATUS	Task status, as formatted by ***taskStatusString*( )**
PC	Program counter

Field	**Meaning**
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. It is called automatically when the task show facility is configured into VxWorks using either of the following methods:

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

– If you use the Tornado project facility, select **INCLUDE_TASK_SHOW**.

RETURNS

N/A

SEE ALSO

taskShow

taskSpawn( )

NAME

taskSpawn( ) – spawn a task

SYNOPSIS

```
int taskSpawn
    (
    char *  name,      /* name of new task (stored at pStackBase) */
    int     priority,  /* priority of new task */
    int     options,   /* task option word */
    int     stackSize, /* size (bytes) of stack needed plus name */
    FUNCPTR entryPt,   /* entry point of new task */
    int     arg1,      /* 1st of 10 req'd task args to pass to func */
    int     arg2,
    int     arg3,
    int     arg4,
    int     arg5,
    int     arg6,
    int     arg7,
    int     arg8,
    int     arg9,
    int     arg10
    )
```

DESCRIPTION

This routine creates and activates a new task with a specified priority and options and returns a system-assigned ID. See ***taskInit*( )** and ***taskActivate*( )** for the building blocks of this routine.

A task may be assigned a name as a debugging aid. This name will appear in displays generated by various system information facilities such as ***i*( )**. The name may be of arbitrary length and content, but the current VxWorks convention is to limit task names to ten characters and prefix them with a "t". If *name* is specified as NULL, an ASCII name will be assigned to the task of the form "t*n*" where *n* is an integer which increments as new tasks are spawned.

The only resource allocated to a spawned task is a stack of a specified size *stackSize*, which is allocated from the system memory partition. Stack size should be an even integer. A task control block (TCB) is carved from the stack, as well as any memory required by the task name. The remaining memory is the task's stack and every byte is filled with the value 0xEE for the ***checkStack*( )** facility. See the manual entry for ***checkStack*( )** for stack-size checking aids.

The entry address *entryPt* is the address of the "main" routine of the task. The routine will be called once the C environment has been set up. The specified routine will be called with the ten given arguments. Should the specified main routine return, a call to ***exit*( )** will automatically be made.

Note that ten (and only ten) arguments must be passed for the spawned function.

Bits in the options argument may be set to run with the following modes:

VX_FP_TASK (0x0008)
execute with floating-point coprocessor support.

VX_PRIVATE_ENV (0x0080)
include private environment support (see **envLib**).

VX_NO_STACK_FILL (0x0100)
do not fill the stack for use by ***checkStack*( )**.

VX_UNBREAKABLE (0x0002)
do not allow breakpoint debugging.

See the definitions in **taskLib.h**.

RETURNS
The task ID, or ERROR if memory is insufficient or the task cannot be created.

ERRNO
S_intLib_NOT_ISR_CALLABLE, **S_objLib_OBJ_ID_ERROR**, **S_smObjLib_NOT_INITIALIZED**, **S_memLib_NOT_ENOUGH_MEMORY**, **S_memLib_BLOCK_ERROR**

SEE ALSO
taskLib, ***taskInit*( )**, ***taskActivate*( )**, ***sp*( )**, *VxWorks Programmer's Guide: Basic OS*

---

taskSRInit( )

NAME
***taskSRInit*( )** – initialize the default task status register (MIPS)

SYNOPSIS
```
ULONG taskSRInit
    (
    ULONG newSRValue /* new default task status register */
    )
```

DESCRIPTION
This routine sets the default status register for system-wide tasks. All tasks will be spawned with the status register set to this value; thus, it must be called before ***kernelInit*( )**.

RETURNS
The previous value of the default status register.

SEE ALSO
taskArchLib

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 task that is not running (i.e., the TCB must not be that of the calling task). Debugging facilities use this routine to set the trace bit in the status register of a task that is being single-stepped.

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.

String	Meaning
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.

ERRNO **S_objLib_OBJ_ID_ERROR**

SEE ALSO **taskLib**

taskSwitchHookAdd( )

NAME *taskSwitchHookAdd*( ) – add a routine to be called at every task switch

SYNOPSIS

```
STATUS taskSwitchHookAdd
    (
    FUNCPTR switchHook /* routine to be called at every task switch */
    )
```

DESCRIPTION This routine adds a specified routine to a list of routines that will be called at every task switch. The routine should be declared as follows:

```
void switchHook
    (
    WIND_TCB *pOldTcb,    /* pointer to old task's WIND_TCB */
    WIND_TCB *pNewTcb     /* pointer to new task's WIND_TCB */
    )
```

NOTE User-installed switch hooks are called within the kernel context. Therefore, switch hooks do not have access to all VxWorks facilities. The following routines can be called from within a task switch hook:

Library	Routines
bLib	All routines
fppArchLib	*fppSave*( ), *fppRestore*( )
intLib	*intContext*( ), *intCount*( ), *intVecSet*( ), *intVecGet*( )
lstLib	All routines
mathALib	All routines, if *fppSave*( )/*fppRestore*( ) are used
rngLib	All routines except *rngCreate*( )
taskLib	*taskIdVerify*( ), *taskIdDefault*( ), *taskIsReady*( ), *taskIsSuspended*( ), *taskTcb*( )
vxLib	*vxTas*( )

RETURNS OK, or ERROR if the table of task switch routines is full.

SEE ALSO **taskHookLib**, *taskSwitchHookDelete*( )

2

taskSwitchHookDelete( )

NAME *taskSwitchHookDelete*( ) – delete a previously added task switch routine

SYNOPSIS

```
STATUS taskSwitchHookDelete
    (
    FUNCPTR switchHook /* routine to be deleted from list */
    )
```

DESCRIPTION This routine removes the specified routine from the list of routines to be called at each task switch.

RETURNS OK, or ERROR if the routine is not in the table of task switch routines.

SEE ALSO **taskHookLib**, *taskSwitchHookAdd*( )

taskSwitchHookShow( )

NAME *taskSwitchHookShow*( ) – show the list of task switch routines

SYNOPSIS

```
void taskSwitchHookShow (void)
```

DESCRIPTION This routine shows all the switch routines installed in the task switch hook table, in the order in which they were installed.

RETURNS N/A

SEE ALSO **taskHookShow**, *taskSwitchHookAdd*( )

taskTcb( )

NAME *taskTcb*( ) – get the task control block for a task ID

SYNOPSIS

```
WIND_TCB *taskTcb
    (
    int tid /* task ID */
    )
```

DESCRIPTION This routine returns a pointer to the task control block (**WIND_TCB**) for a specified task. Although all task state information is contained in the TCB, users must not modify it directly. To change registers, for instance, use ***taskRegsSet*( )** and ***taskRegsGet*( )**.

RETURNS A pointer to a **WIND_TCB**, or NULL if the task ID is invalid.

ERRNO **S_objLib_OBJ_ID_ERROR**

SEE ALSO **taskLib**

---

taskUnlock( )

NAME ***taskUnlock*( )** – enable task rescheduling

SYNOPSIS `STATUS taskUnlock (void)`

DESCRIPTION This routine decrements the preemption lock count. Typically this call is paired with ***taskLock*( )** and concludes a critical section of code. Preemption will not be unlocked until ***taskUnlock*( )** has been called as many times as ***taskLock*( )**. When the lock count is decremented to zero, any tasks that were eligible to preempt the current task will execute.

The ***taskUnlock*( )** routine is not callable from interrupt service routines.

RETURNS OK or ERROR.

ERRNO **S_intLib_NOT_ISR_CALLABLE**

SEE ALSO **taskLib**, ***taskLock*( )**

---

taskUnsafe( )

NAME ***taskUnsafe*( )** – make the calling task unsafe from deletion

SYNOPSIS `STATUS taskUnsafe (void)`

DESCRIPTION This routine removes the calling task's protection from deletion. Tasks that attempt to delete a protected task will block until the task is unsafe. When a task becomes unsafe, the deleter will be unblocked and allowed to delete the task.

2

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

tcpShow

tcpShowInit( )

NAME *tcpShowInit*( ) – initialize TCP show routines

SYNOPSIS
```
void tcpShowInit (void)
```

DESCRIPTION This routine links the TCP show facility into the VxWorks system. These routines are included automatically if **INCLUDE_NET_SHOW** and **INCLUDE_TCP** are defined in **configAll.h**.

RETURNS N/A

SEE ALSO **tcpShow**

tcpstatShow( )

NAME *tcpstatShow*( ) – display all statistics for the TCP protocol

SYNOPSIS
```
void tcpstatShow (void)
```

DESCRIPTION This routine displays detailed statistics for the TCP protocol.

RETURNS N/A

SEE ALSO **tcpShow**

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 the configuration macro **INCLUDE_TELNET** is defined.

RETURNS N/A

SEE ALSO **telnetLib**

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**

2

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
value = 0 = 0x0
->
```

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:

```
#include "tftpLib.h"
#define BUFFERSIZE       512
int  dataFd;
int  errorFd;
int  num;
char buf [BUFFERSIZE + 1];
if (tftpXfer ("congo", 0, "/usr/fred", "get", "ascii", &dataFd,
              &errorFd) == ERROR)
    return (ERROR);
while ((num = read (dataFd, buf, sizeof (buf))) > 0)
```

```
        {
        ....
        }
    close (dataFd);
    num = read (errorFd, buf, BUFFERSIZE);
    if (num > 0)
        {
        buf [num] = '\0';
        printf ("YIKES! An error occurred!:%s\n", buf);
        .....
        }
    close (errorFd);
```

RETURNS OK, or ERROR if unsuccessful.

ERRNO **S_tftpLib_INVALID_ARGUMENT**

SEE ALSO **tftpLib**, **ftpLib**

ti( )

NAME ***ti*( )** – print complete information from a task's TCB

SYNOPSIS

```
void ti
    (
    int taskNameOrId /* task name or task ID; 0 = use default */
    )
```

DESCRIPTION This command prints the task control block (TCB) contents, including registers, for a specified task. If *taskNameOrId* is omitted or zero, the last task referenced is assumed.

The ***ti*( )** routine uses ***taskShow*( )**; see the documentation for ***taskShow*( )** for a description of the output format.

EXAMPLE The following shows the TCB contents for the shell task:

```
-> ti
  NAME      ENTRY      TID    PRI  STATUS      PC       SP     ERRNO  DELAY
---------- --------- -------- --- --------- -------- -------- ------ -----
tShell     _shell     20efcac   1 READY      201dc90  20ef980      0     0
stack: base 0x20efcac  end 0x20ed59c  size 9532   high 1452   margin 8080
options: 0x1e
VX_UNBREAKABLE      VX_DEALLOC_STACK    VX_FP_TASK         VX_STDIO
```

```
D0 =        0   D4 =        0   A0 =        0   A4 =        0
D1 =        0   D5 =        0   A1 =        0   A5 =  203a084   SR =     3000
D2 =        0   D6 =        0   A2 =        0   A6 =  20ef9a0   PC =  2038614
D3 =        0   D7 =        0   A3 =        0   A7 =  20ef980
value = 34536868 = 0x20efda4
```

RETURNS N/A

SEE ALSO **usrLib**, ***taskShow*( )**, *VxWorks Programmer's Guide: Target Shell*, **windsh**, *Tornado User's Guide: Shell*

---

tickAnnounce( )

NAME ***tickAnnounce*( )** – announce a clock tick to the kernel

SYNOPSIS **`void tickAnnounce (void)`**

DESCRIPTION This routine informs the kernel of the passing of time. It should be called from an interrupt service routine that is connected to the system clock. The most common frequencies are 60Hz or 100Hz. Frequencies in excess of 600Hz are an inefficient use of processor power because the system will spend most of its time advancing the clock. By default, this routine is called by ***usrClock*( )** in **usrConfig.c**.

RETURNS N/A

SEE ALSO **tickLib**, **kernelLib**, **taskLib**, **semLib**, **wdLib**, *VxWorks Programmer's Guide: Basic OS*

---

tickGet( )

NAME ***tickGet*( )** – get the value of the kernel's tick counter

SYNOPSIS **`ULONG tickGet (void)`**

DESCRIPTION This routine returns the current value of the tick counter. This value is set to zero at startup, incremented by ***tickAnnounce*( )**, and can be changed using ***tickSet*( )**.

RETURNS The most recent ***tickSet*( )** value, plus all ***tickAnnounce*( )** calls since.

SEE ALSO **tickLib**, ***tickSet*( )**, ***tickAnnounce*( )**

tickSet( )

NAME

tickSet( ) – set the value of the kernel's tick counter

SYNOPSIS

```
void tickSet
    (
    ULONG ticks /* new time in ticks */
    )
```

DESCRIPTION

This routine sets the internal tick counter to a specified value in ticks. The new count will be reflected by ***tickGet*( )**, but will not change any delay fields or timeouts selected for any tasks. For example, if a task is delayed for ten ticks, and this routine is called to advance time, the delayed task will still be delayed until ten ***tickAnnounce*( )** calls have been made.

RETURNS

N/A

SEE ALSO

tickLib, *tickGet*( ), *tickAnnounce*( )

time( )

NAME

time( ) – determine the current calendar time (ANSI)

SYNOPSIS

```
time_t time
    (
    time_t * timer /* calendar time in seconds */
    )
```

DESCRIPTION

This routine returns the implementation's best approximation of current calendar time in seconds. If *timer* is non-NULL, the return value is also copied to the location to which *timer* points.

INCLUDE FILES

time.h

RETURNS

The current calendar time in seconds, or ERROR (-1) if the calendar time is not available.

SEE ALSO

ansiTime, *clock_gettime*( )

timer_cancel( )

NAME

timer_cancel( ) – cancel a timer

SYNOPSIS

```
int timer_cancel
    (
    timer_t timerid /* timer ID */
    )
```

DESCRIPTION

This routine is a shorthand method of invoking ***timer_settime*( )**, which stops a timer.

NOTE

Non-POSIX.

RETURNS

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

ERRNO

EINVAL

SEE ALSO

timerLib

timer_connect( )

NAME

timer_connect( ) – connect a user routine to the timer signal

SYNOPSIS

```
int timer_connect
    (
    timer_t     timerid, /* timer ID */
    VOIDFUNCPTR routine, /* user routine */
    int         arg      /* user argument */
    )
```

DESCRIPTION

This routine sets the specified *routine* to be invoked with *arg* when fielding a signal indicated by the timer's *evp* signal number, or if *evp* is NULL, when fielding the default signal (**SIGALRM**).

The signal handling routine should be declared as:

```
    void my_handler
        (
        timer_t timerid,      /* expired timer ID */
        int arg               /* user argument    */
        )
```

NOTE Non-POSIX.

RETURNS 0 (OK), or -1 (ERROR) if the timer is invalid or cannot bind the signal handler.

ERRNO **EINVAL**

SEE ALSO **timerLib**

---

timer_create( )

NAME *timer_create*( ) – allocate a timer using the specified clock for a timing base (POSIX)

SYNOPSIS

```
int timer_create
    (
    clockid_t         clock_id, /* clock ID (always CLOCK_REALTIME) */
    struct sigevent * evp,      /* user event handler */
    timer_t *         pTimer    /* ptr to return value */
    )
```

DESCRIPTION This routine returns a value in *pTimer* that identifies the timer in subsequent timer requests. The *evp* argument, if non-NULL, points to a **sigevent** structure, which is allocated by the application and defines the signal number and application-specific data to be sent to the task when the timer expires. If *evp* is NULL, a default signal (**SIGALRM**) is queued to the task, and the signal data is set to the timer ID. Initially, the timer is disarmed.

RETURNS 0 (OK), or -1 (ERROR) if too many timers already are allocated or the signal number is invalid.

ERRNO **EMTIMERS**, **EINVAL**, **ENOSYS**, **EAGAIN**, **S_memLib_NOT_ENOUGH_MEMORY**

SEE ALSO **timerLib**, *timer_delete*( )

timer_delete( )

NAME *timer_delete*( ) – remove a previously created timer (POSIX)

SYNOPSIS

```
int timer_delete
    (
    timer_t timerid /* timer ID */
    )
```

DESCRIPTION This routine removes a timer.

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

ERRNO EINVAL

SEE ALSO **timerLib**, *timer_create*( )

timer_getoverrun( )

NAME *timer_getoverrun*( ) – return the timer expiration overrun (POSIX)

SYNOPSIS

```
int timer_getoverrun
    (
    timer_t timerid /* timer ID */
    )
```

DESCRIPTION This routine returns the timer expiration overrun count for *timerid*, when called from a timer expiration signal catcher. The overrun count is the number of extra timer expirations that have occurred, up to the implementation-defined maximum **_POSIX_DELAYTIMER_MAX**. If the count is greater than the maximum, it returns the maximum.

RETURNS The number of overruns, or **_POSIX_DELAYTIMER_MAX** if the count equals or is greater than **_POSIX_DELAYTIMER_MAX**, or -1 (ERROR) if *timerid* is invalid.

ERRNO EINVAL, ENOSYS

SEE ALSO **timerLib**

timer_gettime( )

NAME

timer_gettime( ) – get the remaining time before expiration and the reload value (POSIX)

SYNOPSIS

```
int timer_gettime
    (
    timer_t               timerid, /* timer ID */
    struct itimerspec * value    /* where to return remaining time */
    )
```

DESCRIPTION

This routine gets the remaining time and reload value of a specified timer. Both values are copied to the *value* structure.

RETURNS

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

ERRNO

EINVAL

SEE ALSO

timerLib

timer_settime( )

NAME

timer_settime( ) – set the time until the next expiration and arm timer (POSIX)

SYNOPSIS

```
int timer_settime
    (
    timer_t                     timerid, /* timer ID */
    int                         flags,   /* absolute or relative */
    const struct itimerspec * value,   /* time to be set */
    struct itimerspec *         ovalue /* previous time set (NULL=no result) */
    )
```

DESCRIPTION

This routine sets the next expiration of the timer, using the **.it_value**of *value*, thus arming the timer. If the timer is already armed, this call resets the time until the next expiration. If **.it_value** is zero, the timer is disarmed.

If *flags* is not equal to **TIMER_ABSTIME**, the interval is relative to the current time, the interval being the **.it_value** of the *value* parameter. If *flags* is equal to **TIMER_ABSTIME**, the expiration is set to the difference between the absolute time of **.it_value** and the current value of the clock associated with *timerid*. If the time has already passed, then the timer expiration notification is made immediately. The task that sets the timer receives the signal; in other words, the taskId is noted. If a timer is set by an ISR, the signal is delivered to the task that created the timer.

The reload value of the timer is set to the value specified by the **.it_interval** field of *value*. When a timer is armed with a nonzero **.it_interval** a periodic timer is set up.

Time values that are between two consecutive non-negative integer multiples of the resolution of the specified timer are rounded up to the larger multiple of the resolution.

If *ovalue* is non-NULL, the routine stores a value representing the previous amount of time before the timer would have expired. Or if the timer is disarmed, the routine stores zero, together with the previous timer reload value. The *ovalue* parameter is the same value as that returned by ***timer_gettime*( )** and is subject to the timer resolution.

WARNING

If ***clock_settime*( )** is called to reset the absolute clock time after a timer has been set with ***timer_settime*( )**, and if *flags* is equal to **TIMER_ABSTIME**, then the timer will behave unpredictably. If you must reset the absolute clock time after setting a timer, do not use *flags* equal to **TIMER_ABSTIME**.

RETURNS

0 (OK), or -1 (ERROR) if *timerid* is invalid, the number of nanoseconds specified by *value* is less than 0 or greater than or equal to 1,000,000,000, or the time specified by *value* exceeds the maximum allowed by the timer.

ERRNO

EINVAL

SEE ALSO

timerLib

timex( )

NAME

***timex*( )** – time a single execution of a function or functions

SYNOPSIS

```
void timex
    (
    FUNCPTR func, /* function to time (optional) */
    int     arg1, /* first of up to 8 args to call 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 the configuration macro **INCLUDE_TIMEX** is defined, it is called by the root task, ***usrRoot*( )**, in **usrConfig.c**.

RETURNS N/A

SEE ALSO **timexLib**

timexN( )

NAME *timexN*( ) – time repeated executions of a function or group of functions

SYNOPSIS

```
void timexN
    (
    FUNCPTR func, /* function to time (optional) */
    int     arg1, /* first of up to 8 args to call function with */
    int     arg2,
    int     arg3,
    int     arg4,
    int     arg5,
    int     arg6,
    int     arg7,
    int     arg8
    )
```

DESCRIPTION This routine times the execution of the current list of functions to be timed in the same manner as ***timex*( )**; however, the list of functions is called a variable number of times until sufficient resolution is achieved to establish the time with an error less than 2%. (Since each iteration of the list may be measured to a resolution of +/- 1 clock tick, repetitive timings decrease this error to 1/N ticks, where N is the number of repetitions.)

RETURNS N/A

SEE ALSO **timexLib**, *timexFunc*( ), *timex*( )

2

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**

2

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( ) – display a stack trace of a task

SYNOPSIS

```
STATUS tt
    (
    int taskNameOrId /* task name or task ID */
    )
```

DESCRIPTION

This routine displays a list of the nested routine calls that the specified task is in. Each routine call and its parameters are shown.

If *taskNameOrId* is not specified or zero, the last task referenced is assumed. The ***tt( )*** routine can only trace the stack of a task other than itself. For instance, when ***tt( )*** is called from the shell, it cannot trace the shell's stack.

EXAMPLE

```
-> tt "logTask"
 3ab92 _vxTaskEntry   +10 : _logTask (0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
  ee6e _logTask       +12 : _read (5, 3f8a10, 20)
  d460 _read          +10 : _iosRead (5, 3f8a10, 20)
  e234 _iosRead       +9c : _pipeRead (3fce1c, 3f8a10, 20)
 23978 _pipeRead      +24 : _semTake (3f8b78)
value = 0 = 0x0
```

This indicates that ***logTask( )*** is currently in ***semTake( )*** (with one parameter) and was called by ***pipeRead( )*** (with three parameters), which was called by ***iosRead( )*** (with three parameters), and so on.

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 line-delete character is CTRL+U.

Typing the delete character to any device operating in line protocol mode (**OPT_LINE** set) will cause all characters in the current line to be deleted. Note that the character set by this routine applies to all devices whose handlers use the standard tty package **tyLib**.

RETURNS

N/A

SEE ALSO

tyLib

tyDevInit( )

NAME

tyDevInit( ) – initialize the tty device descriptor

SYNOPSIS

```
STATUS tyDevInit
    (
    TY_DEV_ID pTyDev,     /* ptr to tty dev descriptor to init */
    int       rdBufSize,  /* size of read buffer in bytes */
    int       wrtBufSize, /* size of write buffer in bytes */
    FUNCPTR   txStartup   /* device transmit start-up routine */
    )
```

DESCRIPTION

This routine initializes a tty device descriptor according to the specified parameters. The initialization includes allocating read and write buffers of the specified sizes from the memory pool, and initializing their respective buffer descriptors. The semaphores are initialized and the write semaphore is given to enable writers. Also, the transmitter start-up routine pointer is set to the specified routine. All other fields in the descriptor are zeroed.

This routine should be called only by serial drivers.

RETURNS

OK, or ERROR if there is not enough memory to allocate data structures.

SEE ALSO

tyLib

tyEOFSet( )

NAME

tyEOFSet( ) – change the end-of-file character

SYNOPSIS

```
void tyEOFSet
    (
    char ch /* char to be EOF */
    )
```

DESCRIPTION

This routine sets the EOF character to *ch*. The default EOF character is CTRL-D.

Typing the EOF character to any device operating in line protocol mode (**OPT_LINE** set) will cause no character to be entered in the current line, but will cause the current line to be terminated (thus without a newline character). The line is made available to reading tasks. Thus, if the EOF character is the first character input on a line, a line length of zero characters is returned to the reader. This is the standard end-of-file indication on a read call. Note that the EOF character set by this routine will apply to all devices whose handlers use the standard tty package **tyLib**.

RETURNS N/A

SEE ALSO **tyLib**

---

tyIoctl( )

NAME *tyIoctl*( ) – handle device control requests

SYNOPSIS

```
STATUS tyIoctl
    (
    TY_DEV_ID pTyDev,  /* ptr to device to control */
    int       request, /* request code */
    int       arg      /* some argument */
    )
```

DESCRIPTION This routine handles *ioctl*( ) requests for tty devices. The I/O control functions for tty devices are described in the manual entry for **tyLib**.

BUGS In line protocol mode (**OPT_LINE** option set), the **FIONREAD** function actually returns the number of characters available plus the number of lines in the buffer. Thus, if five lines consisting of just NEWLINEs were in the input buffer, the **FIONREAD** function would return the value ten (five characters + five lines).

RETURNS OK or ERROR.

SEE ALSO **tyLib**

---

tyIRd( )

NAME *tyIRd*( ) – interrupt-level input

SYNOPSIS

```
STATUS tyIRd
    (
    TY_DEV_ID pTyDev, /* ptr to tty device descriptor */
    char      inchar  /* character read */
    )
```

DESCRIPTION This routine handles interrupt-level character input for tty devices. A device driver calls this routine when it has received a character. This routine adds the character to the ring buffer for the specified device, and gives a semaphore if a task is waiting for it.

This routine also handles all the special characters, as specified in the option word for the device, such as X-on, X-off, NEWLINE, or backspace.

RETURNS OK, or ERROR if the ring buffer is full.

SEE ALSO **tyLib**

---

tyITx( )

NAME *tyITx*( ) – interrupt-level output

SYNOPSIS

```
STATUS tyITx
    (
    TY_DEV_ID pTyDev, /* pointer to tty device descriptor */
    char *    pChar   /* where to put character to be output */
    )
```

DESCRIPTION This routine gets a single character to be output to a device. It looks at the ring buffer for *pTyDev* and gives the caller the next available character, if there is one. The character to be output is copied to *pChar*.

RETURNS OK if there are more characters to send, or ERROR if there are no more characters.

SEE ALSO **tyLib**

---

tyMonitorTrapSet( )

NAME *tyMonitorTrapSet*( ) – change the trap-to-monitor character

SYNOPSIS

```
void tyMonitorTrapSet
    (
    char ch /* char to be monitor trap */
    )
```

DESCRIPTION This routine sets the trap-to-monitor character to *ch*. The default trap-to-monitor character is CTRL+X.

Typing the trap-to-monitor character to any device whose **OPT_MON_TRAP** option is set will cause the resident ROM monitor to be entered, if one is present. Once the ROM monitor is entered, the normal multitasking system is halted.

Note that the trap-to-monitor character set by this routine will apply to all devices whose handlers use the standard tty package **tyLib**. Also note that not all systems have a monitor trap available.

RETURNS N/A

SEE ALSO **tyLib**

tyRead( )

NAME *tyRead*( ) – do a task-level read for a tty device

SYNOPSIS

```
int tyRead
    (
    TY_DEV_ID pTyDev,  /* device to read */
    char *    buffer,  /* buffer to read into */
    int       maxbytes /* maximum length of read */
    )
```

DESCRIPTION This routine handles the task-level portion of the tty handler's read function. It reads into the buffer up to *maxbytes* available bytes.

This routine should only be called from serial device drivers.

RETURNS The number of bytes actually read into the buffer.

SEE ALSO **tyLib**

tyWrite( )

NAME *tyWrite*( ) – do a task-level write for a tty device

SYNOPSIS

```
int tyWrite
    (
    TY_DEV_ID pTyDev, /* ptr to device structure */
    char *    buffer, /* buffer of data to write */
    int       nbytes  /* number of bytes in buffer */
    )
```

DESCRIPTION This routine handles the task-level portion of the tty handler's write function.

RETURNS The number of bytes actually written to the device.

SEE ALSO **tyLib**

udpShowInit( )

NAME *udpShowInit*( ) – initialize UDP show routines

SYNOPSIS

```
void udpShowInit (void)
```

DESCRIPTION This routine links the UDP show facility into the VxWorks system. These routines are included automatically if **INCLUDE_NET_SHOW** and **INCLUDE_UDP** are defined in **configAll.h**.

RETURNS N/A

SEE ALSO **udpShow**

udpstatShow( )

NAME *udpstatShow*( ) – display statistics for the UDP protocol

SYNOPSIS

```
void udpstatShow (void)
```

DESCRIPTION This routine displays statistics for the UDP protocol.

RETURNS N/A

SEE ALSO **udpShow**

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.

ERRNO **S_if_ul_UNIT_ALREADY_INITIALIZED**

SEE ALSO **if_ulip**, *VxSim User's Guide*

ulipDebugSet( )

NAME *ulipDebugSet*( ) – Set debug flag in UNIX's ULIP driver

SYNOPSIS

```
STATUS ulipDebugSet
    (
    int debugFlag
    )
```

DESCRIPTION This function uses an ioctl call to UNIX's (Solaris's) ULIP driver to set that driver's debugging flag to the value in debugFlag. Because there is no simple way for the caller to assertain the unit number of the interface in use, all unit numbers are looped over and each receives the ioctl. Possible values for the debug flag are discussed above in this file, although all the levels have not been implemented.

This is not the right place to put this function (user callable routines would be more appropriately placed in **simLib.h**). Because of the requirement to use both Sun structures (to bundle ioctl data) and VxWorks structures (**ul_softc**), and given the same requirements when calling the **FIOSETUSED** ioctl, this seems the best place to put it.

RETURNS OK or ERROR if the ioctl fails

SEE ALSO **if_ulip**

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 system resources taken up by this ULIP interface.

RETURNS OK, or ERROR if the unit number is invalid or the interface is uninitialized.

ERRNO **S_if_ul_INVALID_UNIT_NUMBER**, **S_if_ul_UNIT_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.

ERRNO **S_if_ul_INVALID_UNIT_NUMBER**

SEE ALSO **if_ulip**, *VxSim User's Guide*

---

ulStartOutput( )

NAME *ulStartOutput*( ) – push packets onto "interface"

SYNOPSIS

```
#ifdef BSD43_DRIVER LOCAL STATUS ulStartOutput
    (
    int unit
    )
```

SEE ALSO **if_ulip**

---

ultraAddrFilterSet( )

NAME *ultraAddrFilterSet*( ) – set the address filter for multicast addresses

SYNOPSIS

```
void ultraAddrFilterSet
    (
    ULTRA_DEVICE * pDrvCtrl /* device pointer */
    )
```

DESCRIPTION This routine goes through all of the multicast addresses on the list of addresses (added with the ***ultraMCastAdd*( )** routine) and sets the device's filter correctly.

RETURNS N/A.

SEE ALSO **ultraEnd**

2

ultraattach( )

NAME

ultraattach( ) – publish **ultra** interface and initialize 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**

ultraLoad( )

NAME

ultraLoad( ) – initialize the driver and device

SYNOPSIS

```
END_OBJ* ultraLoad
    (
    char * initString /* String to be parsed by the driver. */
    )
```

DESCRIPTION

This routine initializes the driver and the device to the operational state. All of the device-specific parameters are passed in *initString*, which expects a string of the following format:

unit:*ioAddr*:*memAddr*:*vecNum*:*intLvl*:*config*:*offset*"

This routine can be called in two modes. If it is called with an empty but allocated string, it places the name of this device (that is, "ultra") into the *initString* and returns 0.

If the string is allocated and not empty, the routine attempts to load the driver using the values specified in the string.

RETURNS
An END object pointer, or NULL on error, or 0 and the name of the device if the *initString* was NULL.

SEE ALSO
ultraEnd

ultraMemInit( )

NAME
ultraMemInit( ) – initialize memory for the chip

SYNOPSIS
```
STATUS ultraMemInit
    (
    ULTRA_DEVICE * pDrvCtrl, /* device to be initialized */
    int            clNum     /* number of clusters to allocate */
    )
```

DESCRIPTION
Using data in the control structure, setup and initialize the memory areas needed. If the memory address is not already specified, then allocate cache safe memory.

RETURNS
OK or ERROR.

SEE ALSO
ultraEnd

ultraParse( )

NAME
ultraParse( ) – parse the init string

SYNOPSIS
```
STATUS ultraParse
    (
    ULTRA_DEVICE * pDrvCtrl,  /* device pointer */
    char *         initString /* information string */
    )
```

DESCRIPTION
Parse the input string. Fill in values in the driver control structure. The initialization string format is: *unit*:*ioAddr*:*memAddr*:*vecNum*:*intLvl*:*config*:*offset*"

unit
Device unit number, a small integer.

ioAddr
I/O address

memAddr
Memory address, assumed to be 16k bytes in length.

vecNum
Interrupt vector number (used with ***sysIntConnect*( )**).

intLvl
Interrupt level.

config
Ultra config (0: RJ45 + AUI(Thick) 1: RJ45 + BNC(Thin)).

offset
Memory offset for alignment.

RETURNS OK, or ERROR if any arguments are invalid.

SEE ALSO **ultraEnd**

---

ultraPut( )

NAME ***ultraPut*( )** – copy a packet to the interface.

SYNOPSIS

```
#ifdef BSD43_DRIVER LOCAL void ultraPut
    (
    int unit /* device unit number */
    )
```

DESCRIPTION Copy from mbuf chain to transmitter buffer in shared memory.

RETURNS N/A

SEE ALSO **if_ultra**

*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. 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_ultra

*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***( )**

---

unixDevInit( )

NAME *unixDevInit***( )** – initialize a **UNIX_DUSART**

SYNOPSIS

```
void unixDevInit
    (
    UNIX_CHAN * pChan
    )
```

DESCRIPTION This routine initializes the driver function pointers and then resets to a quiescent state. The BSP must have already opened all the file descriptors in the structure before passing it to this routine.

RETURNS N/A

SEE ALSO **unixSio**

unixDevInit2( )

NAME *unixDevInit2*( ) – enable interrupts

SYNOPSIS

```
void unixDevInit2
    (
    UNIX_CHAN * pChan
    )
```

RETURNS N/A

SEE ALSO **unixSio**

unixDiskDevCreate( )

NAME *unixDiskDevCreate*( ) – create a UNIX disk device

SYNOPSIS

```
BLK_DEV *unixDiskDevCreate
    (
    char * unixFile,     /* name of the UNIX file */
    int    bytesPerBlk,  /* number of bytes per block */
    int    blksPerTrack, /* number of blocks per track */
    int    nBlocks       /* number of blocks on this device */
    )
```

DESCRIPTION This routine creates a UNIX disk device.

The *unixFile* parameter specifies the name of the UNIX file to use for the disk device.

The *bytesPerBlk* parameter specifies the size of each logical block on the disk. If *bytesPerBlk* is zero, 512 is the default.

The *blksPerTrack* parameter specifies the number of blocks on each logical track of the disk. If *blksPerTrack* is zero, the count of blocks per track is set to *nBlocks* (i.e., the disk is defined as having only one track).

The *nBlocks* parameter specifies the size of the disk, in blocks. If *nBlocks* is zero, a default size is used. The default is calculated as the size of the UNIX disk divided by the number of bytes per block.

This routine is only applicable to VxSim for Solaris and VxSim for HP.

RETURNS A pointer to block device (**BLK_DEV**) structure, or NULL, if unable to open the UNIX disk.

SEE ALSO **unixDrv**

---

unixDiskInit( )

NAME *unixDiskInit*( ) – initialize a dosFs disk on top of UNIX

SYNOPSIS

```
void unixDiskInit
    (
    char * unixFile, /* UNIX file name */
    char * volName,  /* dosFs name */
    int    diskSize  /* number of bytes */
    )
```

DESCRIPTION This routine provides some convenience for a user wanting to create a UNIX disk-based dosFs file system under VxWorks. The user only specifes the UNIX file to use, the dosFs volume name, and the size of the volume in bytes, if the UNIX file needs to be created.

This routine is only applicable to VxSim for Solaris and VxSim for HP.

RETURNS N/A

SEE ALSO **unixDrv**

---

unixDrv( )

NAME *unixDrv*( ) – install UNIX disk driver

SYNOPSIS

```
STATUS unixDrv (void)
```

DESCRIPTION Used in **usrConfig.c** to cause the UNIX disk driver to be linked in when building VxWorks. Otherwise, it is not necessary to call this routine before using the UNIX disk driver.

This routine is only applicable to VxSim for Solaris and VxSim for HP.

RETURNS OK (always).

SEE ALSO **unixDrv**

unixIntRcv( )

NAME *unixIntRcv*( ) – handle a channel's receive-character interrupt.

SYNOPSIS

```
void unixIntRcv
    (
    UNIX_CHAN * pChan /* channel generating the interrupt */
    )
```

RETURNS N/A

SEE ALSO **unixSio**

unld( )

NAME *unld*( ) – unload an object module by specifying a file name or module ID

SYNOPSIS

```
STATUS unld
    (
    void * nameOrId, /* name or ID of the object module file */
    int    options
    )
```

DESCRIPTION This routine unloads the specified object module from the system. The module can be specified by name or by module ID. For a.out and ECOFF format modules, unloading does the following:

(1) It frees the space allocated for text, data, and BSS segments, unless *loadModuleAt*( ) was called with specific addresses, in which case the user is responsible for freeing the space.

(2) It removes all symbols associated with the object module from the system symbol table.

(3) It removes the module descriptor from the module list.

For other modules of other formats, unloading has similar effects.

Before any modules are unloaded, all breakpoints in the system are deleted. If you need to keep breakpoints, set the options parameter to **UNLD_KEEP_BREAKPOINTS**. No breakpoints can be set in code that is unloaded.

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. The value of offset is passed to the macro **ATA_SWAP** for endian adjustment. For the *drive* parameter, 0 is **C:** and 1 is **D:**.

RETURNS The offset to the partition

SEE ALSO **usrAta**

usrClock( )

NAME *usrClock*( ) – user-defined system clock interrupt routine

SYNOPSIS

```
void usrClock ()
```

DESCRIPTION This routine is called at interrupt level on each clock interrupt. It is installed by *usrRoot*( ) with a *sysClkConnect*( ) call. It calls all the other packages that need to know about clock ticks, including the kernel itself.

If the application needs anything to happen at the system clock interrupt level, it can be added to this routine.

RETURNS N/A

SEE ALSO **usrConfig**

usrFdConfig( )

NAME *usrFdConfig*( ) – mount a DOS file system from a floppy disk

SYNOPSIS

```
STATUS usrFdConfig
    (
    int    drive,   /* drive number of floppy disk (0 - 3) */
    int    type,    /* type of floppy disk */
    char * fileName /* mount point */
    )
```

DESCRIPTION This routine mounts a DOS file system from a floppy disk device.

The *drive* parameter is the drive number of the floppy disk; valid values are 0 to 3.

The *type* parameter specifies the type of diskette, which is described in the structure table **fdTypes[]** in **sysLib.c**. *type* is an index to the table. Currently the table contains two diskette types:

– A *type* of 0 indicates the first entry in the table (3.5" 2HD, 1.44MB);

– A *type* of 1 indicates the second entry in the table (5.25" 2HD, 1.2MB).

The *fileName* parameter is the mount point, e.g., **/fd0/**.

NOTE
Do not attempt to unmount a volume that was mounted with ***usrFdConfig*( )** using ***dosFsVolUnmount*( )**. ***usrFdConfig*( )** does not return the **DOS_VOL_CONFIG** structure required by ***dosFsVolUnmount*( )**. Instead use ***ioctl*( )** with **FIOUNMOUNT** which accesses the volume information via the file descriptor.

RETURNS
OK or ERROR.

SEE ALSO
usrFd, *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
usrIde, *VxWorks Programmer's Guide: I/O System, Local File Systems, Intel i386/i486 Appendix*

usrInit( )

NAME

usrInit( ) – user-defined system initialization routine

SYNOPSIS

```
void usrInit
    (
    int startType
    )
```

DESCRIPTION

This is the first C code executed after the system boots. This routine is called by the assembly language start-up routine ***sysInit*( )** which is in the **sysALib** module of the target-specific directory. It is called with interrupts locked out. The kernel is not multitasking at this point.

This routine starts by clearing BSS; thus all variables are initialized to 0, as per the C specification. It then initializes the hardware by calling ***sysHwInit*( )**, sets up the interrupt/exception vectors, and starts kernel multitasking with ***usrRoot*( )** as the root task.

RETURNS

N/A

SEE ALSO

usrConfig, **kernelLib**

usrRoot( )

NAME

usrRoot( ) – the root task

SYNOPSIS

```
void usrRoot
    (
    char *   pMemPoolStart, /* start of system memory partition */
    unsigned memPoolSize    /* initial size of mem pool */
    )
```

DESCRIPTION

This is the first task to run under the multitasking kernel. It performs all final initialization and then starts other tasks.

It initializes the I/O system, installs drivers, creates devices, and sets up the network, etc., as necessary for a particular configuration. It may also create and load the system symbol table, if one is to be included. It may then load and spawn additional tasks as needed. In the default configuration, it simply initializes the VxWorks shell.

RETURNS N/A

SEE ALSO **usrConfig**

---

usrScsiConfig( )

NAME *usrScsiConfig*( ) – configure SCSI peripherals

SYNOPSIS

```
STATUS usrScsiConfig (void)
```

DESCRIPTION This code configures the SCSI disks and other peripherals on a SCSI controller chain.

The macro **SCSI_AUTO_CONFIG** will include code to scan all possible device/lun id's and to configure a scsiPhysDev structure for each device found. Of course this doesn't include final configuration for disk partitions, floppy configuration parameters, or tape system setup. All of these actions must be performed by user code, either through ***sysScsiConfig*( )**, the startup script, or by the application program.

The user may customize this code on a per BSP basis using the **SYS_SCSI_CONFIG** macro. If defined, then this routine will call the routine ***sysScsiConfig*( )**. That routine is to be provided by the BSP, either in **sysLib.c** or **sysScsi.c**. If **SYS_SCSI_CONFIG** is not defined, then ***sysScsiConfig*( )** will not be called as part of this routine.

An example ***sysScsiConfig*( )** routine can be found in **target/src/config/usrScsi.c**. The example code contains sample configurations for a hard disk, a floppy disk and a tape unit.

RETURNS OK or ERROR.

SEE ALSO **usrScsi**, *VxWorks Programmer's Guide: I/O System, Local File Systems*

---

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.

NOTE

The shared memory objects library requires information from fields in the VxWorks boot line. The functions are contained in the **usrNetwork.c** file. If no network services are included, **usrNetwork.c** is not included and the shared memory initialization fails. To avoid this problem, either add **INCLUDE_NETWORK** to **configAll.h** or extract the bootline cracking routines from **usrNetwork.c** and include them elsewhere.

RETURNS

OK, or ERROR if unsuccessful.

SEE ALSO

usrSmObj

***uswab*( )**

NAME

***uswab*( )** – swap bytes with buffers that are not necessarily aligned

SYNOPSIS

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

DESCRIPTION

This routine gets the specified number of bytes from *source*, exchanges the adjacent even and odd bytes, and puts them in *destination*.

NOTE

Due to speed considerations, this routine should only be used when absolutely necessary. Use ***swab*( )** for aligned swaps.

It is an error for *nbytes* to be odd.

RETURNS

N/A

SEE ALSO

bLib, ***swab*( )**

2

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*( )

va_arg( )

NAME *va_arg*( ) – expand to an expression having the type and value of the call's next argument

SYNOPSIS

```
void va_arg
    (
    )
```

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
    (
    )
```

DESCRIPTION

This macro facilitates a normal return from the function whose variable argument list was referred to by the expansion of *va_start*( ) that initialized the **va_list** object.

va_end( ) may modify the **va_list** object so that it is no longer usable (without an intervening invocation of *va_start*( )). If there is no corresponding invocation of the *va_start*( ) macro, or if the *va_end*( ) macro is not invoked before the return, the behavior is undefined.

RETURNS

N/A

SEE ALSO

ansiStdarg

va_start( )

NAME

va_start( ) – initialize a **va_list** object for use by *va_arg*( ) and *va_end*( )

SYNOPSIS

```
void va_start
    (
    )
```

DESCRIPTION

This macro initializes an object of type **va_list** (*ap*) for subsequent use by *va_arg*( ) and *va_end*( ). The parameter *parmN* is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the , ...). If *parmN* is declared with the register storage class with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions, the behavior is undefined.

RETURNS

N/A

SEE ALSO

ansiStdarg

valloc( )

NAME *valloc*( ) – allocate memory on a page boundary

SYNOPSIS

```
void * valloc
    (
    unsigned size /* number of bytes to allocate */
    )
```

DESCRIPTION This routine allocates a buffer of *size* bytes from the system memory partition. Additionally, it insures that the allocated buffer begins on a page boundary. Page sizes are architecture-dependent.

RETURNS A pointer to the newly allocated block, or NULL if the buffer could not be allocated or the memory management unit (MMU) support library has not been initialized.

ERRNO **S_memLib_PAGE_SIZE_UNAVAILABLE**

SEE ALSO **memLib**

version( )

NAME *version*( ) – print VxWorks version information

SYNOPSIS

```
void version (void)
```

DESCRIPTION This command prints the VxWorks version number, the date this copy of VxWorks was made, and other pertinent information.

EXAMPLE

```
-> version
VxWorks (for Mizar 7170) version 5.1
Kernel: WIND version 2.1.
Made on Tue Jul 27 20:26:23 CDT 1997.
Boot line:
enp(0,0)host:/usr/wpwr/target/config/mz7170/vxWorks e=90.0.0.50 h=90.0.0.4
u=target
```

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***( )**

2

vmBaseGlobalMapInit( )

NAME

vmBaseGlobalMapInit( ) – initialize global mapping

SYNOPSIS

```
VM_CONTEXT_ID vmBaseGlobalMapInit
    (
    PHYS_MEM_DESC * pMemDescArray,        /* pointer to array of mem descs */
    int             numDescArrayElements, /* no. of elements */
                                          /* in pMemDescArray */
    BOOL            enable                /* enable virtual memory */
    )
```

DESCRIPTION

This routine creates and installs a virtual memory context with mappings defined for each contiguous memory segment defined in *pMemDescArray*. In the standard VxWorks configuration, an instance of **PHYS_MEM_DESC** (called **sysPhysMemDesc**) is defined in **sysLib.c**; the variable is passed to ***vmBaseGlobalMapInit*( )** by the system configuration mechanism.

The physical memory descriptor also contains state information used to initialize the state information in the MMU's translation table for that memory segment. The following state bits may be or'ed together:

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

vmBaseLibInit( )

vmBaseLibInit( )

NAME *vmBaseLibInit*( ) – initialize base virtual memory support

SYNOPSIS

```
STATUS vmBaseLibInit
    (
    int pageSize /* size of page */
    )
```

DESCRIPTION This routine initializes the virtual memory context class and module-specific data structures. It is called only once during system initialization, and should be followed with a call to ***vmBaseGlobalMapInit*( )**, which initializes and enables the MMU.

RETURNS OK.

SEE ALSO **vmBaseLib**, *vmBaseGlobalMapInit*( )

vmBasePageSizeGet( )

NAME *vmBasePageSizeGet*( ) – return the page size

SYNOPSIS

```
int vmBasePageSizeGet (void)
```

DESCRIPTION This routine returns the architecture-dependent page size.

This routine is callable from interrupt level.

RETURNS The page size of the current architecture.

SEE ALSO **vmBaseLib**

vmBaseStateSet( )

NAME *vmBaseStateSet*( ) – change the state of a block of virtual memory

SYNOPSIS

```
STATUS vmBaseStateSet
    (
    VM_CONTEXT_ID context,   /* context - NULL == currentContext */
    void *        pVirtual,  /* virtual address to modify state of */
    int           len,       /* len of virtual space to modify state of */
    UINT          stateMask, /* state mask */
    UINT          state      /* state */
    )
```

DESCRIPTION This routine changes the state of a block of virtual memory. Each page of virtual memory has at least three elements of state information: validity, writability, and cacheability. Specific architectures may define additional state information; see **vmLib.h** for additional architecture-specific states. Memory accesses to a page marked as invalid will result in an exception. Pages may be invalidated to prevent them from being corrupted by invalid references. Pages may be defined as read-only or writable, depending on the state of the writable bits. Memory accesses to pages marked as not-cacheable will always result in a memory cycle, bypassing the cache. This is useful for multiprocessing, multiple bus masters, and hardware control registers.

The following states are provided and may be or'ed together in the state parameter:

VM_STATE_VALID	**VM_STATE_VALID_NOT**	valid/invalid
VM_STATE_WRITABLE	**VM_STATE_WRITABLE_NOT**	writable/write-protected
VM_STATE_CACHEABLE	**VM_STATE_CACHEABLE_NOT**	cacheable/not-cacheable

Additionally, the following masks are provided so that only specific states may be set. These may be or'ed together in the **stateMask** parameter.

VM_STATE_MASK_VALID
VM_STATE_MASK_WRITABLE
VM_STATE_MASK_CACHEABLE

If *context* is specified as NULL, the current context is used.

This routine is callable from interrupt level.

RETURNS OK, or ERROR if the validation fails, *pVirtual* is not on a page boundary, *len* is not a multiple of the page size, or the architecture-dependent state set fails for the specified virtual address.

ERRNO **S_vmLib_NOT_PAGE_ALIGNED, S_vmLib_BAD_STATE_PARAM, S_vmLib_BAD_MASK_PARAM**

SEE ALSO **vmBaseLib**

vmContextCreate( )

NAME

vmContextCreate( ) – create a new virtual memory context (VxVMI Opt.)

SYNOPSIS

```
VM_CONTEXT_ID vmContextCreate (void)
```

DESCRIPTION

This routine creates a new virtual memory context. The newly created context does not become the current context until explicitly installed by a call to ***vmCurrentSet*( )**. Modifications to the context state (mappings, state changes, etc.) may be performed on any virtual memory context, even if it is not the current context.

This routine should not be called from interrupt level.

AVAILABILITY

This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS

A pointer to a new virtual memory context, or NULL if the allocation or initialization fails.

SEE ALSO

vmLib

vmContextDelete( )

NAME

vmContextDelete( ) – delete a virtual memory context (VxVMI Opt.)

SYNOPSIS

```
STATUS vmContextDelete
    (
    VM_CONTEXT_ID context
    )
```

DESCRIPTION

This routine deallocates the underlying translation table associated with a virtual memory context. It does not free the physical memory already mapped to the virtual memory space.

This routine should not be called from interrupt level.

AVAILABILITY

This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS

OK, or ERROR if *context* is not a valid context descriptor or if an error occurs deleting the translation table.

SEE ALSO

vmLib

vmContextShow( )

NAME *vmContextShow*( ) – display the translation table for a context (VxVMI Opt.)

SYNOPSIS

```
STATUS vmContextShow
    (
    VM_CONTEXT_ID context /* context - NULL == currentContext */
    )
```

DESCRIPTION This routine displays the translation table for a specified context. If *context* is specified as NULL, the current context is displayed. Output is formatted to show blocks of virtual memory with consecutive physical addresses and the same state. State information shows the writable and cacheable states. If the block is in global virtual memory, the word "global" is appended to the line. Only virtual memory that has its valid state bit set is displayed.

This routine should be used for debugging purposes only.

Note that this routine cannot report non-standard architecture-dependent states.

AVAILABILITY This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS OK, or ERROR if the virtual memory context is invalid.

SEE ALSO **vmShow**

vmCurrentGet( )

NAME *vmCurrentGet*( ) – get the current virtual memory context (VxVMI Opt.)

SYNOPSIS

```
VM_CONTEXT_ID vmCurrentGet  (void)
```

DESCRIPTION This routine returns the current virtual memory context.

This routine is callable from interrupt level.

AVAILABILITY This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS The current virtual memory context, or NULL if no virtual memory context is installed.

SEE ALSO **vmLib**

vmCurrentSet( )

NAME

vmCurrentSet( ) – set the current virtual memory context (VxVMI Opt.)

SYNOPSIS

```
STATUS vmCurrentSet
    (
    VM_CONTEXT_ID context /* context to install */
    )
```

DESCRIPTION

This routine installs a specified virtual memory context.

This routine is callable from interrupt level.

AVAILABILITY

This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS

OK, or ERROR if the validation or context switch fails.

SEE ALSO

vmLib

vmEnable( )

NAME

vmEnable( ) – enable or disable virtual memory (VxVMI Opt.)

SYNOPSIS

```
STATUS vmEnable
    (
    BOOL enable /* TRUE == enable MMU, FALSE == disable MMU */
    )
```

DESCRIPTION

This routine turns virtual memory on and off. Memory management should not be turned off once it is turned on except in the case of system shutdown.

This routine is callable from interrupt level.

AVAILABILITY

This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS

OK, or ERROR if the validation or architecture-dependent code fails.

SEE ALSO

vmLib

2

vmGlobalInfoGet( )

NAME *vmGlobalInfoGet*( ) – get global virtual memory information (VxVMI Opt.)

SYNOPSIS

```
UINT8 *vmGlobalInfoGet  (void)
```

DESCRIPTION This routine provides a description of those parts of the virtual memory space dedicated to global memory. The routine returns a pointer to an array of UINT8. Each element of the array corresponds to a block of virtual memory, the size of which is architecture-dependent and can be obtained with a call to ***vmPageBlockSizeGet*( )**. To determine if a particular address is in global virtual memory, use the following code:

```
UINT8 *globalPageBlockArray = vmGlobalInfoGet ();
int pageBlockSize = vmPageBlockSizeGet ();

if (globalPageBlockArray[addr/pageBlockSize])
   ...
```

The array pointed to by the returned pointer is guaranteed to be static as long as no calls are made to ***vmGlobalMap*( )** while the array is being examined. The information in the array can be used to determine what portions of the virtual memory space are available for use as private virtual memory within a virtual memory context.

This routine is callable from interrupt level.

AVAILABILITY This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS A pointer to an array of UINT8.

SEE ALSO **vmLib**, *vmPageBlockSizeGet*( )

vmGlobalMap( )

NAME *vmGlobalMap*( ) – map physical pages to virtual space in shared global virtual memory (VxVMI Opt.)

SYNOPSIS

```
STATUS vmGlobalMap
    (
    void * virtualAddr,  /* virtual address */
    void * physicalAddr, /* physical address */
    UINT   len           /* len of virtual and physical spaces */
    )
```

DESCRIPTION

This routine maps physical pages to virtual space that is shared by all virtual memory contexts. Calls to ***vmGlobalMap*( )** should be made before any virtual memory contexts are created to insure that the shared global mappings are included in all virtual memory contexts. Mappings created with ***vmGlobalMap*( )** after virtual memory contexts are created are not guaranteed to appear in all virtual memory contexts. After the call to ***vmGlobalMap*( )**, the state of all pages in the the newly mapped virtual memory is unspecified and must be set with a call to ***vmStateSet*( )**, once the initial virtual memory context is created.

This routine should not be called from interrupt level.

AVAILABILITY

This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS

OK, or ERROR if *virtualAddr* or *physicalAddr* are not on page boundaries, *len* is not a multiple of the page size, or the mapping fails.

ERRNO

S_vmLib_NOT_PAGE_ALIGNED

SEE ALSO

vmLib

vmGlobalMapInit( )

NAME

***vmGlobalMapInit*( )** – initialize global mapping (VxVMI Opt.)

SYNOPSIS

```
VM_CONTEXT_ID vmGlobalMapInit
    (
    PHYS_MEM_DESC * pMemDescArray,       /* pointer to array of mem descs */
    int             numDescArrayElements,/* # of elements in pMemDescArray */
    BOOL            enable               /* enable virtual memory */
    )
```

DESCRIPTION

This routine is a convenience routine that creates and installs a virtual memory context with global mappings defined for each contiguous memory segment defined in the physical memory descriptor array passed as an argument. The context ID returned becomes the current virtual memory context.

The physical memory descriptor also contains state information used to initialize the state information in the MMU's translation table for that memory segment. The following state bits may be or'ed together:

VM_STATE_VALID	**VM_STATE_VALID_NOT**	valid/invalid
VM_STATE_WRITABLE	**VM_STATE_WRITABLE_NOT**	writable/write-protected
VM_STATE_CACHEABLE	**VM_STATE_CACHEABLE_NOT**	cacheable/not-cacheable

Additionally, mask bits are or'ed together in the **initialStateMask** structure element to describe which state bits are being specified in the **initialState**structure element:

VM_STATE_MASK_VALID
VM_STATE_MASK_WRITABLE
VM_STATE_MASK_CACHEABLE

If the *enable* parameter is TRUE, the MMU is enabled upon return. The ***vmGlobalMapInit*( )** routine should be called only after ***vmLibInit*( )** has been called.

AVAILABILITY
This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS
A pointer to a newly created virtual memory context, or NULL if the memory cannot be mapped.

SEE ALSO
vmLib

---

vmLibInit( )

NAME
***vmLibInit*( )** – initialize the virtual memory support module (VxVMI Opt.)

SYNOPSIS
```
STATUS vmLibInit
    (
    int pageSize /* size of page */
    )
```

DESCRIPTION
This routine initializes the virtual memory context class. It is called only once during system initialization.

AVAILABILITY
This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS
OK.

SEE ALSO
vmLib

vmMap( )

NAME

vmMap( ) – map physical space into virtual space (VxVMI Opt.)

SYNOPSIS

```
STATUS vmMap
    (
    VM_CONTEXT_ID context,      /* context - NULL == currentContext */
    void *        virtualAddr,  /* virtual address */
    void *        physicalAddr, /* physical address */
    UINT          len           /* len of virtual and physical spaces */
    )
```

DESCRIPTION

This routine maps physical pages into a contiguous block of virtual memory. *virtualAddr* and *physicalAddr* must be on page boundaries, and *len* must be evenly divisible by the page size. After the call to ***vmMap*( )**, the state of all pages in the the newly mapped virtual memory is valid, writable, and cacheable.

The ***vmMap*( )** routine can fail if the specified virtual address space conflicts with the translation tables of the global virtual memory space. The global virtual address space is architecture-dependent and is initialized at boot time with calls to ***vmGlobalMap*( )** by ***vmGlobalMapInit*( )**. If a conflict results, **errno** is set to **S_vmLib_ADDR_IN_GLOBAL_SPACE**. To avoid this conflict, use ***vmGlobalInfoGet*( )** to ascertain which portions of the virtual address space are reserved for the global virtual address space. If *context* is specified as NULL, the current virtual memory context is used.

This routine should not be called from interrupt level.

AVAILABILITY

This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS

OK, or ERROR if *virtualAddr* or *physicalAddr* are not on page boundaries, *len* is not a multiple of the page size, the validation fails, or the mapping fails.

ERRNO

S_vmLib_NOT_PAGE_ALIGNED, **S_vmLib_ADDR_IN_GLOBAL_SPACE**

SEE ALSO

vmLib

vmPageBlockSizeGet( )

NAME
vmPageBlockSizeGet( ) – get the architecture-dependent page block size (VxVMI Opt.)

SYNOPSIS
```
int vmPageBlockSizeGet  (void)
```

DESCRIPTION
This routine returns the size of a page block for the current architecture. Each MMU architecture constructs translation tables such that a minimum number of pages are pre-defined when a new section of the translation table is built. This minimal group of pages is referred to as a "page block." This routine may be used in conjunction with ***vmGlobalInfoGet*( )** to examine the layout of global virtual memory.

This routine is callable from interrupt level.

AVAILABILITY
This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS
The page block size of the current architecture.

SEE ALSO
vmLib, *vmGlobalInfoGet*( )

vmPageSizeGet( )

NAME
vmPageSizeGet( ) – return the page size (VxVMI Opt.)

SYNOPSIS
```
int vmPageSizeGet (void)
```

DESCRIPTION
This routine returns the architecture-dependent page size.

This routine is callable from interrupt level.

AVAILABILITY
This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS
The page size of the current architecture.

SEE ALSO
vmLib

vmShowInit( )

NAME

vmShowInit( ) – include virtual memory show facility (VxVMI Opt.)

SYNOPSIS

```
void vmShowInit (void)
```

DESCRIPTION

This routine acts as a hook to include *vmContextShow*( ). It is called automatically when the virtual memory show facility is configured into VxWorks using either of the following methods:

– If you use the configuration header files, define both **INCLUDE_MMU_FULL** and **INCLUDE_SHOW_ROUTINES** in **config.h**.

– If you use the Tornado project facility, select **INCLUDE_MMU_FULL_SHOW**.

AVAILABILITY

This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS

N/A

SEE ALSO

vmShow

vmStateGet( )

NAME

vmStateGet( ) – get the state of a page of virtual memory (VxVMI Opt.)

SYNOPSIS

```
STATUS vmStateGet
    (
    VM_CONTEXT_ID context,   /* context - NULL == currentContext */
    void *        pPageAddr, /* virtual page addr */
    UINT *        pState     /* where to return state */
    )
```

DESCRIPTION

This routine extracts state bits with the following masks:

VM_STATE_MASK_VALID
VM_STATE_MASK_WRITABLE
VM_STATE_MASK_CACHEABLE

Individual states may be identified with the following constants:

VM_STATE_VALID	**VM_STATE_VALID_NOT**	valid/invalid
VM_STATE_WRITABLE	**VM_STATE_WRITABLE_NOT**	writable/write-protected
VM_STATE_CACHEABLE	**VM_STATE_CACHEABLE_NOT**	cacheable/not-cacheable

For example, to see if a page is writable, the following code would be used:

```
vmStateGet (vmContext, pageAddr, &state);
if ((state & VM_STATE_MASK_WRITABLE) & VM_STATE_WRITABLE)
   ...
```

If *context* is specified as NULL, the current virtual memory context is used.

This routine is callable from interrupt level.

AVAILABILITY
This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS
OK, or ERROR if *pageAddr* is not on a page boundary, the validity check fails, or the architecture-dependent state get fails for the specified virtual address.

ERRNO
S_vmLib_NOT_PAGE_ALIGNED

SEE ALSO
vmLib

vmStateSet( )

NAME
vmStateSet( ) – change the state of a block of virtual memory (VxVMI Opt.)

SYNOPSIS

```
STATUS vmStateSet
    (
    VM_CONTEXT_ID context,   /* context - NULL == currentContext */
    void *        pVirtual,  /* virtual address to modify state of */
    int           len,       /* len of virtual space to modify state of */
    UINT          stateMask, /* state mask */
    UINT          state      /* state */
    )
```

DESCRIPTION
This routine changes the state of a block of virtual memory. Each page of virtual memory has at least three elements of state information: validity, writability, and cacheability. Specific architectures may define additional state information; see **vmLib.h** for additional architecture-specific states. Memory accesses to a page marked as invalid will result in an exception. Pages may be invalidated to prevent them from being corrupted by invalid references. Pages may be defined as read-only or writable, depending on the state of the writable bits. Memory accesses to pages marked as not-cacheable will always result in a memory cycle, bypassing the cache. This is useful for multiprocessing, multiple bus masters, and hardware control registers.

The following states are provided and may be or'ed together in the state parameter:

VM_STATE_VALID	**VM_STATE_VALID_NOT**	valid/invalid
VM_STATE_WRITABLE	**VM_STATE_WRITABLE_NOT**	writable/write-protected
VM_STATE_CACHEABLE	**VM_STATE_CACHEABLE_NOT**	cacheable/not-cacheable

Additionally, the following masks are provided so that only specific states may be set. These may be or'ed together in the **stateMask** parameter.

VM_STATE_MASK_VALID
VM_STATE_MASK_WRITABLE
VM_STATE_MASK_CACHEABLE

If *context* is specified as NULL, the current context is used.

This routine is callable from interrupt level.

AVAILABILITY
This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS
OK or, ERROR if the validation fails, *pVirtual* is not on a page boundary, *len* is not a multiple of page size, or the architecture-dependent state set fails for the specified virtual address.

ERRNO
S_vmLib_NOT_PAGE_ALIGNED, **S_vmLib_BAD_STATE_PARAM**, **S_vmLib_BAD_MASK_PARAM**

SEE ALSO
vmLib

vmTextProtect( )

NAME
vmTextProtect( ) – write-protect a text segment (VxVMI Opt.)

SYNOPSIS

```
STATUS vmTextProtect (void)
```

DESCRIPTION
This routine write-protects the VxWorks text segment and sets a flag so that all text segments loaded by the incremental loader will be write-protected. The routine should be called after both ***vmLibInit*( )** and ***vmGlobalMapInit*( )** have been called.

AVAILABILITY
This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS
OK, or ERROR if the text segment cannot be write-protected.

ERRNO S_vmLib_TEXT_PROTECTION_UNAVAILABLE

SEE ALSO **vmLib**

vmTranslate( )

NAME *vmTranslate*( ) – translate a virtual address to a physical address (VxVMI Opt.)

SYNOPSIS

```
STATUS vmTranslate
    (
    VM_CONTEXT_ID context,      /* context - NULL == currentContext */
    void *        virtualAddr,  /* virtual address */
    void *        *physicalAddr /* place to put result */
    )
```

DESCRIPTION This routine retrieves mapping information for a virtual address from the page translation tables. If the specified virtual address has never been mapped, the returned status can be either OK or ERROR; however, if it is OK, then the returned physical address will be -1. If *context* is specified as NULL, the current context is used.

This routine is callable from interrupt level.

AVAILABILITY This routine is distributed as a component of the unbundled virtual memory support option, VxVMI.

RETURNS OK, or ERROR if the validation or translation fails.

SEE ALSO **vmLib**

vprintf( )

NAME *vprintf*( ) – write a string formatted with a variable argument list to standard output (ANSI)

SYNOPSIS

```
int vprintf
    (
    const char * fmt,   /* format string to write */
    va_list      vaList /* arguments to format */
    )
```

DESCRIPTION This routine prints a string formatted with a variable argument list to standard output. It is identical to ***printf*( )**, except that it takes the variable arguments to be formatted as a list *vaList* of type **va_list**rather than as in-line arguments.

RETURNS The number of characters output, or ERROR if there is an error during output.

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

---

vsprintf( )

NAME ***vsprintf*( )** – write a string formatted with a variable argument list to a buffer (ANSI)

SYNOPSIS

```
int vsprintf
    (
    char *       buffer, /* buffer to write to */
    const char * fmt,    /* format string */
    va_list      vaList  /* optional arguments to format */
    )
```

DESCRIPTION This routine copies a string formatted with a variable argument list to a specified buffer. This routine is identical to ***sprintf*( )**, except that it takes the variable arguments to be formatted as a list *vaList* of type **va_list** rather than as in-line arguments.

RETURNS The number of characters copied to *buffer*, not including the NULL terminator.

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

---

vxMemArchProbe( )

NAME ***vxMemArchProbe*( )** – architecture specific part of vxMemProbe

SYNOPSIS

```
STATUS vxMemArchProbe
    (
    char * adrs,   /* address to be probed */
    int    mode,   /* VX_READ or VX_WRITE */
    int    length, /* 1, 2, 4, or 8 */
    char * pVal    /* where to return value, or ptr to value to be written */
    )
```

DESCRIPTION This is the routine implementing the architecture specific part of the vxMemProbe routine. It traps the relevant exceptions while accessing the specified address. If an exception occurs, then the result will be ERROR. If no exception occurs then the result will be OK.

RETURNS OK or ERROR if an exception occurred during access.

SEE ALSO **vxLib**

---

vxMemProbe( )

NAME *vxMemProbe*( ) – probe an address for a bus error

SYNOPSIS

```
STATUS vxMemProbe
    (
    char * adrs,   /* address to be probed */
    int    mode,   /* VX_READ or VX_WRITE */
    int    length, /* 1, 2, 4, or 8 */
    char * pVal    /* where to return value, or ptr to value to be written */
    )
```

DESCRIPTION This routine probes a specified address to see if it is readable or writable, as specified by *mode*. The address is read or written as 1, 2, or 4 bytes, as specified by *length* (values other than 1, 2, or 4 yield unpredictable results). If the probe is a **VX_READ** (0), the value read is copied to the location pointed to by *pVal*. If the probe is a **VX_WRITE** (1), the value written is taken from the location pointed to by *pVal*. In either case, *pVal* should point to a value of 1, 2, or 4 bytes, as specified by *length*.

Note that only bus errors are trapped during the probe, and that the access must otherwise be valid (i.e., it must not generate an address error).

EXAMPLE

```
testMem (adrs)
    char *adrs;
    {
    char testW = 1;
    char testR;
    if (vxMemProbe (adrs, VX_WRITE, 1, &testW) == OK)
        printf ("value %d written to adrs %x\n", testW, adrs);
    if (vxMemProbe (adrs, VX_READ, 1, &testR) == OK)
        printf ("value %d read from adrs %x\n", testR, adrs);
    }
```

MODIFICATION The BSP can modify the behaviour of ***vxMemProbe*( )** by supplying an alternate routine and placing the address in the global variable _func_vxMemProbeHook. The BSP routine will be called instead of the architecture specific routine ***vxMemArchProbe*( )**.

RETURNS OK, or ERROR if the probe caused a bus error or was misaligned.

SEE ALSO **vxLib**, ***vxMemArchProbe*( )**

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

```
UINT32 vxPowerDown (void)
```

DESCRIPTION This routine activates the reduced-power mode if power management is enabled. It is called by the scheduler when the kernel enters the idle loop. The power management mode is selected by ***vxPowerModeSet*( )**.

RETURNS OK, or ERROR if power management is not supported or if external interrupts are disabled.

SEE ALSO **vxLib**, *vxPowerModeSet*( ), *vxPowerModeGet*( )

STATUS vxPowerDown (void)

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, PPCEC603 and PPC860 only).

VX_POWER_MODE_DOZE (0x4)
Only the decrementer, data cache, and bus snooping are active while the kernel is idle (PPC603, PPCEC603 and PPC860).

VX_POWER_MODE_NAP (0x8)
Only the decrementer is active while the kernel is idle (PPC603, PPCEC603 and PPC604).

VX_POWER_MODE_SLEEP (0x10)
All CPU units are inactive while the kernel is idle (PPC603, PPCEC603 and PPC860 – not recommended for the PPC603 and PPCEC603 architecture).

VX_POWER_MODE_DEEP_SLEEP (0x20)
All CPU units are inactive while the kernel is idle (PPC860 only – not recommended).

VX_POWER_MODE_DPM (0x40)
Dynamic Power Management Mode (PPC603 and PPCEC603 only).

VX_POWER_MODE_DOWN (0x80)
Only a hard reset causes an exit from power-down low power mode (PPC860 only – not recommended).

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**
ARM **swpb**

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

VXWSem

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 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 ***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, though desirable, is not currently feasible. Without explicit knowledge of the state of the guarded resource or region, reckless automatic reclamation of a semaphore could leave the resource in a partial state. Thus, if a task ceases execution unexpectedly, as with a bus error, currently owned semaphores are not given back, effectively leaving a resource permanently unavailable. The mutual-exclusion semaphores provided by **VXWMSem** offer protection from unexpected task deletion.

RETURNS N/A

SEE ALSO **VXWSem**

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 **VXWList**

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 **VXWList**

VXWList::count( )

NAME *VXWList::count*( ) – report the number of nodes in a list (WFC Opt.)

SYNOPSIS

```
    int count ()
```

DESCRIPTION This routine returns the number of nodes in a specified list.

RETURNS The number of nodes in the list.

SEE ALSO **VXWList**

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 **VXWList**

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 **VXWList**

VXWList::first( )

NAME *VXWList::first*( ) – find first node in list (WFC Opt.)

SYNOPSIS

```
NODE * first ()
```

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 **VXWList**

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 **VXWList**

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 **VXWList**

VXWList::last( )

NAME *VXWList::last*( ) – find the last node in list (WFC Opt.)

SYNOPSIS

```
    NODE * last ()
```

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 **VXWList**

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 **VXWList**

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 **VXWList**

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 *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 **VXWList**

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 **VXWList**

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 **VXWList**

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 **VXWList**

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 **VXWList**

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 **VXWList**

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
VXWMemPart

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
VXWMemPart

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 *VXWMemPart::free*( )

VXWMemPart::findMax( )

NAME *VXWMemPart::findMax*( ) – find the size of the largest available free block (WFC Opt.)

SYNOPSIS

```
    int findMax ()
```

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 **VXWMemPart**

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 *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 *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 **VXWMemPart**

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 **VXWMemPart**

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

VXWMemPart

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

VXWMemPart

VXWModule::flags( )

NAME *VXWModule::flags*( ) – get the flags associated with this module (WFC Opt.)

SYNOPSIS

```
    int flags ()
```

DESCRIPTION This routine returns the flags associated with its module.

RETURNS The option flags.

SEE ALSO **VXWModule**

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 **VXWModule**

VXWModule::name( )

NAME *VXWModule::name*( ) – get the name associated with module (WFC Opt.)

SYNOPSIS

```
    char * name ()
```

DESCRIPTION This routine returns a pointer to the name associated with its module.

RETURNS A pointer to the module name.

SEE ALSO **VXWModule**

VXWModule::segFirst( )

NAME *VXWModule::segFirst*( ) – find the first segment in module (WFC Opt.)

SYNOPSIS

```
    SEGMENT_ID segFirst ()
```

DESCRIPTION This routine returns information about the first segment of a module descriptor.

RETURNS A pointer to the segment ID.

SEE ALSO *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 *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 **VXWModule**

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 **VXWModule**, **loadLib**

VXWModule::VXWModule( )

NAME *VXWModule::VXWModule*( ) – load object module at 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 added to the symbol table to indicate the start of each segment: *file*_text, *file*_data, and *file*_bss, where *file* 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 **VXWModule**, *VxWorks Programmer's Guide: C++ 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 **VXWModule**

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 **VXWModule**

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 **VXWModule**

VXWMSem::giveForce( )

NAME *VXWMSem::giveForce*( ) – give mutex 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 *VXWSem::give*( )

VXWMSem::VXWMSem( )

NAME *VXWMSem::VXWMSem*( ) – create and initialize a mutex 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

VXWSem, ***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 **VXWMsgQ**

VXWMsgQ::numMsgs( )

NAME *VXWMsgQ::numMsgs*( ) – report the number of messages queued (WFC Opt.)

SYNOPSIS

```
    int numMsgs ()
```

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 **VXWMsgQ**

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

VXWMsgQ

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 any tasks are already waiting to receive messages on the queue, the message is immediately delivered to the first waiting task. If no task is waiting to receive messages, the message is saved in the message queue.

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

VXWMsgQ

2

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

VXWMsgQ

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

VXWMsgQ, **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 **VXWMsgQ**, **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 **VXWMsgQ**

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 **VXWRingBuf**

VXWRingBuf::freeBytes( )

NAME *VXWRingBuf::freeBytes*( ) – determine the number of free bytes in ring buffer (WFC Opt.)

SYNOPSIS

```
int freeBytes ()
```

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 **VXWRingBuf**

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 **VXWRingBuf**

VXWRingBuf::isEmpty( )

NAME *VXWRingBuf::isEmpty*( ) – test whether ring buffer is empty (WFC Opt.)

SYNOPSIS

```
BOOL isEmpty ()
```

DESCRIPTION This routine reports on whether the ring buffer is empty.

RETURNS TRUE if empty, FALSE if not.

SEE ALSO **VXWRingBuf**

VXWRingBuf::isFull( )

NAME *VXWRingBuf::isFull*( ) – test whether ring buffer is full (no more room) (WFC Opt.)

SYNOPSIS

```
    BOOL isFull ()
```

DESCRIPTION This routine reports on whether the ring buffer is completely full.

RETURNS TRUE if full, FALSE if not.

SEE ALSO **VXWRingBuf**

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 **VXWRingBuf**

VXWRingBuf::nBytes( )

NAME *VXWRingBuf::nBytes*( ) – determine the number of bytes in ring buffer (WFC Opt.)

SYNOPSIS

```
int nBytes ()
```

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 **VXWRingBuf**

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 **VXWRingBuf**

2

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 **VXWRingBuf**

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 **VXWRingBuf**

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 permits you to use the C++ ring-buffer interfaces even if the ring buffer itself was created by a routine written in C.

RETURNS
N/A.

SEE ALSO
VXWRingBuf, **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
VXWRingBuf

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 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 **VXWSem**, ***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 **VXWSem**, ***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 ()
```

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 **VXWSem**, **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 **VXWSem**

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
VXWSem

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
VXWSem, ***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

VXWSem, **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

VXWSem, *VxWorks Programmer's Guide: Basic OS*

VXWSmName::nameGet( )

NAME

VXWSmName::nameGet( ) – get name and type of a shared memory object (VxMP Opt.)

SYNOPSIS

```
STATUS nameGet
    (
    char * name,
    int *  pType,
    int    waitType
    )
```

DESCRIPTION

This routine searches the shared memory name database for an object matching 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

VXWSmName

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 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

VXWSmName

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
VXWSmName

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
VXWSmName

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
VXWSymTab, **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
VXWSymTab

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 **VXWSymTab**

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 **VXWSymTab**

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 actual 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

VXWSymTab

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.

2

RETURNS OK, or ERROR if *value* is less than the lowest value in the table.

SEE ALSO **VXWSymTab**

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 **VXWSymTab**

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 **VXWSymTab**

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 **VXWSymTab**

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 **VXWSymTab**

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 *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 ***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 **VXWTask**, **envLib**

VXWTask::errNo( )

NAME *VXWTask::errNo*( ) – retrieve error status value (WFC Opt.)

SYNOPSIS

```
    int errNo ()
```

DESCRIPTION This routine gets the error status for the task.

RETURNS The error status value contained in **errno**.

SEE ALSO **VXWTask**

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 **VXWTask**

VXWTask::id( )

NAME *VXWTask::id*( ) – reveal task ID (WFC Opt.)

SYNOPSIS

```
int id ()
```

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 **VXWTask**, **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_TCB**s should be restricted to debugging aids.

RETURNS OK

SEE ALSO **VXWTask**

VXWTask::isReady( )

NAME *VXWTask::isReady*( ) – check if task is ready to run (WFC Opt.)

SYNOPSIS

```
    BOOL isReady ()
```

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 **VXWTask**

VXWTask::isSuspended( )

NAME *VXWTask::isSuspended*( ) – check if task is suspended (WFC Opt.)

SYNOPSIS

```
    BOOL isSuspended ()
```

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 **VXWTask**

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 **VXWTask**

VXWTask::name( )

NAME *VXWTask::name***( )** – get the name associated with a task ID (WFC Opt.)

SYNOPSIS

```
char * name ()
```

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 **VXWTask**

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 **VXWTask**

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 **VXWTask**

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 **VXWTask**

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 **VXWTask**

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 *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 *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 **VXWTask**

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 **VXWTask**

VXWTask::show( )

NAME *VXWTask::show*( ) – display the contents of task registers (WFC Opt.)

SYNOPSIS

```
void show ()
```

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 **VXWTask**

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*( )**
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

***VXWTaskstatusString*( )**, *Tornado User's Guide: The Tornado Shell*

2

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 **VXWTask**

VXWTask::SRSet( )

NAME *VXWTask::SRSet*( ) – set the task status register (MC680x0, MIPS, i386/i486) (WFC Opt.)

SYNOPSIS

```
STATUS SRSet
    (
    UINT16 sr
    )
```

SYNOPSIS (I80X86)

```
STATUS SRSet
    (
    UINT sr
    )
```

SYNOPSIS (MIPS)

```
STATUS SRSet
    (
    UINT32 sr
    )
```

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 **VXWTask**

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 **VXWTask**

2

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 **VXWTask**

VXWTask::tcb( )

NAME *VXWTask::tcb*( ) – get the task control block (WFC Opt.)

SYNOPSIS

```
WIND_TCB * tcb ()
```

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 **VXWTask**

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 *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 *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
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
VXWTask

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 *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 **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 ***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 ***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 **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 ***VXWWd::start*( )**

2

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 *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 *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

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

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:

2

```
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 **wd33c93Lib**1, **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 **wd33c93Lib**2, 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 memory is insufficient or the parameters are invalid.

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**

wdbPipePktDevInit( )

NAME *wdbPipePktDevInit*( ) – initialize a pipe packet device.

SYNOPSIS

```
STATUS wdbPipePktDevInit
    (
    WDB_PIPE_PKT_DEV * pPktDev,    /* pipe device structure to init */
    void (*            stackRcv)() /* receive packet callback (udpRcv) */
    )
```

SEE ALSO **wdbPipePktDrv**

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**

wdbSystemSuspend( )

NAME *wdbSystemSuspend*( ) – suspend the system.

SYNOPSIS

```
STATUS wdbSystemSuspend (void)
```

DESCRIPTION
This routine transfers control from the run time system to the WDB agent running in external mode. In order to give back the control to the system it must be resumed by the the external WDB agent.

EXAMPLE
The code below, called in a vxWorks application, suspends the system :

```
if (wdbSystemSuspend != OK)
    printf ("External mode is not supported by the WDB agent.\n");
```

From a host tool, we can detect that the system is suspended.

First, attach to the target server :

```
wtxtcl> wtxToolAttach EP960CX
EP960CX_ps@sevre
```

Then, you can get the agent mode :

```
wtxtcl> wtxAgentModeGet
AGENT_MODE_EXTERN
```

To get the status of the system context, execute :

```
wtxtcl> wtxContextStatusGet CONTEXT_SYSTEM 0
CONTEXT_SUSPENDED
```

In order to resume the system, simply execute :

```
wtxtcl>  wtxContextResume CONTEXT_SYSTEM 0
0
```

You will see that the system is now running :

```
wtxtcl> wtxContextStatusGet CONTEXT_SYSTEM 0
CONTEXT_RUNNING
```

RETURNS
OK upon successful completion, ERROR if external mode is not supported by the WDB agent.

SEE ALSO
wdbLib

wdbTsfsDrv( )

NAME *wdbTsfsDrv*( ) – initialize the TSFS device driver for a WDB agent

SYNOPSIS

```
STATUS wdbTsfsDrv
    (
    char * name /* root name in i/o system */
    )
```

DESCRIPTION This routine initializes the VxWorks virtual I/O "2" driver and creates a TSFS 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 **/tgtsvr**.

After this routine has been called, individual virtual I/O channels can be opened by appending the host file name to the virtual I/O device name. For example, to get a file descriptor for the host file **/etc/passwd**, call ***open*( )** as follows:

```
fd = open ("/tgtsvr/etc/passwd", O_RDWR, 0)
```

RETURNS OK, or ERROR if the driver can not be installed.

SEE ALSO **wdbTsfsDrv**

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)()/* routine 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**

wdbUserEvtLibInit( )

NAME *wdbUserEvtLibInit*( ) – include the WDB user event library

SYNOPSIS

```
void wdbUserEvtLibInit (void)
```

DESCRIPTION This null routine is provided so that **wdbUserEvtLib** can be linked into the system. If **INCLUDE_WDB_USER_EVENT** is defined in **configAll.h**, wdbUserEvtLibInit is called by the WDB config routine, *wdbConfig*( ), in **usrWdb.c**.

RETURNS N/A

SEE ALSO **wdbUserEvtLib**

wdbUserEvtPost( )

NAME *wdbUserEvtPost*( ) – post a user event string to host tools.

SYNOPSIS

```
STATUS wdbUserEvtPost
    (
    char * event /* event string to send */
    )
```

DESCRIPTION This routine posts the string *event* to host tools that have registered for it. Host tools will receive a USER WTX event string. The maximum size of the event is **WDB_MAX_USER_EVT_SIZE** (defined in $**WIND_BASE/target/h/wdb/wdbLib.h**).

EXAMPLE The code below sends a WDB user event to host tools :

```
char * message = "Alarm: reactor overheating !!!";
if (wdbUserEvtPost (message) != OK)
    printf ("Can't send alarm message to host tools");
```

This event will be received by host tools that have registered for it. For example a WTX TCL based tool would do :

```
wtxtcl> wtxToolAttach EP960CX
EP960CX_ps@sevre
```

```
wtxtcl> wtxRegisterForEvent "USER.*"
0
wtxtcl> wtxEventGet
USER Alarm: reactor overheating !!!
```

Host tools can register for more specific user events :

```
wtxtcl> wtxToolAttach EP960CX
EP960CX_ps@sevre
wtxtcl> wtxRegisterForEvent "USER Alarm.*"
0
wtxtcl> wtxEventGet
USER Alarm: reactor overheating !!!
```

In this piece of code, only the USER events beginning with "Alarm" will be received.

RETURNS OK upon successful completion, a WDB error code if unable to send the event to the host or ERROR if the size of the event is greater than **WDB_MAX_USER_EVT_SIZE**.

SEE ALSO **wdbUserEvtLib**

---

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. It is called automatically when the watchdog show facility is configured into VxWorks using either of the following methods:

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

– If you use the Tornado project facility, select **INCLUDE_WATCHDOGS_SHOW**.

RETURNS N/A

SEE ALSO **wdShow**

---

wdStart( )

NAME *wdStart*( ) – start a watchdog timer

SYNOPSIS

```
STATUS wdStart
    (
    WDOG_ID wdId,     /* watchdog ID */
    int     delay,    /* delay count, in ticks */
    FUNCPTR pRoutine, /* routine to call on time-out */
    int     parameter /* parameter with which to call routine */
    )
```

DESCRIPTION This routine adds a watchdog timer to the system tick queue. The specified watchdog routine will be called from interrupt level after the specified number of ticks has elapsed. Watchdog timers may be started from interrupt level.

To replace either the timeout *delay* or the routine to be executed, call ***wdStart*( )** again with the same *wdId*; only the most recent ***wdStart*( )** on a given watchdog ID has any effect. (If your application requires multiple watchdog routines, use ***wdCreate*( )** to generate separate a watchdog ID for each.) To cancel a watchdog timer before the specified tick count is reached, call ***wdCancel*( )**.

Watchdog timers execute only once, but some applications require periodically executing timers. To achieve this effect, the timer routine itself must call ***wdStart*( )** to restart the timer on each invocation.

WARNING The watchdog routine runs in the context of the system-clock ISR; thus, it is subject to all ISR restrictions.

RETURNS OK, or ERROR if the watchdog timer cannot be started.

SEE ALSO **wdLib**, *wdCancel*( )

whoami( )

NAME
whoami( ) – display the current remote identity

SYNOPSIS

```
void whoami (void)
```

DESCRIPTION
This routine displays the user name currently used for remote machine access. The user name is set with ***iam*( )** or ***remCurIdSet*( )**.

RETURNS
N/A

SEE ALSO
remLib, ***iam*( )**, ***remCurIdGet*( )**, ***remCurIdSet*( )**

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*

winDevInit( )

NAME
winDevInit( ) – initialize a **WIN_CHAN**

SYNOPSIS

```
void winDevInit
    (
    WIN_CHAN * pChan
    )
```

DESCRIPTION This routine initializes the driver function pointers and then resets the chip in a quiescent state. The BSP must have already initialized all the device addresses and the baudFreq fields in the **WIN_CHAN** structure before passing it to this routine.

RETURNS N/A

SEE ALSO **winSio**

winDevInit2( )

NAME *winDevInit2*( ) – initialize a **WIN_CHAN**, part 2

SYNOPSIS

```
void winDevInit2
    (
    WIN_CHAN * pChan /* device to initialize */
    )
```

DESCRIPTION This routine is called by the BSP after interrupts have been connected. The driver can now operate in interrupt mode. Before this routine is called only polled mode operations should be allowed.

RETURNS N/A

SEE ALSO **winSio**

winIntRcv( )

NAME *winIntRcv*( ) – handle a channel's receive-character interrupt

SYNOPSIS

```
void winIntRcv
    (
    WIN_CHAN * pChan, /* channel generating the interrupt */
    UINT16     wparam /* message args get passed if you look */
    )
```

DESCRIPTION This function is attached to the simulator's interrupt handler, and passes the character received in the message to the callback.

RETURNS N/A

SEE ALSO **winSio**

winIntTx( )

NAME *winIntTx*( ) – transmit a single character.

SYNOPSIS

```
void winIntTx
    (
    WIN_CHAN * pChan /* channel generating the interrupt */
    )
```

DESCRIPTION This displays a single character to the simulator's window.

RETURNS N/A

SEE ALSO **winSio**

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**

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 to a quiescent state.

RETURNS N/A

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.

RETURNS N/A

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**

2

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 page for a full 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 the configuration macro **INCLUDE_ZBUF_SOCK** is defined.

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

IX

IX

IX

IX

IX

IX

IX

IX

IX

IX

IX

IX

IX

IX