Vx960™

REFERENCE
MANUAL

Intel Corporation

Master Index

Master Index

abs - absolute value routine 1-1
abs( ) - absolute value of integer 1-1
accept( ) - accept connection from socket 1-354
acw( ) - return contents of ACW register 1-508
arptabShow( ) - display known ARP entries 1-246

bLib - buffer manipulation library 1-2
bcmp( ) - compare one buffer to another 1-3
bcopyBytes( ) - copy one buffer to another one byte at a time 1-6
bcopyLongs( ) - copy one buffer to another one long word at a time 1-7
bcopyWords( ) - copy one buffer to another one word at a time 1-7
bcopy( ) - copy one buffer to another 1-6
bdall( ) - delete all breakpoints 1-28
bd( ) - delete breakpoint 1-28
b( ) - set or display breakpoints 1-26
bfillBytes( ) - fill buffer with specified character one byte at a time 1-9
bfill( ) - fill buffer with specified character 1-8
bind( ) - bind name to socket 1-352
bindresvport( ) - bind socket to privileged IP port 1-269
binvert( ) - invert order of bytes in buffer 1-4
bootBpAnchorExtract( ) - extract backplane address from device field 1-18
bootChange( ) - change boot line 1-484
bootConfig - system configuration module for boot ROMs 1-10
bootInit - ROM initialization module 1-11
bootLib - boot ROM subroutine library 1-13
bootNetmaskExtract( ) - extract netmask field from Internet address 1-17
bootParamsPrompt( ) - prompt for boot line parameters 1-16
bootParamsShow( ) - display boot line parameters 1-16
bootStringToStruct( ) - interpret boot parameters from boot line 1-15

bootStructToString( )	- construct boot line	1-15
bpInit( )	- initialize backplane anchor	1-127
bpShow( )	- show information about backplane network	1-128
bpattach( )	- attach *bp* interface to network	1-127
bswap( )	- swap buffers	1-4
bzero( )	- zero out buffer	1-5
cALib	- C-language support routines	1-19
calloc( )	- allocate space for array	1-225
cd( )	- change default directory	1-494
c( )	- continue from breakpoint	1-29
cfree( )	- free block of memory	1-227
chdir( )	- set current default path	1-167
checkStack( )	- print summary of each task's stack usage	1-488
clearerr( )	- reset error and end-of-file indicators	1-385
closedir( )	- close directory	1-38
close( )	- close file	1-163
compress	- general purpose file compression utility	3-1
connect( )	- initiate connection to socket	1-354
copyStreams( )	- copy from/to specified streams	1-497
copy( )	- copy *in* (or *stdin*) to *out* (or *stdout*)	1-496
creat( )	- create file	1-160
cret( )	- continue until return from current subroutine	1-29
ctypeLib	- character classification and conversion macros	1-21
db( )	- set data breakpoint	1-27
dbgHelp( )	- display debugging help menu	1-24
dbgInit( )	- initialize debug package	1-25
dbgLib	- debugging facilities	1-22
delete( )	- delete file	1-161
devs( )	- list all system-known devices	1-504
d( )	- display memory	1-494
dirLib	- POSIX directory handling library	1-34
diskFormat( )	- format disk	1-498
diskInit( )	- initialize file system on block device	1-498
dosFsConfigInit( )	- initialize dosFs volume configuration structure	1-53
dosFsDateSet( )	- set current date	1-54
dosFsDateTimeInstall( )	- install user-supplied date/time function	1-54
dosFsDevInit( )	- associate block device with dosFs file system functions.	1-55
dosFsInit( )	- prepare to use dosFs library	1-56
dosFsLib	- MS-DOS media-compatible file system library	1-40
dosFsMkfs( )	- initialize device and create dosFs file system	1-57

dosFsModeChange( ) - modify mode of dosFs volume 1-58
dosFsReadyChange( ) - notify **dosFsLib** of change in ready status 1-58
dosFsTimeSet( ) - set current time 1-59
dosFsVolUnmount( ) - unmount dosFs volume 1-60
dsm960Inst( ) - disassemble i960 object code 1-62
dsm960Lib - Vx960/80960 disassembler 1-61

enpattach( ) - attach *enp* interface to network 1-130
errnoGet( ) - get error status value of calling task 1-65
errnoLib - error status library 1-63
errnoOfTaskGet( ) - get error status value of specified task 1-65
errnoOfTaskSet( ) - set error status value of specified task 1-66
errnoSet( ) - set error status value of calling task 1-66
etherAddrResolve( ) - resolve Ethernet address for given Internet address 1-72
etherInputHookAdd( ) - add routine to receive all Ethernet input packets 1-70
etherLib - Ethernet raw I/O routines and hooks 1-68
etherOutputHookAdd( ) - add routine to receive all Ethernet output packets 1-71
etherOutput( ) - send packet on Ethernet interface 1-69
exattach( ) - attach *ex* interface to network 1-131
exc960Lib - architecture-dependent exception handling: i960 1-74
excHookAdd( ) - specify routine to be called with exceptions 1-78
excInfoShow( ) - display exception information 1-75
excInit( ) - initialize exception handling package 1-77
excLib - exception handling facilities 1-76
excNMIHookAdd( ) - add hook routine to be called in event of NMI 1-75
excTask( ) - handle task-level exceptions 1-79
excVecInit( ) - initialize exception/interrupt vectors 1-74
exit( ) - exit task 1-422
extract - conditionally extract files from archive 3-3

fclose( ) - empty stream buffers and close file 1-372
fdopen( ) - associate stream with fd 1-372
fdprintf( ) - print formatted string to specified fd 1-83
feof( ) - determine if end-of-file has been read 1-386
ferror( ) - determine if error has occurred while reading or writing 1-386
fflush( ) - write out any buffers on output stream 1-374
fgetc( ) - return next character in input stream 1-373
fgets( ) - read string from input stream 1-373
fileno( ) - get fd associated with stream 1-387
fioFormatV( ) - convert format string 1-86
fioLib - formatted I/O library 1-80

fioRdString( ) - read string from file 1-89
fioRead( ) - read buffer 1-89
floatInit( ) - initialize floating-point I/O support 1-91
floatLib - floating-point formatting and scanning library 1-91
fopen( ) - open stream on file 1-374
fppALib - floating-point support assembly language routines 1-93
fppLib - floating-point support library 1-94
fppProbe( ) - probe for presence of floating-point support 1-96
fppTaskRegsGet( ) - get floating-point registers from task TCB 1-95
fppTaskRegsSet( ) - set floating-point registers of task 1-96
fppTaskRegsShow( ) - display floating-point registers of task 1-95
fprintf( ) - print formatted string to stream 1-375
fputc( ) - append character to output stream 1-376
fputs( ) - copy NULL-terminated string to output stream 1-376
fread( ) - perform buffered read 1-377
free( ) - free block of memory 1-226
freopen( ) - substitute named file in place of open stream 1-377
fscanf( ) - read and convert characters from input stream 1-382
fseek( ) - reposition stream 1-378
fstat( ) - get file status information 1-38
ftell( ) - return current offset in stream 1-378
ftpCommand( ) - send FTP command and get reply 1-99
ftpDataConnGet( ) - get completed FTP data connection 1-105
ftpDataConnInit( ) - initialize FTP data connection 1-104
ftpHookup( ) - get control connection to FTP server on specified host ... 1-103
ftpLib - ARPA File Transfer Protocol library 1-98
ftpLogin( ) - log in to remote FTP server 1-104
ftpReplyGet( ) - get FTP command reply 1-102
ftpXfer( ) - initiate transfer via FTP 1-100
ftpdDelete( ) - clean up and finalize FTP server task 1-108
ftpdInit( ) - initialize FTP server task 1-108
ftpdLib - ARPA File Transfer Protocol server 1-106
ftpdTask( ) - FTP server daemon task 1-107
fwrite( ) - perform buffered write 1-379

getc( ) - return next character in input stream 1-388
getchar( ) - return next character in standard in stream 1-387
getcwd( ) - get current default path 1-168
gethostname( ) - get symbolic name of this machine 1-113
getpeername( ) - get name of connected peer 1-360
gets( ) - read string from standard in stream 1-379
getsockname( ) - get socket name 1-359
getsockopt( ) - get socket options 1-359

getwd( ) - get current default path 1-168
getw( ) - read next word (32-bit integer) from stream 1-380

help( ) - print synopsis of selected routines 1-481
h( ) - display (or set) shell history 1-510
hostAdd( ) - add host to host table 1-110
hostDelete( ) - delete host from host table. 1-111
hostGetByAddr( ) - look up host in host table by Internet address 1-112
hostGetByName( ) - look up host in host table by name. 1-111
hostLib - host table subroutine library 1-109
hostShow( ) - display host table 1-112
hostTblInit( ) - initialize network host table 1-110
htonl( ) - convert long (32 bit) from host to network byte order 1-392
htons( ) - convert short (16 bit) from host to network byte order. 1-393

iam( ) - set remote user name and password 1-268
icmpstatShow( ) - display statistics for ICMP 1-243
i( ) - print summary of each task's TCB 1-489
ifAddrGet( ) - get Internet address of network interface 1-116
ifAddrSet( ) - set interface address for network interface 1-116
ifBroadcastGet( ) - get broadcast address for network interface 1-118
ifBroadcastSet( ) - set broadcast address for network interface 1-117
ifDstAddrGet( ) - get Internet address of point-to-point peer 1-119
ifDstAddrSet( ) - define address for other end of point-to-point link 1-118
ifFlagChange( ) - change network interface flags 1-120
ifFlagGet( ) - get network interface flags 1-122
ifFlagSet( ) - specify flags for network interface 1-121
ifLib - network interface library 1-115
ifMaskGet( ) - get subnet mask for network interface 1-120
ifMaskSet( ) - define subnet for network interface 1-119
ifMetricGet( ) - get metric for network interface 1-123
ifMetricSet( ) - specify network interface hop count 1-122
ifRouteDelete( ) - delete routes associated with network interface 1-123
ifShow( ) - display attached network interfaces 1-242
if_bp - Vx960 & SunOS backplane driver 1-125
if_enp - CMC ENP 10/L Ethernet interface driver 1-130
if_ex - Excelan EXOS 201/202/302 Ethernet interface driver 1-131
if_ln - LANCE Ethernet interface driver 1-132
if_sl - serial line IP (SLIP) network interface driver 1-134
ifunit( ) - map interface name to interface structure pointer 1-124
inetLib - Internet address manipulation routines 1-139

inet_addr( ) - convert dot notation Internet address to long integer...... 1-140
inet_lnaof( ) - get local address (host number) from Internet address ... 1-141
inet_makeaddr_b( ) - form Internet address from network and host numbers 1-142
inet_makeaddr( ) - form Internet address from network and host numbers 1-142
inet_netof( ) - return network number from Internet address 1-143
inet_netof_string( ) - extract network address in dot notation.......... 1-144
inet_network( ) - convert Internet network number from string to address.......... 1-145
inet_ntoa_b( ) - convert network dot notation address to ASCII 1-145
inet_ntoa( ) - convert network dot notation address to ASCII 1-146
inetstatShow( ) - display all active connections for Internet protocol sockets 1-244
intALib - interrupt library assembly language routines.......... 1-147
intConnect( ) - connect C routine to hardware interrupt 1-149
intContext( ) - determine if we are in interrupt context or task context 1-152
intCount( ) - get current interrupt nesting depth.......... 1-153
intHandlerCreate( ) - construct interrupt handler for C routine.......... 1-150
intLevelSet( ) - set interrupt level.......... 1-147
intLib - interrupt subroutine library.......... 1-148
intLockLevelGet( ) - get current interrupt lock-out level.......... 1-152
intLockLevelSet( ) - set current interrupt lock-out level.......... 1-151
intLock( ) - lock out interrupts.......... 1-153
intRestrict( ) - restrict interrupt context from using routine 1-152
intUnlock( ) - cancel effect of intLock 1-154
intVecBaseGet( ) - get vector base address.......... 1-155
intVecBaseSet( ) - set vector base address 1-155
intVecGet( ) - get vector 1-156
intVecSet( ) - set CPU vector.......... 1-156
ioDefPathGet( ) - get current default path.......... 1-167
ioDefPathSet( ) - set current default path.......... 1-166
ioGlobalStdGet( ) - get fd for global standard in/out/error 1-169
ioGlobalStdSet( ) - set fd for global standard in/out/error.......... 1-169
ioLib - I/O interface library.......... 1-158
ioTaskStdGet( ) - get fd for task standard in/out/error 1-170
ioTaskStdSet( ) - set fd for task standard in/out/error.......... 1-170
ioctl( ) - perform file-specific control function.......... 1-165
iosDevAdd( ) - add device to I/O system 1-175
iosDevDelete( ) - delete device from I/O system.......... 1-175
iosDevFind( ) - find I/O device in device list.......... 1-176
iosDevShow( ) - display list of devices in system.......... 1-176
iosDrvInstall( ) - install I/O driver.......... 1-173

iosDrvRemove( ) - remove I/O driver 1-174
iosDrvShow( ) - display list of system drivers 1-174
iosFdShow( ) - display list of fd names in system 1-177
iosFdValue( ) - validate open fd and return driver-specific value 1-177
iosInit( ) - initialize I/O system 1-173
iosLib - I/O system 1-172
ipstatShow( ) - display IP statistics 1-244
isatty( ) - return whether underlying driver is tty device 1-371

jump - show jumpering of target and associated boards 3-4

kernelInit( ) - initialize kernel 1-179
kernelLib - Vx960 kernel library 1-178
kernelTimeSlice( ) - enable round-robin selection 1-181
kernelVersion( ) - return kernel revision string 1-180
kill( ) - send signal to task 1-349

ld( ) - load object module into memory 1-499
ledClose( ) - discard line-editor ID 1-185
ledControl( ) - change line-editor ID parameters 1-186
ledLib - line-editing library 1-182
ledOpen( ) - create new line-editor ID 1-184
ledRead( ) - read line with line-editing 1-185
l( ) - disassemble and display specified number of instructions 1-31
listen( ) - enable connections to socket 1-353
lkAddr( ) - list symbols whose values are near given value 1-505
lkup( ) - list global symbols 1-505
ll( ) - do long listing of directory contents 1-501
lnattach( ) - attach *ln* interface to network 1-132
loadLib - object module loader 1-187
loadModuleAt( ) - load object module into memory 1-188
loadModule( ) - load object module into memory 1-188
logFdAdd( ) - add logging fd 1-196
logFdDelete( ) - delete logging fd 1-197
logFdSet( ) - set primary logging fd 1-196
logInit( ) - initialize message logging library 1-194
logLib - message logging library 1-192
logMsg( ) - log formatted error message 1-194
logTask( ) - message-logging support task 1-197
loginDefaultEncrypt( ) - default password encryption routine 1-205

loginEncryptInstall( ) - install encryption routine 1-204
loginInit( ) - initialize login table 1-200
loginLib - user login/password subroutine library 1-198
loginPrompt( ) - display login prompt and validate user entry 1-203
loginStringSet( ) - change login string 1-204
loginUserAdd( ) - add user to login table 1-200
loginUserDelete( ) - delete user entry from login table 1-201
loginUserShow( ) - display user login table 1-202
loginUserVerify( ) - verify user name and password in login table 1-202
logout( ) - log out of Vx960 system 1-510
longjmp( ) - perform non-local goto 1-20
lsOld( ) - list contents of RT-11, ftp, or rsh directory 1-502
lseek( ) - set file read/write pointer 1-165
ls( ) - list contents of directory 1-500
lstAdd( ) - add node to end of list 1-209
lstConcat( ) - concatenate two lists 1-209
lstCount( ) - report number of nodes in list 1-210
lstDelete( ) - delete specified node from list 1-210
lstExtract( ) - extract sublist from list 1-211
lstFind( ) - find node in list 1-215
lstFirst( ) - find first node in list 1-211
lstFree( ) - free up list 1-216
lstGet( ) - delete and return first node from list 1-212
lstInit( ) - initialize list descriptor 1-208
lstInsert( ) - insert node in list after specified node 1-212
lstLast( ) - find last node in list 1-213
lstLib - doubly linked list subroutine library 1-207
lstNStep( ) - find list node *nStep* nodes away from specified node 1-215
lstNext( ) - find next node in list 1-213
lstNth( ) - find Nth node in list 1-214
lstPrevious( ) - find previous node in list 1-214

mRegs( ) - modify registers 1-508
makeStatTbl - make table of status values 3-5
makeSymTbl - make table of symbols 3-6
makeVersion - create Vx960 version module 3-7
malloc( ) - allocate block of memory from system memory partition 1-225
mb87030CtrlCreate( ) - create control structure for mb87030 SPC 2-1
mb87030CtrlInit( ) - initialize control structure for mb87030 SPC 2-3
mb87030Lib - Fujitsu mb87030 SCSI Protocol Controller (SPC) Library 2-1
mbufShow( ) - report mbuf statistics 1-245
memAddToPool( ) - add memory to system memory partition 1-224

memDevCreate( ) - create memory device 2-6
memDrv - pseudo memory device driver 2-5
memDrv( ) - install memory driver 2-6
memFindMax( ) - find largest free block in system memory partition 1-227
memLib - the Vx960 memory manager 1-217
memOptionsSet( ) - set debug options for system memory partition 1-224
memPartAddToPool( ) - add memory to memory partition 1-220
memPartAlloc( ) - allocate block of memory from specified partition 1-221
memPartCreate( ) - create memory partition 1-219
memPartFindMax( ) - find size of largest available free block 1-223
memPartFree( ) - free block of memory in specified partition 1-222
memPartOptionsSet( ) - set debug options for memory partition 1-221
memPartRealloc( ) - reallocate block of memory in specified partition 1-222
memPartShow( ) - show partition blocks and statistics 1-223
memShow( ) - show system memory partition blocks and statistics 1-228
m( ) - modify memory 1-493
mkdir( ) - make directory 1-502
msgQCreate( ) - create and initialize message queue 1-231
msgQDelete( ) - delete message queue 1-232
msgQInfoGet( ) - get information about message queue 1-236
msgQLib - message queue library 1-230
msgQNumMsgs( ) - get number of messages queued to message queue 1-235
msgQReceive( ) - receive message from message queue 1-234
msgQSend( ) - send message to message queue 1-233
msgQShow( ) - show information about message queue 1-238

netDevCreate( ) - create remote file device 2-10
netDrv - network remote file I/O driver 2-8
netDrv( ) - install network remote file driver 2-10
netHelp( ) - print synopsis of network routines 1-483
netLibInit( ) - initialize network package 1-240
netLib - network interface library 1-240
netShowInit( ) - initialize network show routines 1-245
netShow - network related information display routines 1-242
netTask( ) - network task entry point 1-241
nfsAuthUnixGet( ) - get NFS UNIX authentication parameters 1-252
nfsAuthUnixPrompt( ) - modify NFS UNIX authentication parameters 1-251
nfsAuthUnixSet( ) - set NFS UNIX authentication parameters 1-252
nfsAuthUnixShow( ) - display NFS UNIX authentication parameters 1-251
nfsDevShow( ) - display mounted NFS devices 2-16
nfsDrv - Network File System I/O driver 2-12
nfsDrv( ) - install NFS driver 2-14
nfsExportShow( ) - display exported file systems of remote host 1-250

nfsHelp( )	- display NFS help menu	1-249
nfsIdSet( )	- set ID number of NFS UNIX authentication parameters	1-253
nfsLib	- Network File System library	1-248
nfsMountAll( )	- mount all file systems exported by specified host	2-15
nfsMount( )	- mount NFS file system	2-15
nfsUnmount( )	- unmount NFS device	2-16
ntohl( )	- convert long (32 bit) from network to host byte order.	1-393
ntohs( )	- convert short (16 bit) from network to host byte order.	1-394
opendir( )	- open directory for searching	1-36
open( )	- open file	1-162
pause( )	- sleep until occurrence of signal	1-348
pcw( )	- return contents of PCW register	1-507
periodRun( )	- call function periodically	1-485
period( )	- spawn task to call function periodically	1-485
pfp( )	- return contents of register pfp (also sp, rip, r3-r15, g0-g14, fp)	1-506
picLib	- jump program support routines	3-8
pipeDevCreate( )	- create pipe device	2-21
pipeDrv	- pipe I/O driver	2-18
pipeDrv( )	- initialize pipe driver	2-20
printErr( )	- print formatted string to standard error	1-83
printErrno( )	- print definition of specified error status value	1-509
printLogo( )	- print Vx960 logo	1-510
printf( )	- print formatted string to standard out	1-81
ptyDevCreate( )	- create pseudo terminal	2-23
ptyDrv	- pseudo-terminal driver	2-22
ptyDrv( )	- initialize pseudo-terminal driver	2-23
putc( )	- append character to output stream	1-389
putchar( )	- append character to standard out stream	1-388
puts( )	- copy NULL-terminated string to output stream	1-380
putw( )	- append word (32-bit integer) to output stream	1-381
pwd( )	- print current default directory	1-496
ramDevCreate( )	- create RAM disk device	2-26
ramDrv	- RAM disk driver	2-25
ramDrv( )	- prepare RAM disk driver for use (optional)	2-26
rawFsDevInit( )	- associate block device with raw volume functions	1-258

rawFsInit( ) - prepare to use raw volume library 1-259

rawFsLib - raw block device file system library 1-254

rawFsModeChange( ) - modify mode of raw device volume 1-260

rawFsReadyChange( ) - notify **rawFsLib** of change in ready status 1-260

rawFsVolUnmount( ) - disable raw device volume 1-261

rcmd( ) - execute shell command on remote machine 1-265

readdir( ) - read one entry from directory 1-36

read( ) - read bytes from file or device 1-164

realloc( ) - reallocate block of memory 1-226

rebootHookAdd( ) - add routine to be called at reboot 1-263

rebootLib - reboot support library 1-262

reboot( ) - reset network devices and transfer control to boot ROMs 1-262

recv( ) - receive data from socket 1-357

recvfrom( ) - receive message from socket 1-356

recvmsg( ) - receive message from socket 1-358

remCurIdGet( ) - get current user name and password 1-267

remCurIdSet( ) - set remote user name and password 1-267

remLib - remote command library 1-265

remove( ) - delete file 1-160

rename( ) - change name of file 1-163

repeatRun( ) - call function repeatedly 1-486

repeat( ) - spawn task to call function repeatedly 1-487

rewinddir( ) - reset position to start of directory 1-37

rewind( ) - position stream at beginning 1-381

rlogInit( ) - initialize remote login facility 1-270

rlogLib - remote login library 1-270

rlogind( ) - the Vx960 remote login daemon 1-271

rlogin( ) - log in to remote host 1-272

rmdir( ) - remove directory 1-503

rm( ) - remove file 1-503

rngBufGet( ) - get characters from ring buffer 1-276

rngBufPut( ) - put bytes into ring buffer 1-276

rngCreate( ) - create empty ring buffer 1-274

rngDelete( ) - delete ring buffer 1-275

rngFlush( ) - make ring buffer empty 1-275

rngFreeBytes( ) - determine number of free bytes in ring buffer 1-278

rngIsEmpty( ) - test if ring buffer is empty 1-277

rngIsFull( ) - test if ring buffer is full (no more room) 1-277

rngLib - ring buffer subroutine library 1-273

rngMoveAhead( ) - advance ring pointer by *n* bytes 1-279

rngNBytes( ) - determine number of bytes in ring buffer 1-278

rngPutAhead( ) - put byte ahead in ring buffer without moving ring pointers 1-279

romStart( )	- generic ROM initialization	1-11
routeAdd( )	- add route	1-281
routeDelete( )	- delete route	1-282
routeLib	- network route manipulation library	1-281
routeShow( )	- display host and network routing tables	1-283
rpcInit( )	- initialize RPC package	1-286
rpcLib	- RPC support library	1-285
rpcTaskInit( )	- initialize task's access to RPC package	1-286
rresvport( )	- open socket with privileged port bound to it	1-266
rt11FsDateSet( )	- set current date	1-295
rt11FsDevInit( )	- initialize RT-11 device descriptor	1-292
rt11FsInit( )	- prepare to use RT-11 library	1-294
rt11FsLib	- RT-11 media-compatible file system library	1-287
rt11FsMkfs( )	- initialize device and create RT-11 file system	1-294
rt11FsModeChange( )	- modify mode of RT-11 volume	1-296
rt11FsReadyChange( )	- notify **rt11FsLib** of change in ready status	1-296
sbicShow( )	- display values of all readable wd33c93 chip registers	2-32
scanf( )	- read and convert characters from standard in stream	1-382
scsiAutoConfig( )	- configure all devices connected to SCSI controller	1-304
scsiBlkDevCreate( )	- define logical partition on SCSI block device	1-306
scsiBlkDevInit( )	- initialize fields in SCSI logical partition	1-306
scsiBusReset( )	- pulse reset signal on SCSI bus	1-307
scsiFormatUnit( )	- issue FORMAT_UNIT command to SCSI device	1-308
scsiInquiry( )	- issue INQUIRY command to SCSI device	1-308
scsiIoctl( )	- perform device-specific control function	1-312
scsiLib	- Small Computer System Interface (SCSI) library	1-298
scsiModeSelect( )	- issue MODE SELECT command to SCSI device	1-309
scsiModeSense( )	- issue MODE SENSE command to SCSI device	1-309
scsiPhysDevCreate( )	- create SCSI physical device structure	1-303
scsiPhysDevDelete( )	- delete SCSI physical device structure	1-303
scsiPhysDevIdGet( )	- return pointer to SCSI physical device structure	1-304
scsiRdSecs( )	- read sectors from SCSI block device	1-310
scsiReadCapacity( )	- issue READ_CAPACITY command to SCSI device	1-310
scsiReqSense( )	- issue REQUEST SENSE command to device and read results	1-311
scsiShow( )	- list physical devices attached to SCSI controller	1-305
scsiTestUnitRdy( )	- issue TEST_UNIT_READY command to SCSI device	1-307
scsiWrtSecs( )	- write sectors to SCSI block device	1-311
selNodeAdd( )	- add wake-up node to *select*( ) wake-up list	1-316
selNodeDelete( )	- find and delete node on wake-up list	1-317
selWakeupAll( )	- wake up all tasks on *select*( ) wake-up list	1-316
selWakeupListInit( )	- initialize *select*( ) wake-up list	1-317

***selWakeupListLen*( )**	- get number of nodes on *select*( ) wake-up list	1-318
***selWakeupType*( )**	- get type of given SEL_WAKEUP_NODE	1-318
***selWakeup*( )**	- wake up task pended in *select*( )	1-315
***selectInit*( )**	- initialize *select*( ) library	1-314
selectLib	- UNIX BSD 4.3 select library	1-313
***select*( )**	- pend on set of fds	1-314
***semBCreate*( )**	- create and initialize binary semaphore	1-322
semBLib	- binary semaphore library	1-320
***semCCreate*( )**	- create and initialize counting semaphore	1-325
semCLib	- counting semaphore library	1-324
***semClear*( )**	- take version 4.x semaphore if semaphore is available	1-337
***semCreate*( )**	- create and initialize version 4.x binary semaphore	1-336
***semDelete*( )**	- delete semaphore	1-329
***semFlush*( )**	- unblock every task pended on semaphore	1-329
***semGive*( )**	- give semaphore	1-328
***semInfo*( )**	- get list of task IDs that are blocked on semaphore	1-330
***semInit*( )**	- initialize static binary semaphore	1-337
semLib	- general semaphore library	1-326
***semMCreate*( )**	- create and initialize mutual-exclusion semaphore	1-334
***semMGiveForce*( )**	- give mutual-exclusion semaphore without restrictions	1-335
semMLib	- mutual-exclusion semaphore library	1-331
semOLib	- version 4.x binary semaphore library	1-336
***semTake*( )**	- take semaphore	1-328
***send*( )**	- send data to socket	1-355
***sendmsg*( )**	- send message to socket	1-356
***sendto*( )**	- send message to socket	1-355
***setbuf*( )**	- specify buffer to be used on stream	1-383
***setbuffer*( )**	- set buffer to be used on stream	1-383
***sethostname*( )**	- set symbolic name of this machine	1-113
***setjmp*( )**	- set non-local goto	1-19
***setlinebuf*( )**	- set line buffering for either *stdout* or *stderr*	1-384
***setsockopt*( )**	- set socket options	1-358
***s*( )**	- single-step	1-30
***shellHistory*( )**	- display (or set) shell history	1-340
***shellInit*( )**	- start shell	1-338
shellLib	- shell execution routines	1-338
***shellLock*( )**	- lock access to shell	1-342
***shellOrigStdSet*( )**	- set shell's default input/output/error fds	1-341
***shellPromptSet*( )**	- change shell prompt	1-341
***shellScriptAbort*( )**	- signal shell to stop processing script	1-340
***shell*( )**	- the shell entry point	1-339
***shutdown*( )**	- shut down network connection	1-361
***sigInit*( )**	- initialize signal facilities	1-345

sigLib - software signal facility library 1-343
sigRaise( ) - send signal to task 1-349
sigblock( ) - add to set of blocked signals 1-348
sigsetmask( ) - set signal mask 1-347
sigstack( ) - install separate signal stack 1-347
sigvec( ) - install signal handler 1-346
slattach( ) - attach SLIP interface to network 1-137
slipBaudSet( ) - set baud rate for SLIP interface 1-136
slipDelete( ) - delete SLIP interface 1-137
slipInit( ) - initialize SLIP interface 1-135
sockLib - UNIX BSD 4.3 compatible socket library 1-351
socket( ) - open socket 1-352
so( ) - single-step, but step over subroutine 1-31
spcShow( ) - display values of all readable mb87030 (SPC) registers 2-4
sp( ) - spawn task with default parameters 1-487
sprintf( ) - put formatted string in specified buffer 1-84
spyClkStart( ) - start collecting task activity data 1-363
spyClkStop( ) - stop collecting task activity data 1-364
spyHelp( ) - display task monitoring help menu 1-367
spyLib - spy CPU activity library 1-362
spyReport( ) - display task activity data 1-364
spyStop( ) - stop spying and reporting 1-365
spyTask( ) - run periodic task activity reports 1-365
spy( ) - begin periodic task activity reports 1-366
squeeze( ) - reclaim fragmented free space on RT-11 volume 1-499
sscanf( ) - obtain values for arguments from ASCII string 1-87
stat( ) - get file status information using pathname 1-39
stdioInit( ) - initialize **stdioLib** support 1-371
stdioLib - standard I/O library 1-368
strLib - string subroutine library 1-390
swab( ) - swap bytes 1-5
swapLib - byte swapping functions for little endian machines 1-392
symAdd( ) - create and add symbol to symbol table 1-398
symEach( ) - call routine to examine each entry in symbol table 1-402
symFindByNameAndType( ) - look up symbol by name and type 1-400
symFindByName( ) - look up symbol by name 1-399
symFindByValueAndType( ) - look up symbol by value and type 1-401
symFindByValue( ) - look up symbol by value 1-400
symLibInit( ) - initialize symbol table library 1-396
symLib - symbol table subroutine library 1-395
symRemove( ) - remove symbol from symbol table 1-399
symTblCreate( ) - create symbol table 1-397
symTblDelete( ) - delete symbol table 1-398

taskACWSet( ) - set task arithmetic control word 1-406
taskActivate( ) - activate task that has been initialized 1-422
taskArchLib - architecture specific task management routines for kernel 1-403
taskArgsGet( ) - get task's arguments 1-404
taskArgsSet( ) - set task's arguments 1-404
taskCreateHookAdd( ) - add routine to be called at every task create 1-411
taskCreateHookDelete( ) - delete previously added task create routine 1-412
taskCreateHookShow( ) - show list of task create routines 1-412
taskDelay( ) - delay task from executing 1-429
taskDeleteForce( ) - delete task without restriction 1-423
taskDeleteHookAdd( ) - add routine to be called at every task delete 1-414
taskDeleteHookDelete( ) - delete previously added task delete routine 1-415
taskDeleteHookShow( ) - show delete routines 1-415
taskDelete( ) - delete task 1-423
taskHookLib - task hook library 1-410
taskIdDefault( ) - set default task ID 1-434
taskIdListGet( ) - get list of active task IDs 1-436
taskIdSelf( ) - get task ID of running task 1-433
taskIdVerify( ) - verify existence of task 1-433
taskInfoGet( ) - get information about task 1-436
taskInit( ) - initialize task with stack at a specified address 1-420
taskIsReady( ) - check if task is ready to run 1-434
taskIsSuspended( ) - check if task is suspended 1-435
taskLib - task management library for Vx960 kernel 1-416
taskLock( ) - disable task rescheduling 1-426
taskNameToId( ) - look up task ID associated with task name 1-432
taskName( ) - get name associated with task ID 1-432
taskOptionsGet( ) - examine task options 1-430
taskOptionsSet( ) - change task options 1-429
taskPCWSet( ) - set task proccessor control word 1-405
taskPriorityGet( ) - examine priority of task 1-426
taskPrioritySet( ) - change priority of task 1-425
taskRegsGet( ) - get task's registers from TCB 1-431
taskRegsInit( ) - initialize task's registers 1-403
taskRegsSet( ) - set task's registers 1-431
taskRegsShow( ) - display contents of task's registers 1-405
taskRegsStackToTcb( ) - move i960 local registers from stack to TCB 1-408
taskRegsTcbToStack( ) - move i960 local registers from TCB to stack 1-409
taskRestart( ) - restart task 1-425
taskResume( ) - resume task 1-424
taskSafe( ) - make calling task safe from deletion 1-428
taskSpawn( ) - spawn task 1-419
taskStackAllot( ) - allot memory from caller's stack 1-407

taskStatusString( ) - get task's status as string ... 1-437
taskSuspend( ) - suspend task ... 1-424
taskSwitchHookAdd( ) - add routine to be called at every task switch ... 1-413
taskSwitchHookDelete( ) - delete previously added task switch routine ... 1-413
taskSwitchHookShow( ) - show switch routines ... 1-414
taskTCWSet( ) - set task trace control word ... 1-406
taskTcb( ) - get task control block for task ID ... 1-435
taskUnlock( ) - enable task rescheduling ... 1-427
taskUnsafe( ) - make calling task unsafe from deletion ... 1-428
taskVarAdd( ) - add task variable to task ... 1-440
taskVarDelete( ) - remove task variable from task ... 1-441
taskVarGet( ) - get value of task variable ... 1-442
taskVarInfo( ) - get list of task variables of task ... 1-443
taskVarInit( ) - initialize task variables facility ... 1-439
taskVarLib - task variables support library ... 1-438
taskVarSet( ) - set value of task variable ... 1-442
tcpstatShow( ) - display all statistics for TCP protocol ... 1-246
tcw( ) - return contents of TCW register ... 1-507
td( ) - delete task ... 1-491
telnetInit( ) - initialize telnet daemon ... 1-444
telnetLib - telnet library ... 1-444
telnetd( ) - Vx960 telnet daemon ... 1-445
tickAnnounce( ) - announce clock tick to kernel ... 1-446
tickGet( ) - get value of kernel's tick counter ... 1-448
tickLib - clock tick support library ... 1-446
tickSet( ) - set value of kernel's tick counter ... 1-447
ti( ) - print complete information from task's TCB ... 1-491
timexClear( ) - clear list of function calls to be timed ... 1-451
timexFunc( ) - specify functions to be timed ... 1-452
timexHelp( ) - display synopsis of execution timer facilities ... 1-452
timexInit( ) - include execution timer library ... 1-451
timexLib - execution timer facilities ... 1-449
timexN( ) - time repeated executions of function or group of functions ... 1-454
timexPost( ) - specify functions to be called after timing ... 1-455
timexPre( ) - specify functions to be called prior to timing ... 1-456
timexShow( ) - display list of function calls to be timed ... 1-456
timex( ) - time single execution of function or functions ... 1-453
trcArgCntSet( ) - change number of arguments printed by *trcStack*( ) ... 1-459
trcLib - stack trace library ... 1-458
trcStack( ) - print trace of function calls from stack ... 1-458
tr( ) - resume task ... 1-490
ts( ) - suspend task ... 1-490
tt( ) - print stack trace of task ... 1-32

tyAbortFuncSet( ) - set abort function 1-467

tyAbortSet( ) - change abort character 1-468

tyBackspaceSet( ) - change backspace character 1-468

tyDeleteLineSet( ) - change line-delete character 1-469

tyDevInit( ) - initialize tty device descriptor 1-466

tyEOFSet( ) - change end-of-file character 1-469

tyIRd( ) - interrupt level input 1-473

tyITx( ) - interrupt-level output 1-472

tyIoctl( ) - handle device control requests 1-471

tyLib - tty driver support library 1-461

tyMonitorTrapSet( ) - change trap-to-monitor character 1-470

tyRead( ) - do task-level read for tty device 1-472

tyWrite( ) - do task-level write for tty device 1-471

udpstatShow( ) - display statistics for UDP protocol 1-246

ungetc( ) - push character back into input stream 1-385

unlink( ) - delete file 1-161

usrClock( ) - user-defined system clock interrupt routine 1-476

usrConfig - user-defined system configuration module 1-474

usrInit( ) - user-defined system initialization routine 1-474

usrLib - user interface subroutine library 1-479

usrRoot( ) - the root task 1-475

usrScsiConfig( ) - configure SCSI peripheral (example) 1-476

version( ) - print Vx960 version information 1-492

vfdprintf( ) - print formatted string with variable argument list to specified fd 1-85

vprintf( ) - print formatted string with variable argument list to standard out 1-84

vsprintf( ) - put formatted string with variable argument list in specified buffer 1-85

vwbug - submit Technical Support Request (TSR) to Intel 3-9

vwman - print Vx960 manual entries 3-10

vxALib - miscellaneous assembly language routines 1-512

vxLib - miscellaneous support routines 1-513

vxMemProbe( ) - probe address for bus error 1-513

vxTas( ) - C-callable atomic test-and-set primitive 1-512

vxencrypt - encryption program for **loginLib** 3-11

wd33c93CtrlCreate( ) - create and partially initialize SBIC structure 2-29

wd33c93CtrlInit( ) - initialize user-specified fields in SBIC structure 2-31

wd33c93Lib - library for wd33c93 SCSI-Bus Interface Controller (SBIC) 2-29

***wdCancel*( )** - cancel currently counting watchdog 1-517

***wdCreate*( )** - create watchdog timer 1-516

***wdDelete*( )** - delete watchdog timer 1-516

wdLib - watchdog timer library 1-515

***wdStart*( )** - start watchdog timer 1-517

***whoami*( )** - display current remote identity 1-268

***write*( )** - write bytes to file 1-164

xsym - extract symbol table from object module 3-12

1

Libraries

CONTENTS

abs	- absolute value routine	1-1
bLib	- buffer manipulation library	1-2
bootConfig	- system configuration module for boot ROMs	1-10
bootInit	- ROM initialization module	1-11
bootLib	- boot ROM subroutine library	1-13
cALib	- C-language support routines	1-19
ctypeLib	- character classification and conversion macros	1-21
dbgLib	- debugging facilities	1-22
dirLib	- POSIX directory handling library	1-34
dosFsLib	- MS-DOS media-compatible file system library	1-40
dsm960Lib	- Vx960/80960 disassembler	1-61
errnoLib	- error status library	1-63
etherLib	- Ethernet raw I/O routines and hooks	1-68
exc960Lib	- architecture-dependent exception handling: i960	1-74
excLib	- exception handling facilities	1-76
fioLib	- formatted I/O library	1-80
floatLib	- floating-point formatting and scanning library	1-91
fppALib	- floating-point support assembly language routines	1-93
fppLib	- floating-point support library	1-94
ftpLib	- ARPA File Transfer Protocol library	1-98
ftpdLib	- ARPA File Transfer Protocol server	1-106
hostLib	- host table subroutine library	1-109
ifLib	- network interface library	1-115
if_bp	- Vx960 & SunOS backplane driver	1-125
if_enp	- CMC ENP 10/L Ethernet interface driver	1-130
if_ex	- Excelan EXOS 201/202/302 Ethernet interface driver	1-131
if_ln	- LANCE Ethernet interface driver	1-132
if_sl	- serial line IP (SLIP) network interface driver	1-134
inetLib	- Internet address manipulation routines	1-139
intALib	- interrupt library assembly language routines	1-147

intLib	- interrupt subroutine library	1-148
ioLib	- I/O interface library	1-158
iosLib	- I/O system	1-172
kernelLib	- Vx960 kernel library	1-178
ledLib	- line-editing library	1-182
loadLib	- object module loader	1-187
logLib	- message logging library	1-192
loginLib	- user login/password subroutine library	1-198
lstLib	- doubly linked list subroutine library	1-207
memLib	- the Vx960 memory manager	1-217
msgQLib	- message queue library	1-230
netLib	- network interface library	1-240
netShow	- network related information display routines	1-242
nfsLib	- Network File System library	1-248
rawFsLib	- raw block device file system library	1-254
rebootLib	- reboot support library	1-262
remLib	- remote command library	1-265
rlogLib	- remote login library	1-270
rngLib	- ring buffer subroutine library	1-273
routeLib	- network route manipulation library	1-281
rpcLib	- RPC support library	1-285
rt11FsLib	- RT-11 media-compatible file system library	1-287
scsiLib	- Small Computer System Interface (SCSI) library	1-298
selectLib	- UNIX BSD 4.3 select library	1-313
semBLib	- binary semaphore library	1-320
semCLib	- counting semaphore library	1-324
semLib	- general semaphore library	1-326
semMLib	- mutual-exclusion semaphore library	1-331
semOLib	- version 4.x binary semaphore library	1-336
shellLib	- shell execution routines	1-338
sigLib	- software signal facility library	1-343
sockLib	- UNIX BSD 4.3 compatible socket library	1-351
spyLib	- spy CPU activity library	1-362
stdioLib	- standard I/O library	1-368
strLib	- string subroutine library	1-390
swapLib	- byte swapping functions for little endian machines	1-392
symLib	- symbol table subroutine library	1-395
taskArchLib	- architecture specific task management routines for kernel	1-403
taskHookLib	- task hook library	1-410
taskLib	- task management library for the Vx960 kernel	1-416
taskVarLib	- task variables support library	1-438
telnetLib	- telnet library	1-444
tickLib	- clock tick support library	1-446
timexLib	- execution timer facilities	1-449

trcLib - stack trace library 1-458
tyLib - tty driver support library 1-461
usrConfig - user-defined system configuration module 1-474
usrLib - user interface subroutine library 1-479
vxALib - miscellaneous assembly language routines 1-512
vxLib - miscellaneous support routines 1-513
wdLib - watchdog timer library 1-515

abs

NAME

abs - absolute value routine

SYNOPSIS

abs( ) - absolute value of an integer

```
int abs (i)
```

abs( )

NAME

abs( ) - absolute value of an integer

SYNOPSIS

```
int abs (i)
    int  i;  /* integer for which to return absolute value */
```

DESCRIPTION

This routine returns the absolute value of the specified integer.

RETURNS

The value i if *i* is positive, or -i if *i* is negative.

SEE ALSO

abs

bLib

NAME

bLib - buffer manipulation library

SYNOPSIS

memccpy( ) - copy upto n bytes of second buffer to first or until c is copied

memchr( ) - return pointer to first occurance of char if in n bytes of buffer

memcmp( ) - compare first n bytes of two buffers numerically, byte for byte

memcpy( ) - copy n bytes of second buffer to the non-overlapping first

memmove( ) - copy n bytes of second buffer to the possibly overlapping first

memset( ) - set n bytes of buffer to the given value

bcopy( ) - copy n bytes from first buffer to second as quickly as possible

bcopyBytes( ) - copy n bytes from first buffer to second, a byte at a time

bcopyWords( ) - copy n shorts from first buffer to second, a short at at time

bcopyLongs( ) - copy n longs from first buffer to second, a long at a time

bfill( ) - fill buffer with n chars, may optimize to multi-byte stores

bfillBytes( ) - bfill one byte at a time

bzero( ) - fill buffer with n zero bytes, may optimize to multi-byte stores

bcmp( ) - same as ANSI memcmp

binvert( ) - reverse the order of bytes in the buffer

bswap( ) - swap the contents of two buffers

swab( ) - swap n bytes in from first buffer, place in second

```
void * memccpy (void *dest, const void *source, int c, int n)
void * memchr (const void *s, int c, size_t n)
int    memcmp (const void *s1, const void *s2, size_t n)
void * memcpy (void *dest, const void *source, size_t n)
void * memmove (void *dest, const void *source, size_t n)
void * memset (void *buf, int c, size_t n)
void   bcopy (char *source, char *dest, int n)
void   bcopyBytes (char *source, char *dest, int nbytes)
void   bcopyWords (char *source, char *dest, int nwords)
void   bcopyLongs (char *source, char *dest, int nlongs)
void   bfill (char *s, int nbytes, unsigned char v)
void   bfillBytes (char *s, int n, unsigned char v)
void   bzero (char *s, int n)
int    bcmp (char *s1, char *s2, int n)
void   binvert (char *s, int n)
void   bswap (char *source, char *dest, int n)
void   swab (char *source, char *dest, int n)
```

DESCRIPTION

This library contains routines that duplicate the ANSI-compliant versions of the UNIX buffer processing package and also supplies several tradition as well as Vx960 specific buffer manipulation routines. The functions operate on variable lenth arrays of bytes. They do not check for the overflow of any resulting strings, nor do they check for null termination as the routines of **strLib** do.

NOTE

The non-ANSI routine *memccpy*( ), and the ANSI routines *memchr*( ), *memcmp*( ), *memcpy*( ), *memmove*( ), and *memset*( ) actually come from libraries supplied with your compiler tool set, for example the Intel GNU/960 tool set. The documentation for these routines are not included as part of Vx960, but they are documented in the GNU/960 documentation set in the book "C, A Reference Manual," by Samuel P. Harbison and Guy L. Steele Jr.

INCLUDE FILE

#include "**strLib**.h"

SEE ALSO

strLib, "C, A Reference Manual," Harbison and Steele, Prentice Hall

bcmp( )

NAME

bcmp( ) - compare one buffer to another

SYNOPSIS

```
int bcmp (buf1, buf2, nbytes)
    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 = first *nbytes* of *buf1* and *buf2* are identical
-1 = *buf1* is less than *buf2*
1 = *buf1* is greater than *buf2*

SEE ALSO

bLib, GNU memcmp

binvert( )

NAME

binvert( ) - invert the order of bytes in a buffer

SYNOPSIS

```
VOID binvert (buf, nbytes)
    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, GNU libc doc or source

bswap( )

NAME

bswap( ) - swap buffers

SYNOPSIS

```
VOID bswap (buf1, buf2, nbytes)
    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

swab( )

NAME

swab( ) - swap bytes

SYNOPSIS

```
VOID swab (source, destination, nbytes)
    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. It is an error for *nbytes* to be odd.

RETURNS

N/A

SEE ALSO

bLib

bzero( )

NAME

bzero( ) - zero out a buffer

SYNOPSIS

```
VOID bzero (buffer, nbytes)
    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, memset

bcopy( )

NAME

bcopy( ) - copy one buffer to another

SYNOPSIS

```
VOID bcopy (source, destination, nbytes)
    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*( ), *memcpy*( )

bcopyBytes( )

NAME

bcopyBytes( ) - copy one buffer to another one byte at a time

SYNOPSIS

```
VOID bcopyBytes (source, destination, nbytes)
    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*( )

bcopyWords( )

NAME

bcopyWords( ) - copy one buffer to another one word at a time

SYNOPSIS

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

DESCRIPTION

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

RETURNS

N/A

SEE ALSO

bLib, *bcopy*( )

bcopyLongs( )

NAME

bcopyLongs( ) - copy one buffer to another one long word at a time

SYNOPSIS

```
VOID bcopyLongs (source, destination, nlongs)
    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*( )

bfill( )

NAME

bfill( ) - fill buffer with a specified character

SYNOPSIS

```
VOID bfill (buf, nbytes, ch)
    char           *buf;     /* pointer to buffer              */
    int            nbytes;   /* number of bytes to fill        */
    unsigned char  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. (For filling that is restricted to byte stores, see the manual entry for ***bfillBytes***( ).)

RETURNS

N/A

SEE ALSO

bLib, *bfillBytes*( ), memset

bfillBytes( )

NAME

bfillBytes( ) - fill buffer with a specified character one byte at a time

SYNOPSIS

```
VOID bfillBytes (buf, nbytes, ch)
    char          *buf;    /* pointer to buffer              */
    int           nbytes;  /* number of bytes to fill        */
    unsigned char 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*( )

bootConfig

NAME

bootConfig - system configuration module for boot ROMs

SYNOPSIS

NO CALLABLE ROUTINES

DESCRIPTION

This is the Intel-supplied configuration module for the Vx960 boot ROM. It is a stripped down version of **usrConfig.c**, having no Vx960 shell or debugging facilities. Its primary function is to load an object module over the network with either rsh or ftp protocols. Additionally, a simple set of single letter commands is provided for displaying and modifying memory contents. Use this module as a starting point for placing applications in ROM.

bootInit

NAME

bootInit - ROM initialization module

SYNOPSIS

romStart( ) - generic ROM initialization

```
VOID romStart (startType)
```

DESCRIPTION

This module provides a generic boot ROM facility. The target-specific romInit.s module does 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 Vx960 ROM image in RAM.

A modified version of the Public Domain **compress** program is used to uncompress the Vx960 boot ROM executable linked with it. Compressing object code typically achieves a 40% compression factor, permitting much larger systems to be burned into ROM. The only expense is the added few seconds delay while the first two stages complete.

SEE ALSO

compress( ), *romInit*( )

AUTHOR

The original compress program was written by: Spencer W. Thomas, Jim McKie, Steve Davies, Ken Turkowski, James A. Woods, Joe Orost.

romStart( )

NAME

romStart( ) - generic ROM initialization

SYNOPSIS

```
VOID romStart (startType)
    int  startType;
```

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

bootLib

NAME

bootLib - boot ROM subroutine library

SYNOPSIS

bootStringToStruct( ) - interpret the boot parameters from the boot line
bootStructToString( ) - construct a boot line
bootParamsShow( ) - display boot line parameters
bootParamsPrompt( ) - prompt for boot line parameters
bootNetmaskExtract( ) - extract netmask field from Internet address
bootBpAnchorExtract( ) - extract backplane address from device field

```
char *bootStringToStruct (bootString, pBootParams)
STATUS bootStructToString (paramString, pBootParams)
VOID bootParamsShow (paramString)
VOID bootParamsPrompt (string)
STATUS bootNetmaskExtract (string, pNetmask)
STATUS bootBpAnchorExtract (string, pAnchorAdrs)
```

DESCRIPTION

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

When Vx960 is first booted, certain parameters can be specified such as network addresses, boot device, host, and start-up file. This information is encoded into a single ASCII string known as the boot line. The boot line is placed at a known address (specified in **config.h**) by the boot ROMs so that the system being booted can discover the parameters that were used to boot the system. The boot line is the only means of communication from the boot ROMs to the booted system.

The boot line is of the form:

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

where:

bootdev - the boot device (e.g., "ei" for Intel 82596 LAN coprocessor, "bp" for backplane). For the backplane, this field can have an optional anchor address specification of the form "bp=*adrs*" (see *bootBpAnchorExtract*( )).

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

hostname - the name of the boot host.

filename - the file to be booted.

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

b - the Internet address of the backplane interface. This field can have an optional subnet mask as "e".

h - the Internet address of the boot host.

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

u - a valid user name on the boot host.

pw - the password for user on the host. This parameter is usually left blank. If specified, then 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.

INCLUDE FILE

bootLib.h

EXAMPLE

```
ei(0,0)host:/usr/vw/mz7122/config/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
```

SEE ALSO

bootConfig

bootStringToStruct( )

NAME

bootStringToStruct( ) - interpret the boot parameters from the boot line

SYNOPSIS

```
char *bootStringToStruct (bootString, pBootParams)
    char          *bootString;  /* boot line to be parsed */
    BOOT_PARAMS *pBootParams;   /* structure to be filled */
```

DESCRIPTION

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

See the manual entry for **bootLib** for a description of the format of the boot line.

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 (paramString, pBootParams)
    char         *paramString;  /* where to return the encoded boot line */
    BOOT_PARAMS  *pBootParams;
```

DESCRIPTION

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

See the manual entry for **bootLib** for a description of the format of the boot line.

RETURNS

OK

SEE ALSO

bootLib

bootParamsShow( )

NAME

bootParamsShow( ) - display boot line parameters

SYNOPSIS

```
VOID bootParamsShow (paramString)
    char  *paramString;  /* boot parameter string */
```

DESCRIPTION

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

SEE ALSO

bootLib

bootParamsPrompt( )

NAME

bootParamsPrompt( ) - prompt for boot line parameters

SYNOPSIS

```
VOID bootParamsPrompt (string)
    char  *string;  /* default boot line */
```

DESCRIPTION

This routine prompts the user for each of the boot parameters. For each parameter, the current value is displayed and a new value is prompted for. Typing a RETURN leaves the parameter unchanged. Typing a period (.) clears the parameter.

The parameter *string* is used for the initial values. A new boot line will be produced or changed as the user specified. That line will be copied over *string*. If there are no initial values, *string* should be empty on entry. It is the caller's responsibility to assure that *string* is long enough to hold the full boot line.

SEE ALSO

bootLib

bootNetmaskExtract( )

NAME

bootNetmaskExtract( ) - extract netmask field from Internet address

SYNOPSIS

```
STATUS bootNetmaskExtract (string, pNetmask)
    char  *string;    /* string containing adrs field */
    int   *pNetmask;  /* pointer where to return netmask */
```

DESCRIPTION

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

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

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

RETURNS

A 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

bootBpAnchorExtract( )

NAME

bootBpAnchorExtract( ) - extract backplane address from device field

SYNOPSIS

```
STATUS bootBpAnchorExtract (string, pAnchorAdrs)
    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 equals 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 equals 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

A 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

cALib

NAME

cALib - C-language support routines

SYNOPSIS

setjmp( ) - set non-local goto
longjmp( ) - perform non-local goto

```
int setjmp (env)
VOID longjmp (env, val)
```

DESCRIPTION

This library contains C-language support routines. It includes the "non-local goto" routines *setjmp*( ) and *longjmp*( ), and other routines and data necessary to satisfy the compilers or libraries of particular host systems.

INCLUDE FILE

setjmp.h

setjmp( )

NAME

setjmp( ) - set non-local goto

SYNOPSIS

```
int setjmp (env)
    jmp_buf  env;  /* where to save stack environment */
```

DESCRIPTION

This routine saves the current task context and program counter in *env* for later use by *longjmp*( ). It returns 0 when called. However, when program execution returns to the point at which *setjmp*( ) was called and the task context is restored by *longjmp*( ), *setjmp*( ) will then return the value *val*, as specified in *longjmp*( ).

RETURNS

0 or *val* if return is via *longjmp*( ).

SEE ALSO

cALib, *longjmp*( )

longjmp()

NAME

longjmp( ) - perform non-local goto

SYNOPSIS

```
VOID longjmp (env, val)
    jmp_buf  env;  /* where stack environment was saved *
    int      val;  /* value for setjmp to return    *
```

DESCRIPTION

This routine restores the previously saved *setjmp*( ) context and jumps to where *setjmp*( ) was called. The routine *setjmp*( ) will then return the value *val*.

NOTE

This routine does not return anything. Instead, it causes *setjmp*( ) to return *val*. If *val* is 0, then 1 is returned.

RETURNS

N/A

SEE ALSO

cALib, *setjmp*( )

ctypeLib

NAME

ctypeLib - character classification and conversion macros

DESCRIPTION

This UNIX-compatible library consists of macros that classify or convert ASCII-coded characters by table lookup. The macros are defined in **ctype.h**.

The following macros take a character parameter and return non-zero for TRUE or zero for FALSE.

isalpha(c) - c is a letter.

isupper(c) - c is an upper case letter.

islower(c) - c is a lower case letter.

isdigit(c) - c is a digit.

isxdigit(c) - c is a hexadecimal digit.

isspace(c) - c is a space, tab, carriage return, newline, or formfeed.

ispunct(c) - c is a punctuation character (neither control nor alphanumeric).

isalnum(c) - c is an alphanumeric character.

isprint(c) - c is a printable character, code 040 (space) through 0176 (tilde).

isgraph(c) - c is a visible graphics character, code 041 (exclamation mark) through 0176 (tilde).

iscntrl(c) - c is a delete character (0177) or ordinary control character (less than 040).

isascii(c) - c is an ASCII character, code less than 0200.

These macros perform simple conversions on single characters.

toupper(c) - converts c to its upper case equivalent.

tolower(c) - converts c to its lower case equivalent.

toascii(c) - forces c to be ASCII (strips highest eighth bit of character).

INCLUDE FILE

ctype.h

dbgLib

NAME

dbgLib - debugging facilities

SYNOPSIS

dbgHelp( ) - display debugging help menu
dbgInit( ) - initialize debug package
b( ) - set or display breakpoints
db( ) - set a data breakpoint
bd( ) - delete breakpoint
bdall( ) - delete all breakpoints
c( ) - continue from breakpoint
cret( ) - continue until return from current subroutine
s( ) - single-step
so( ) - single-step, but step over subroutine
l( ) - disassemble and display a specified number of instructions
tt( ) - print a stack trace of a task

```
VOID dbgHelp ()
STATUS dbgInit (breakpointTrapNum)
STATUS b (addr, taskNameOrId, count, quiet)
STATUS db (addr, access, taskNameOrId, count, quiet)
STATUS bd (addr, taskNameOrId)
STATUS bdall (taskNameOrId)
STATUS c (taskNameOrId, addr)
STATUS cret (taskNameOrId)
STATUS s (taskNameOrId, addr)
STATUS so (taskNameOrId)
VOID l (pInstr, count)
STATUS tt (taskNameOrId)
```

DESCRIPTION

This module provides Vx960' primary interactive debugging routines, which provide the following facilities:

- task breakpoints
- task single-stepping

In addition, **dbgLib** provides the facilities needed to provide enhanced use of other Vx960 modules, including:

- symbolic disassembly (via **dsmLib**),
- symbolic task stack tracing (via **trcLib**),
- 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 Vx960 development configuration as distributed, the debugging package is included in a Vx960 system by defining INCLUDE_DEBUG in **configAll.h**. This will enable the call to ***dbgInit*( )** in the task ***usrRoot*( )** in **usrConfig**. The *dbgInit*( ) routine initializes **dbgLib** and must be called before any other routines in the module are called.

BREAKPOINTS

Use the routine *b*( ) to set breakpoints. Breakpoints can be set to be hit by a specific task or by 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 status at this point (using ***i*( )** routine), you will see that it is suspended. The instruction at the breakpoint address has not yet been executed.

To continue executing the task, use the *c*( ) routine. At this point, the instruction that had contained the breakpoint is executed, and the task will then continue. The breakpoint remains in until it is explicitly removed.

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 resetting VX_UNBREAKABLE using *taskOptionsSet*( ). Several Vx960 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, using the low-level disassembly routines in **dsmLib**. The *tt*( ) routine provides a symbolic stack tracer, using the low-level stack trace routines in **trcLib**.

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 on a terminal for which the OPT_ABORT option has been set. By default, the abort key is ^C. See the manual entries for *tyAbortSet*( ) and *tyAbortFuncSet*( ).

DEFAULT TASK

Many routines in this package take a task ID as an argument. If this argument is missing or zero, the last task ID referenced is used. The routine *taskIdDefault*( ) is used to set and get the last referenced task ID, as do many Vx960 routines in other modules.

CAVEATS

Currently, some internals of vxWorks should not have breakpoints placed in them. Most notable are ***semGive*( )** and ***semTake*( )** which are used to manage crucial non-preemptable resources. One may work around this restriction by placing a breakpoint on the call to these routines.

ARCHITECTURE

The Vx960 native debugger has been restructured to be more portable across cpu architectures. Three files make up the debugger; **dbgLib.c**, dbg*arch***Lib.c**, and **dbgALib.s**. **dbgLib.c** contains the architecture independent functions, dbg*arch***Lib.c** contains the architecture dependent support functions, and **dbgALib.s** contains the assembly language stub for handling debugging exceptions.

SEE ALSO

excLib, dsmLib, tyLib, tyAbortSet, tyAbortFuncSet

dbgHelp( )

NAME

dbgHelp( ) - display debugging help menu

SYNOPSIS

```
VOID dbgHelp ()
```

DESCRIPTION

This routine displays a summary of **dbgLib** utilities with a short description of each:

```
dbgHelp                           Print this list
dbgInit                           Install debug facilities
b                                 Display breakpoints
b        addr[,task[,count]]      Set breakpoint
bd       addr[,task]              Delete breakpoint
bdall    [task]                   Delete all breakpoints
c        [task[,addr]]            Continue from breakpoint
cret     [task]                   Continue to subroutine return
```

```
db addr[access[,task[,count]]] Set data breakpoint
s          [task[,addr]]        Single step
so         [task]               Single step/step over subroutine
l          [adr[,nInst]]        List disassembled memory
tt         [task]               Do stack trace on task
```

SEE ALSO

dbgLib, "Debugging"

dbgInit( )

NAME

dbgInit( ) - initialize debug package

SYNOPSIS

```
STATUS dbgInit (breakpointTrapNum)
    int breakpointTrapNum; /* trap number for breakpoints */
```

DESCRIPTION

This routine installs the debug package. This involves:

- setting the interrupt vectors of the breakpoint and trace hardware interrupts to the appropriate debug interrupt handlers,
- adding the debug task switch routine to the kernel context switch call out,
- setting the terminal "abort" function to restart the shell,
- setting the exception handling extension to trace and restart the shell if the exception occurred in the shell task.

WHEN TO CALL

It is usually desirable to install the debugging facilities as early as possible in system initialization, if the debugging facilities are to be included at all. It should not, however, be called until the pipe driver, the ty driver used for logging, the exception handler library (**excLib** (1)), and message logging library (**logLib** (1)) have been initialized.

RETURNS

OK, or ERROR if unable to install task switch routine

SEE ALSO

dbgLib

b( )

NAME

b( ) - set or display breakpoints

SYNOPSIS

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

DESCRIPTION

This routine is used to set or examine breakpoints. To see the list of currently active breakpoints, call *b*( ) without arguments:

```
-> b
```

The list shows the address, task and passcount of each breakpoint. The routines *so*( ) and *cret*( ) insert temporary breakpoints and are so marked.

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

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

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

If quiet is specified, debugging information destined for the console will be supressed 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 setting or resetting VX_UNBREAKABLE with the routine *taskOptionsSet*( ).

RETURNS

OK, or ERROR if *addr* is odd or non-existent in memory, or if the breakpoint table is full.

SEE ALSO

dbgLib, bd, taskOptionsSet, *Programmer's Guide: Debugging*

db( )

NAME

db( ) - set a data breakpoint

SYNOPSIS

```
STATUS db (addr, access, taskNameOrId, count, quiet)
    INSTR  *addr;              /* where to set breakpoint, or        */
                               /* 0 = display all breakpoints        */
    UINT32       access;       /* access type                        */
                               /* 00 - store only                    */
                               /* 01 - data only (load or store)     */
                               /* 10 - data or instruction fetch     */
                               /* 11 - any access                    */
    int          taskNameOrId; /* task for which to set breakboint,  */
                               /* 0 = set all tasks                  */
    int          count;        /* number of passes before hit        */
    BOOL         quiet;        /* TRUE = don't print debugging info, */
                               /* FALSE = print debugging info       */
```

DESCRIPTION

This routine is used to set a data breakpoint (not to be confused with *bd*( )). If the architecture allows it, this function will add the breakpoint to the list of breakpoints, and set the hardware data breakpoint register(s). see *b*( ) above.

RETURNS

OK or ERROR.

SEE ALSO

dbgLib

bd( )

NAME

bd( ) - delete breakpoint

SYNOPSIS

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

DESCRIPTION

This routine is used to delete a specific breakpoint. To execute, type:

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

If *task* is missing 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 *so*( ) or *cret*( ) may be deleted.

RETURNS

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

SEE ALSO

Programmer's Guide: Debugging **dbgLib**, b

bdall( )

NAME

bdall( ) - delete all breakpoints

SYNOPSIS

```
STATUS bdall (taskNameOrId)
    int taskNameOrId; /* task for which to delete breakpoints, */
                      /* 0 = delete for all tasks              */
```

DESCRIPTION

This routine is used to remove all breakpoints. To execute, type:

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

If a 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, *Programmer's Guide: Debugging*

c( )

NAME

c( ) - continue from breakpoint

SYNOPSIS

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

DESCRIPTION

This routine is used to continue execution of a task that has stopped at a breakpoint. To execute, type:

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

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

RETURNS

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

SEE ALSO

dbgLib, tr, *Programmer's Guide: Debugging*

cret( )

NAME

cret( ) - continue until return from current subroutine

SYNOPSIS

```
STATUS cret (taskNameOrId)
    int taskNameOrId;    /* task to continue, 0 = default */
```

DESCRIPTION

This routine places a breakpoint at the return address of the current subroutine of a specified task, then continues execution of that task.

To execute, type:

```
    -> 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 displayed 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 if the breakpoint table is full.

SEE ALSO

dbgLib, so, *Programmer's Guide: Debugging*

s( )

NAME

s( ) - single-step

SYNOPSIS

```
STATUS s (taskNameOrId, addr)
    int taskNameOrId; /* task to step; 0 = use default          */
    int addr;         /* address to step to; 0 = next instruction */
```

DESCRIPTION

This routine is used to single-step a task that is stopped at a breakpoint. To execute, type:

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

If *task* is missing or zero, the last task referenced is assumed. If *addr* is non-zero then the program counter is changed to *addr* and the task is stepped.

SEE ALSO

dbgLib, *Programmer's Guide: Debugging*

so( )

NAME

so( ) - single-step, but step over subroutine

SYNOPSIS

```
STATUS so (taskNameOrId)
    int         taskNameOrId; /* task to step; 0 = use default */
```

DESCRIPTION

This routine is used to single-step a task that is stopped at a breakpoint; however, if the next instruction is a JSR or BSR, it breaks at the instruction following the subroutine call instead.

```
 -> so [task]
```

If task is missing or zero, the last task referenced is assumed.

SEE ALSO

dbgLib, *Programmer's Guide: Debugging*

l( )

NAME

l( ) - disassemble and display a specified number of instructions

SYNOPSIS

```
VOID l (pInstr, count)
    INSTR *pInstr; /* Address of first instruction to disassemble, */
                   /* if 0, continue from the last instruction     */
                   /* disassembled on the last call to l.         */
    int count;     /* Number of instruction to disassemble,        */
                   /* if 0, use the same as the last call to l.   */
```

DESCRIPTION

This routine dissasembles 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 printed as a label for that instruction. Also, addresses in the op-code field of instructions will be printed symbolically.

To execute, enter:

```
    -> l [addr [,count]]
```

If *addr* is omitted, disassembly continues from the previous address. If *count* is omitted, the last specified count is used (initially 10). As with all values entered via the shell, the address may be typed symbolically.

SEE ALSO

dbgLib, *Programmer's Guide: Debugging* **dsmLib**

tt( )

NAME

tt( ) - print a stack trace of a task

SYNOPSIS

```
STATUS tt (taskNameOrId)
    int taskNameOrId; /* task whose stack is to be traced */
```

DESCRIPTION

This routine prints a list of the nested routine calls that the specified task is in. Each routine call and its parameters is displayed.

If the task argument is missing or zero, the last task referenced is assumed. *tt*( ) 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.

This higher-level symbolic stack trace is built on top of the low-level routines provided by the routines in trcLib(1).

EXAMPLE

```
-> tt "logTask"
 3ab92 _vxTaskEntry   +10 : _logTask ()
  ee6e _logTask       +12 : _read ()
  d460 _read          +10 : _iosRead ()
  e234 _iosRead       +9c : _pipeRead ()
 23978 _pipeRead      +24 : _semTake (3f8b78, 0, 0, 0, 0, 0)
value = 0 = 0x0
```

This indicates that the *logTask*( ) is currently in *semTake*( ) (with one parameter) and was called by pipeRead, which was called by iosRead, and so on.

CAVEAT

In order to do the trace, some assumptions are made. In general, the trace will work for all C and assembly language routines, However, routines written in other languages, strange entries into routines, or tasks with corrupted stacks, can make the trace very confused. Because the i960 microprocessor passes parameters in global registers, and optimized C code will not necessarily move any of the parameters to the stack (or other knowable location), only the parameters for the last function call will be given. These paramters are the values in the first n global registers. Where n is the number given to the function trcArgCountSet. The defualt is 5. These parameters are only gauranteed if the task hit a breakpoint (or single-step) at the beginning of the function. This is because the register in which the parameter was passed may have been altered inside of the function. Also, all parameters are assumed to be 32-bit quantities, so structures passed as parameters will be displayed as some number of long integers.

RETURNS

OK, or ERROR if task does not exist.

SEE ALSO

dbgLib, *Programmer's Guide: Debugging* **trcLib**, UNIX cc manual entry

dirLib

NAME

dirLib - POSIX directory handling library

SYNOPSIS

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

```
DIR *opendir (dirName)
struct dirent *readdir (pDir)
VOID rewinddir (pDir)
STATUS closedir (pDir)
STATUS fstat (fd, pStat)
STATUS stat (name, pStat)
```

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

The basic directory operation determine the names of files and subdirectories in a directory. The *opendir*( ), *readdir*( ), *rewinddir*( ), and *closedir*( ) routines provide this ability.

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, the *readdir*( ) function 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*( ) function 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, and 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 regular file
S_ISDIR	- test if directory
S_ISCHR	- test if character special file
S_ISBLK	- test if block special file
S_ISFIFO	- test if fifo special file

Only the regular file and directory types are used for Vx960 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

opendir( )

NAME

opendir( ) - open a directory for searching

SYNOPSIS

```
DIR *opendir (dirName)
    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 using *opendir*( ), the *readdir*( ) routine is used to extract individual directory entries. Finally, *closedir*( ) is used to close the directory.

RETURNS

A pointer to a directory descriptor, or NULL if there is an error.

SEE ALSO

dirLib, *closedir*( ), *readdir*( ), *rewinddir*( ), *ls*( )

readdir( )

NAME

readdir( ) - read one entry from a directory

SYNOPSIS

```
struct dirent *readdir (pDir)
    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 at the end of the directory or if an error occurred. Because there are two conditions which might cause a NULL to be returned, you must use the task's error number (*errno*) to determine if there was an actual error. You should set *errno* to OK before calling readdir. 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 took place.

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( )*

rewinddir( )

NAME

rewinddir( ) - reset position to the start of a directory

SYNOPSIS

```
VOID rewinddir (pDir)
    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*( )

closedir( )

NAME

closedir( ) - close a directory

SYNOPSIS

```
STATUS closedir (pDir)
    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*( )

fstat( )

NAME

fstat( ) - get file status information

SYNOPSIS

```
STATUS fstat (fd, pStat)
    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 have been previously 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 have already been allocated before this routine is called.

Upon return, the fields in the stat structure are updated to reflect the characteristics of the file.

RETURNS

OK, or ERROR.

SEE ALSO

dirLib, *stat*( ), *ls*( )

stat( )

NAME

stat( ) - get file status information using a pathname

SYNOPSIS

```
STATUS stat (name, pStat)
    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.

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*( )

dosFsLib

NAME

dosFsLib - MS-DOS media-compatible file system library

SYNOPSIS

dosFsConfigInit( ) - initialize dosFs volume configuration structure
dosFsDateSet( ) - set the current date
dosFsDateTimeInstall( ) - install a user-supplied date/time function
dosFsDevInit( ) - associate a block device with dosFs file system functions
dosFsInit( ) - prepare to use the dosFs library
dosFsMkfs( ) - initialize a device and create a dosFs file system
dosFsModeChange( ) - modify mode of dosFs volume
dosFsReadyChange( ) - notify **dosFsLib** of a change in ready status
dosFsTimeSet( ) - set the current time
dosFsVolUnmount( ) - unmount a dosFs volume

```
STATUS dosFsConfigInit (pConfig, mediaByte, secPerClust, nResrvd, nFats, ...
STATUS dosFsDateSet (year, month, day)
VOID dosFsDateTimeInstall (pDateTimeFunc)
DOS_VOL_DESC *dosFsDevInit (devName, pBlkDev, pConfig)
STATUS dosFsInit (maxFiles)
DOS_VOL_DESC *dosFsMkfs (volName, pBlkDev)
VOID dosFsModeChange (vdptr, newMode)
VOID dosFsReadyChange (vdptr)
STATUS dosFsTimeSet (hour, minute, second)
STATUS dosFsVolUnmount (vdptr)
```

DESCRIPTION

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

USING THIS LIBRARY

The various functions provided by the Vx960 dosFs file system may be separated into three broad groups: general initialization, device initialization, and file system operation.

The *dosFsInit*( ) function is the principal initialization function; it need only be called once, regardless of how many dosFs devices will be used. In addition, the *dosFsDateTimeInstall*( ) function (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*( ) function is provided to easily initialize the data structure used during device initialization; however, its use is optional.

Lastly, several functions are provided to inform the file system of changes in the system environment. The *dosFsDateSet*( ) and *dosFsTimeSet*( ) functions 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*( ) function 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, the *dosFsVolUnmount*( ) function 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 functions is discussed in the following sections.

INITIALIZING DOSFSLIB

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

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

DEFINING A DOSFS DEVICE

To use this library for a particular device, the device descriptor structure used by the device driver must contain, as the very first item, a block device description structure (BLK_DEV). This must be initialized before calling *dosFsDevInit*( ). In the BLK_DEV structure, the driver includes the addresses of five routines which it must supply: one that reads one or more sectors, one that writes one or more sectors, one that performs I/O control on the device (using *ioctl*( )), one that checks the status of the device, and one that resets the device. These routines are described below. The BLK_DEV structure also contains fields which describe the physical configuration of the device. See *Programmer's Guide: I/O System* for more information on defining block devices.

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

After the *dosFsDevInit*( ) call has been made, 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.

An alternative to using the *dosFsDevInit*( ) routine is the *dosFsMkfs*( ) function. 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.

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 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, when one wants 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 sub-directories. Raw I/O does not use the disk sectors used for the boot sector, root directory, or FAT table. For raw disk I/O using the entire disk, see the manual entry for **rawLib**.

DEVICE AND PATH NAMES

On true MS-DOS machines, disk device names are typically of the form "A:", with a single letter designator followed by a colon. Such names may be used with the Vx960 dosFs file system. However, it is possible (and desirable) to use longer, more mnemonic device names, such as "DOS1:", or "/floppy0/". The name is specified during the *dosFsDevInit*( ) or *dosFsMkfs*( ) call.

The pathnames used to specify dosFs files and directories may use either forward slashes ("/") or backslashes ("\") as separators. These may be freely mixed. The choice of forward slashes or backslashes has absolutely no effect on the directory data written to the disk. (Note, however, that forward slashes are not allowed within Vx960 dosFs filenames, although they are normally legal for pure MS-DOS implementations.)

When using the Vx960 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.

READING DIRECTORY ENTRIES

Directories on Vx960 dosFs volumes 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 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 directory. This time is set when the file is created, and it is updated upon the close of a file 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 (pDateTime)
 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 Vx960 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 the *fstat*( ) function, 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", READ, 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 Vx960 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 the *ioctl*( ) call, specifying the FIOCONTIG function. The other parameter to the FIOCONTIG function is the size of the requested contiguous area in bytes. The FAT table 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 on disk to it:

```
fd = creat ("file", UPDATE, 0);          /* open file            */
status = ioctl (fd, FIOCONTIG, 0x10000); /* 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 ioctl (FIOCONTIG) operation should take place before any data is written to the file.

After the contiguous space has been allocated to a file, the file's size (kept in its directory entry) is unchanged. The size value is increased only as space is actually used by writing to the file.

Directories may also be allocated a contiguous disk area. A file descriptor to the directory is used to call ioctl (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.

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 table 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 File Allocation Table (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 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.

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 table 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 do other operations which could 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 the ***dosFsDevInit*( )** call is the address of a DOS_VOL_CONFIG structure, which specifies various configuration parameters. The boolean "dosvc_changeNoWarn" field must be set to TRUE if the driver and/or application is unable to issue an ***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 "dosvc_changeNoWarn" to TRUE also enables auto-sync mode (see below).

Note that all that is required for disk change notification is that either the ***dosFsVolUnmount*( )** call or the ***dosFsReadyChange*( )** call be issued each time the disk is changed. It is not necessary that it 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 an ***dosFsVolUnmount*( )** call before removing the disk, that would be sufficient, and "dosvc_changeNoWarn" should be defined as FALSE. It is important, however, that such a procedure be followed very 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 the ***dosFsVolUnmount*( )** call is made 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 specifying the 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 a boolean field ("dosvc_autoSync") in the DOS_VOL_CONFIG structure to TRUE when calling *dosFsDevInit*( ). When this option is enabled, modified directory and FAT table 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. "dosvc_changeNoWarn" is TRUE during *dosFsDevInit*( )). 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:

First, since the values in the volume descriptor will be 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 had been mounted would be fruitless.)

Second, the volume descriptor data is used when initializing a disk (with FIODISKINIT). The FIODISKINIT function will initialize a disk with the configuration of the most recently mounted disk, regardless of the original specification during *dosFsDevInit*( ). Therefore, it is recommended that you use FIODISKINIT 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 Vx960 DOS file system supports the following *ioctl*( ) functions. The functions listed are defined in the header **ioLib.h**. Unless stated otherwise, the fd used for these functions may be any fd 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:", WRITE);
status = ioctl (fd, FIODISKFORMAT);
```

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 fd should be obtained by opening the entire volume in raw mode:

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

FIODISKCHANGE

- Announces a media change. It performs the same function as *dosFsReadyChange*( ). This function may be called from interrupt level:

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

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);
```

FIOGETNAME- Gets the file name of the fd 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");
```

Only files (not directories) may be renamed.

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);
```

FIOFLUSH - Flushes the file output buffer. It guarantees that any output that has been requested is actually written to the device. If the specified fd 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);
```

FIOSYNC - Performs the same function as FIOFLUSH.

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 volume label to the string specified as *newLabel*. The string may consist of up to 11 ASCII characters:

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

FIOATTRIBSET

- Sets the file attribute byte in the DOS directory entry to the new value *newAttrib*. The fd 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);
```

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", READ); 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", READ); status = ioctl (fd,
FIOFSTATGET, &statStruct);
```

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

INCLUDE FILE

dosFsLib.h

SEE ALSO

ioLib, iosLib, dirLib, ramDrv, *Microsoft MS-DOS Programmer's Reference* (Microsoft Press), *Advanced MS-DOS Programming* (Ray Duncan, Microsoft Press), *Programmer's Guide: I/O System, Local File Systems*

dosFsConfigInit( )

NAME

dosFsConfigInit( ) - initialize dosFs volume configuration structure

SYNOPSIS

```
STATUS dosFsConfigInit (pConfig, mediaByte, secPerClust, nResrvd, nFats,
                        secPerFat, maxRootEnts, nHidden, changeNoWarn, autoSync)
    DOS_VOL_CONFIG  *pConfig;       /* pointer to volume config structure */
    char            mediaByte;      /* media descriptor byte */
    char            secPerClust;    /* sectors per cluster */
    short           nResrvd;        /* number of reserved sectors */
    char            nFats;          /* number of FAT copies */
    short           secPerFat;      /* number of sectors per FAT copy */
    short           maxRootEnts;    /* max number of entries in root dir */
    UINT            nHidden;        /* number of hidden sectors */
    BOOL            changeNoWarn;   /* TRUE = disk change is unannounced */
    BOOL            autoSync;       /* TRUE = sync disk during I/O */
```

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 Vx960 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

dosFsDateSet( )

NAME

dosFsDateSet( ) - set the current date

SYNOPSIS

```
STATUS dosFsDateSet (year, month, day)
    int  year;    /* year (1980...2099) */
    int  month;   /* month (1...12) */
    int  day;     /* day (1...31) */
```

DESCRIPTION

This routine sets the current date for the dosFs file system. All files created or modified will have this date in their directory entries.

SEE ALSO

dosFsLib, *dosFsTimeSet( )*, *dosFsDateTimeInstall( )*.

RETURNS

OK, or ERROR if the date is invalid.

dosFsDateTimeInstall( )

NAME

dosFsDateTimeInstall( ) - install a user-supplied date/time function

SYNOPSIS

```
VOID dosFsDateTimeInstall (pDateTimeFunc)
    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 (devName, pBlkDev, pConfig)
    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 Vx960 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 the *dosFsConfigInit*( ) routine.

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 changeNoWarn and autoSync modes are defined as FALSE.)

This routine allocates and initializes a volume descriptor (DOS_VOL_DESC) for the device. It returns a pointer to the DOS_VOL_DESC.

SEE ALSO

dosFsLib, *dosFsMkfs*( )

RETURNS

Pointer to volume descriptor (DOS_VOL_DESC), or NULL if there is an error.

dosFsInit( )

NAME

dosFsInit( ) - prepare to use the dosFs library

SYNOPSIS

```
STATUS dosFsInit (maxFiles)
    int maxFiles;  /* max # of simultaneously open files */
```

DESCRIPTION

This routine initializes the dosFs library. It must be called exactly once, before any other routine in the library. The argument specifies the number of dosFs files that may be open at once. This routine installs **dosFsLib** as a driver in the I/O system driver table, allocates and sets up the necessary memory structures, and initializes semaphores. The driver number assigned to **dosFsLib** is placed in the global variable, *dosFsDrvNum*.

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

RETURNS

OK, or ERROR.

SEE ALSO

dosFsLib

dosFsMkfs( )

NAME

dosFsMkfs( ) - initialize a device and create a dosFs file system

SYNOPSIS

```
DOS_VOL_DESC *dosFsMkfs (volName, pBlkDev)
    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 table copies
112	- root directory entries
0xF0	- media byte value
0	- hidden sectors
FALSE	- changeNoWarn (disk change without notification)
FALSE	- autoSync (auto-sync mode)

If initializing a large disk, it is quite possible that the entire disk area cannot be described by the maximum 64K clusters if only 2 sectors are contained in each cluster. In such a situation, the *dosFsMkfs*( ) routine 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 table copy is set to the minimum number of sectors which will contain sufficient FAT entries for the entire block device.

RETURNS

Pointer to dosFs volume descriptor, or NULL if there is an error.

SEE ALSO

dosFsLib

dosFsModeChange( )

NAME

dosFsModeChange( ) - modify mode of dosFs volume

SYNOPSIS

```
VOID dosFsModeChange (vdptr, newMode)
    DOS_VOL_DESC  *vdptr;    /* pointer to volume descriptor */
    int           newMode;   /* READ/WRITE/UPDATE (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 UPDATE (i.e., both READ and WRITE).

RETURNS

N/A

SEE ALSO

dosFsLib

dosFsReadyChange( )

NAME

dosFsReadyChange( ) - notify dosFsLib of a change in ready status

SYNOPSIS

```
VOID dosFsReadyChange (vdptr)
    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 a function code of 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 current time

SYNOPSIS

```
STATUS dosFsTimeSet (hour, minute, second)
    int  hour;     /* 0 to 23 */
    int  minute;   /* 0 to 59 */
    int  second;   /* 0 to 59 */
```

DESCRIPTION

This routine sets the the current time for the dosFs file system. All files created or modified will have this date in their directory entries.

RETURNS

OK, or ERROR if the time is invalid.

SEE ALSO

dosFsLib, *dosFsDateSet*( ), *dosFsDateTimeInstall*( )

dosFsVolUnmount( )

NAME

dosFsVolUnmount( ) - unmount a dosFs volume

SYNOPSIS

```
STATUS dosFsVolUnmount (vdptr)
    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*( )

dsm960Lib

NAME

dsm960Lib - Vx960/80960 disassembler

SYNOPSIS

dsm960Inst( ) - disassemble Intel i960 object code.

```
int dsm960Inst (pMem, adrs, prtAdrs)
```

DESCRIPTION

This library has all the routines and data structures necessary to print i960 object code in assembly language format. The programming interface is via dsm960Inst, which prints a single disassembled instruction. To disassemble from the shell, use the l command, which calls this library to do the actual work. See dbgLib for details.

ADDRESS PRINTING ROUTINE

Many of the operands to assembly language instructions are addresses. In order to allow the ability to print these symbolically, the call to dsm960Inst has, as a parameter, the address of a routine to call to print an address. When **dsmLib** needs to print an address as part of an operand field, it will call the supplied routine to do the actual printing. The routine should be declared as:

```
VOID prtAddress (address)
    int address;     * address to print *
```

When called, the routine should print the address on standard out (with printf or whatever) in whatever form, numeric or symbolic, that it desires. The routine used by l, for instance, looks up the address in the system symbol table and prints the symbol associated with it, if there is one. If not, it just prints the address as a hex number.

If the prtAddress argument to dsmPrint is NULL, a default print routine is used, which just prints the address as a hex number.

INCLUDE FILE

dsmLib.h

dsm960Inst( )

NAME

dsm960Inst( ) - disassemble i960 object code.

SYNOPSIS

```
int dsm960Inst (pMem, adrs, prtAdrs)
    UINT32      *pMem;      /* location to disassemble */
    UINT32      adrs;       /* preappend adrs for printing */
    FUNCPTR     prtAdrs;    /* adrs printing function  */
```

DESCRIPTION

This routine is called to disassemble and print (on standard out) a single instruction. The function passed as parameter prtAddress will be used to print any operands which might be construed as addresses. This could be either a simple routine that just prints a number, or one that looks up the address in a symbol table. The disassembled instruction will be prepended with the address passed as a parameter.

If prtAddress == 0, a default routine will be used that prints addresses as hex numbers.

RETURNS

The number of 32-bit words occupied by the instruction.

SEE ALSO

dsm960Lib

errnoLib

NAME

errnoLib - error status library

SYNOPSIS

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

```
int errnoGet ()
int errnoOfTaskGet (taskId)
STATUS errnoSet (errorValue)
STATUS errnoOfTaskSet (taskId, errorValue)
```

DESCRIPTION

This library contains routines for setting and examining the error status values of tasks and interrupts. Most Vx960 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 to the UNIX error status mechanism in which error status values are set in the global variable *errno*. However, in Vx960 there are many task and interrupt contexts that share common memory space and therefore conflict in their use of this global variable. Vx960 resolves this in two ways:

(1) For tasks, Vx960 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, Vx960 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 Vx960 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 Vx960 modules. These are defined in vwModNum.h. All other module numbers are available to applications.

PRINTING ERROR STATUS VALUES

Vx960 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 in vw/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 vw/h/myFile.h contains the line:

```
#define S_myFile_ERROR_TOO_MANY_COOKS 0x230003
```

The table *statSymTbl* is created by first running:

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

This creates a file statTbl.c, which, when compiled, generates *statSymTbl.* The table is then linked in with Vx960. These steps are normally done automatically by the vw/config/all makefile.

If the user now types from the Vx960 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 Vx960 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 Vx960.

INCLUDE FILES

The file vwModNum.h contains the module numbers for every Vx960 module. The include file for each module contains the error numbers which that module can generate.

SEE ALSO

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

errnoGet( )

NAME

errnoGet( ) - get the error status value of the calling task

SYNOPSIS

```
int errnoGet ()
```

DESCRIPTION

This routine gets the error status stored in *errno.* It is provided for compatibility with previous versions of Vx960 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 (taskId)
    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*( )

errnoSet( )

NAME

errnoSet( ) - set the error status value of the calling task

SYNOPSIS

```
STATUS errnoSet (errorValue)
    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 Vx960 and simply accesses *errno* directly.

RETURNS

OK, or ERROR if the interrupt nest level is too deep.

SEE ALSO

errnoLib, *errnoGet*( ), *errnoOfTaskSet*( )

errnoOfTaskSet( )

NAME

errnoOfTaskSet( ) - set the error status value of a specified task

SYNOPSIS

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

etherLib

NAME

etherLib - Ethernet raw I/O routines and hooks

SYNOPSIS

etherOutput( ) - send a packet on an Ethernet interface
etherInputHookAdd( ) - add a routine to receive all Ethernet input packets
etherOutputHookAdd( ) - add a routine to receive all Ethernet output packets
etherAddrResolve( ) - resolve an Ethernet address for a given Internet address

```
STATUS etherOutput (pIf, pEtherHeader, pData, dataLength)
STATUS etherInputHookAdd (inputHook)
STATUS etherOutputHookAdd (outputHook)
STATUS etherAddrResolve (pIf, targetAddr, eHdr, numTries, numTicks)
```

DESCRIPTION

This module 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

Most Vx960 network drivers, even if they are not truly Ethernet, are still accessible via these routines. These drivers include:

- Backplane
- Intel 82596 LAN coprocessor
- CMC ENP-10 Ethernet
- Excelan EXOS 202 and 302 Ethernet
- LANCE (AMD 7990) Ethernet
- National Semiconductor NIC Chip (DP8390) Ethernet

However, the following Vx960 network drivers are *not* accessible via these routines:

- Sun-3/E Ethernet
- Integrated Solutions Ethernet
- SLIP

SEE ALSO

Programmer's Guide: Network

etherOutput( )

NAME

etherOutput( ) - send a packet on an Ethernet interface

SYNOPSIS

```
STATUS etherOutput (pIf, pEtherHeader, pData, dataLength)
    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 given Internet Address.

RETURNS

OK, or ERROR if the routine runs out of mbufs.

SEE ALSO

etherLib, *etherAddrResolve*( )

etherInputHookAdd( )

NAME

etherInputHookAdd( ) - add a routine to receive all Ethernet input packets

SYNOPSIS

```
STATUS etherInputHookAdd (inputHook)
    FUNCPTR  inputHook;  /* routine to receive ethernet input */
```

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 (pIf, buffer, length)
    struct ifnet *pIf;    /* ptr to interface packet was received on *
    char         *buffer; /* ptr to received packet                  *
    int          length;  /* length of received packet               *
```

The hook routine should return TRUE if it has handled the input packet and no further action should be taken with it. It should return FALSE if it has not handled the input packet and normal processing (e.g., Internet) should take place.

The packet is in a temporary buffer when the hook routine is called. This buffer will be reused upon return from the hook. If the hook routine needs to retain the input packet, it should copy it elsewhere.

IMPLEMENTATION

A call to *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

etherOutputHookAdd( )

NAME

etherOutputHookAdd( ) - add a routine to receive all Ethernet output packets

SYNOPSIS

```
STATUS etherOutputHookAdd (outputHook)
    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 (pIf, buffer, length)
     struct ifnet *pIf;     /* ptr to interface packet will be sent on *
     char         *buffer;  /* ptr to packet to be transmitted         *
     int          length;   /* length of packet to be transmitted      *
```

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 *etherOutputHookRtn* should be invoked in the transmit routine of every network driver providing this service. For example:

```
...
#include "etherLib.h"
...
xxxStartOutput ()
/* call output hook if any *
if ((etherOutputHookRtn != NULL) &&
     (* etherOutputHookRtn) (&ls->ls_if, buf0, len))
    {
    /* output hook has already processed this packet *
    }
else
...
```

RETURNS

OK (always).

SEE ALSO

etherLib

etherAddrResolve( )

NAME

etherAddrResolve( ) - resolve an Ethernet address for a given Internet address

SYNOPSIS

```
STATUS etherAddrResolve (pIf, targetAddr, eHdr, numTries, numTicks)
    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 ARP'ing */
    int          numTicks;     /* number of ticks between ARP'ing */
```

DESCRIPTION

This routine uses the ARP (Address Resolution Protocol) 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 6 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*( )

exc960Lib

NAME

exc960Lib - architecture-dependent exception handling: i960

SYNOPSIS

excVecInit( ) - initialize exception/interrupt vectors
excInfoShow( ) - display exception information
excNMIHookAdd( ) - add a hook routine to be called in the event of an NMI.

```
STATUS excVecInit ()
VOID excInfoShow (pExcInfo, taskId)
FUNCPTR excNMIHookAdd (pHook)
```

DESCRIPTION

This module contains the architecture-dependent portions of the exception handling facilities. See excLib for the portions that are architecture independent.

excVecInit( )

NAME

excVecInit( ) - initialize exception/interrupt vectors

SYNOPSIS

```
STATUS excVecInit ()
```

DESCRIPTION

This routine sets all fault vectors to point to the appropriate default fault handlers. 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.

WHEN TO CALL

This routine is usually called from the system startup routine, *usrInit*( ) in usrConfig, before interrupts are enabled.

RETURNS

OK (always).

SEE ALSO

exc960Lib

excInfoShow( )

NAME

excInfoShow( ) - display exception information

SYNOPSIS

```
VOID excInfoShow (pExcInfo, taskId)
    EXC_INFO    *pExcInfo;
    UINT32      taskId;
```

DESCRIPTION

Display information associated with an interrupt or fault.

SEE ALSO

exc960Lib

excNMIHookAdd( )

NAME

excNMIHookAdd( ) - add a hook routine to be called in the event of an NMI.

SYNOPSIS

```
FUNCPTR excNMIHookAdd (pHook)
    FUNCPTR pHook;              /* pointer to hook routine */
```

DESCRIPTION

This routine adds a hook routine to be called in the event of a non-maskable interrupt. Note that only the 80960CA supports NMIs.

RETURNS

pointer to previous hook routine.

SEE ALSO

exc960Lib

excLib

NAME

excLib - exception handling facilities

SYNOPSIS

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

```
STATUS excInit ()
VOID excHookAdd (excepHook)
VOID excTask ()
```

DESCRIPTION

This module provides facilities for handling exceptions. It safely traps and reports exceptions caused by program errors in Vx960 tasks, and it reports occurrences of interrupts that are explicitly connected to other handlers.

INITIALIZATION

Initialization of excLib facilities is in two parts. First, the routine *excVecInit*( ) is called to set all the i960 fault vectors to the default handlers provided by this module. Since this does not involve Vx960' kernel facilities, it is usually done early in the system start-up routine *usrInit*( ) in the library usrConfig with interrupts disabled.

The rest of this package is initialized by calling *excInit*( ), which spawns the exception support task, *excTask*( ), and creates the pipe used to communicate with it. Since this initialization uses Vx960' kernel facilities and the pipe driver, it must be performed in the root task *usrRoot*( ) in the library usrConfig after the pipe driver has been installed.

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 Vx960 kernel and other system tasks continue uninterrupted. The suspended task can be examined with the usual Vx960 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 this module displays a description of the interrupt on standard output.

ADDITIONAL EXCEPTION HANDLING ROUTINE

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

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.

SEE ALSO

dbgLib, sigLib, intLib, *Programmer's Guide: Debugging*

excInit( )( )

NAME

excInit( ) - initialize the exception handling package

SYNOPSIS

```
STATUS excInit ()
```

DESCRIPTION

This routine installs the exception handling facilities and spawns *excTask*( ), which handles exception handling functions that need to be done at task level. It also creates the message queue used to communicate with *excTask*( ).

WHEN TO CALL

Install the exception handling facilities as early as possible in system initialization in the root task *usrRoot*( ) in usrConfig.

RETURNS

OK, or ERROR if a message queue cannot be created or *excTask*( ) cannot be spawned.

SEE ALSO

excLib, *excTask*( )

excHookAdd( )( )

NAME

excHookAdd( ) - specify a routine to be called with exceptions

SYNOPSIS

```
VOID excHookAdd (excepHook)
    FUNCPTR excepHook; /* routine to be called when exceptions occur */
```

DESCRIPTION

This routine specifies a routine that will be called when hardware exceptions occur. This routine is called after normal exception handling, which includes suspending the task that incurred the error and displaying information about the error.

The exception handling routine should be declared as:

```
VOID myHandler (task, vecNum, pEsf)
    int task;      /* id of offending task */
    int vecNum;    /* exception vector number */
    ESFO *pEsf;    /* pointer to exception stack frame */
```

where *task* is the ID of the task that was running when the exception occurred.

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.

SEE ALSO

excLib

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.

SEE ALSO

excLib, *excInit*( )

fioLib

NAME

fioLib - formatted I/O library

SYNOPSIS

printf( ) - print a formatted string to standard output
printErr( ) - print a formatted string to standard error
fdprintf( ) - print a formatted string to a specified file descriptor
sprintf( ) - put a formatted string in a specified buffer
vprintf( ) - print a formatted string with a variable argument list to standard output
vfdprintf( ) - print a formatted string with a variable argument list to a specified fd
vsprintf( ) - put a formatted string with a variable argument list in a specified buffer
fioFormatV( ) - convert a format string
sscanf( ) - obtain values for arguments from an ASCII string
fioRead( ) - read a buffer
fioRdString( ) - read a string from a file

```
int printf (fmt, ...)
int printErr (fmt, ...)
int fdprintf (fd, fmt, ...)
int sprintf (buffer, fmt, ...)
int vprintf (fmt, vaList)
int vfdprintf (fd, fmt, vaList)
int vsprintf (buffer, fmt, vaList)
int fioFormatV (fmt, vaList, outRoutine, outarg)
int sscanf (str, fmt, ...)
int fioRead (fd, buffer, maxbytes)
int fioRdString (fd, string, maxbytes)
```

DESCRIPTION

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

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

These routines do not use the buffered I/O facilities provided by stdioLib. Thus, they can be invoked even if the buffered stdio package has not been included, and even if the calling task was not created with the VX_STDIO task option. This includes *printf*( ), which in most UNIX systems is part of the buffered 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 stdioLib package. See the manual entry for stdioLib for more details.

SEE ALSO

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

printf( )

NAME

printf( ) - print a formatted string to standard output

SYNOPSIS

```
int printf (fmt, ...)
    char  *fmt;  /* format string for print       */
```

DESCRIPTION

This routine prints formatted strings to standard output. The string *fmt* contains ordinary characters, which are copied to standard output, and conversion specifications. The conversion specifications cause the arguments that follow *fmt* to be converted and printed as part of the formatted string. The number of arguments is arbitrary, but they must correspond to the conversion specifications in *fmt*.

This routine is compatible with the ANSI specification for *printf*( ) formats.

The format string contains normal ASCII text, which is passed as is, except for special conversion specifications of the following form:

```
%<-><+><sp><#><0><n><.n><[hlL]>[d|i|u|o|x|X|p|c|s|b|e|f|g|n]
```

- - Left-justify converted argument in its field.

+ - Precede a signed conversion with a sign.

sp - Precede a signed conversion with a sign, or with a blank if positive.

# - Print the result in an alternate form.

0 - Fill leading spaces with zeros.

n - Minimum field width. If the converted argument is shorter than the field width, it will be padded on the left (or right if left-adjustment is indicated). If the field width is specified as * instead of an integer, then the field width will be obtained from the next (integer) argument in sequence. Thus a field specification with an * for field width consumes an additional argument.

.n - Precision. For floating point formats, specifies the number of digits after the decimal point (f), or number of significant digits total (g). For strings (s), specifies the maximum number of characters to format. A precision can be specified as an * instead of an integer, in which case, the precision will be obtained from the next (integer) argument in sequence. Thus a field specification with an * for precision consumes an additional argument.

[hlL]
- Indicate that a corresponding data item is a *short* (h) or *long* (l or L) rather than an *int*.

d - Convert the argument to signed decimal notation.

i - Same as d.

u - Convert the argument to unsigned decimal notation.

o - Convert the argument to unsigned octal notation.

x - Convert the argument to unsigned hexadecimal notation.

X - Same as x but with upper case letters (A-F).

p - Convert the argument to system-dependent "pointer" representation (in Vx960, same as x with leading "0x").

c - Print a single-character argument.

s - Print a string argument; characters are printed until a null or maximum field width is reached.

b - Print a buffer argument; characters are printed until minimum field width is reached.

e - Print a *float* argument in the style "[-]d.ddde+dd", where there is one digit before the decimal point and the number after is equal to the precision specification for the argument. When the precision is missing, six digits are produced. If preceded by an l, the argument is a double.

f - Print a *float* argument in the style "[-]ddd.ddd", where the number of ds after the decimal point is equal to the precision specification for the argument. If the precision is missing, six digits are given; if the precision is explicitly 0, no digits and no decimal point are printed. If preceded by an l, the argument is a double.

g - Convert a *float* to either e or f, whichever is more compact.

n - Store number of characters printed so far in the integer pointed to by the next argument in sequence.

RETURNS

The number of characters output, or ERROR if there is an error in the output.

SEE ALSO

fioLib, ANSI C specification for *printf*( ) formatting

printErr( )

NAME

printErr( ) - print a formatted string to standard error

SYNOPSIS

```
int printErr (fmt, ...)
    char  *fmt;  /* format string for print */
```

DESCRIPTION

This routine is exactly like *printf*( ), except that it sends output to standard error.

SEE ALSO

fioLib, *printf*( )

fdprintf( )

NAME

fdprintf( ) - print a formatted string to a specified file descriptor

SYNOPSIS

```
int fdprintf (fd, fmt, ...)
    int   fd;    /* fd to which to print  */
    char  *fmt;  /* format string for print */
```

DESCRIPTION

This routine is exactly like *printf*( ), except that it sends its output to the specified *fd* rather than to standard output.

SEE ALSO

fioLib, *printf*( )

sprintf( )

NAME

sprintf( ) - put a formatted string in a specified buffer

SYNOPSIS

```
int sprintf (buffer, fmt, ...)
    char  *buffer;  /* buffer to receive decoded text */
    char  *fmt;     /* format string */
```

DESCRIPTION

This routine is exactly like *printf*( ), except that it copies its output to *buffer,* rather than to standard output. The buffer is null-terminated.

RETURNS

The number of characters copied to *buffer,* not including null termination.

NOTE

This routine has changed to be ANSI compatible. Previous Vx960 versions of *sprintf*( ) returned the number of characters put into *buffer* including the null termination. The ANSI C standard specifies that the null character should not be included in the returned count. Thus, the return value will be one less than in previous versions.

SEE ALSO

fioLib, *printf*( )

vprintf( )

NAME

vprintf( ) - print a formatted string with a variable argument list to standard output

SYNOPSIS

```
int vprintf (fmt, vaList)
    char     *fmt;    /* format string for print      */
    va_list  vaList;  /* optional arguments to format */
```

DESCRIPTION

This routine is exactly like *printf*( ), except that it takes the variable arguments to be formatted as a list of type *va_list* rather than as in-line arguments.

SEE ALSO

fioLib, *printf*( )

vfdprintf( )

NAME

vfdprintf( ) - print a formatted string with a variable argument list to a specified fd

SYNOPSIS

```
int vfdprintf (fd, fmt, vaList)
    int       fd;       /* fd to which to print  */
    char      *fmt;     /* format string for print */
    va_list   vaList;   /* optional arguments to format */
```

DESCRIPTION

This routine is exactly like *fdprintf*( ), except that it takes the variable arguments to be formatted as a list of type *va_list* rather than as in-line arguments.

SEE ALSO

fioLib, *fdprintf*( )

vsprintf( )

NAME

vsprintf( ) - put a formatted string with a variable argument list in a specified buffer

SYNOPSIS

```
int vsprintf (buffer, fmt, vaList)
    char      *buffer;  /* buffer to receive decoded text */
    char      *fmt;     /* format string */
    va_list   vaList;   /* optional arguments to format */
```

DESCRIPTION

This routine is exactly like *sprintf*( ), except that it takes the variable arguments to be formatted as a list of type *va_list* rather than as in-line arguments.

SEE ALSO

fioLib, *sprintf*( )

fioFormatV( )

NAME

fioFormatV( ) - convert a format string

SYNOPSIS

```
int fioFormatV (fmt, vaList, outRoutine, outarg)
    char     *fmt;        /* format string     */
    va_list  vaList;      /* pointer to list of varargs list              */
    FUNCPTR  outRoutine;  /* routine to handle 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 an fd, a buffer address, or any other value that can be passed in an "int".

The output routine should be declared as follows:

```
STATUS outRoutine (buffer, nchars, outarg)
    char *buffer;      /* buffer passed to routine      */
    int nchars;        /* length of buffer              */
    int outarg;        /* arbitrary argument passed to  */
                       /* format 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

sscanf( )

NAME

sscanf( ) - obtain values for arguments from an ASCII string

SYNOPSIS

```
int sscanf (str, fmt, ...)
    char *str;  /* string to scan  */
    char *fmt;  /* the format string  */
```

DESCRIPTION

This routine reads characters from the string *str,* interprets them according to format specifications in the string *fmt,* and assigns values to the arguments.

This routine is compatible with ANSI specification for *scanf*( ) formats.

The format string can contain white-space characters (space, tab, carriage return, newline, and form-feed), which are ignored, and ordinary characters (not %). Ordinary characters are expected to match the next non-white-space character in the string *str.*

Conversion specifications are of the following form:

```
[%<*><n><h|l>[d|i|u|o|x|X|c|s|e|f|g|E|F|G|[...]]
```

* - Suppress assignment. If present, will cause field to be scanned but no assignment to arguments.

n - An optional number specifying a maximum field width.

h|l - An optional character indicating that the value converted is to be assigned to a *short* (h) or *long* (l) integer or a *double* (l) precision float.

d - A decimal integer is expected in the input; the corresponding argument should be a pointer to an integer (*short* or *long*).

i - An integer is expected in the input. The input may be a decimal, hex, or octal character as determined by a leading prefix of "0" (octal) or "0x" (hex). The corresponding argument should be a pointer to an integer (*short* or *long*).

u - A decimal integer is expected in the input; the corresponding argument should be a pointer to an unsigned integer (*short* or *long*).

o - An octal integer is expected in the input; the corresponding argument should be a pointer to an integer (*short* or *long*).

x - A hexadecimal integer is expected in the input; the corresponding argument should be a pointer to an integer (*short* or *long*).

c - A character is expected in the input; the corresponding argument should be a character pointer. If a field width is given, the argument should be an array.

s - A character string is expected, delimited by white space, a field width specification, or an EOS; the corresponding argument should be a character pointer, pointing to a character array large enough to accept both the string and a terminating NULL, which will be added.

[...] - A character string is expected, delimited by any character not contained in the bracketed string. The left bracket is followed by a set of characters and a right bracket. The characters between the brackets define a set of characters making up the string. If the first character is not a circumflex (^), the input field is all characters until the first character not in the set between the brackets. If the first character after the left bracket is ^, the input field is all characters until the first character which is in the remaining set of characters between the brackets. Ranges of the form "a-z" are allowed in the bracketed set. The special characters ^ and] can be included in the set by making them the first characters in it. The corresponding argument should be a character pointer, pointing to a character array large enough to accept both the string and a terminating NULL, which will be added.

e|f|g|E|F|G

- A floating-point number is expected; the corresponding argument should be a pointer to a float, or double if there is a preceding l.

White-space characters in the string to be decoded are ignored, except insofar as they serve to delimit substrings to be decoded. Conversion proceeds until either it completes or something cannot be decoded according to the format specification. Arguments for which a value cannot be decoded are left unchanged.

RETURNS

The number of arguments to which values have been successfully assigned.

SEE ALSO

fioLib, ANSI specification for *scanf*( ) conversions

fioRead( )

NAME

fioRead( ) - read a buffer

SYNOPSIS

```
int fioRead (fd, buffer, maxbytes)
    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*.

NOTE

In previous versions of Vx960 (4.0.2 and earlier), *read*( ) did not always return as many bytes as the user specified, even if there were sufficient bytes remaining in a file. Thus *fioRead*( ) was often used to make sure that a complete buffer was read. As of 5.0, the definition of *read*( ) has been changed to be POSIX compliant, such that a complete buffer will be returned if there are sufficient bytes remaining in a file system fd. Some uses of *fioRead*( ) may therefore be unnecessary. However, *fioRead*( ) may still be useful for reading from non-file-system fds, such as serial channels or network sockets, where *read*( ) may still return with a partially filled buffer.

RETURNS

The number of bytes read, or ERROR if there is an error during the read operation.

SEE ALSO

fioLib, *read*( )

fioRdString( )

NAME

fioRdString( ) - read a string from a file

SYNOPSIS

```
int fioRdString (fd, string, maxbytes)
    int   fd;         /* file descriptor of device to read */
    char  string[];   /* buffer to receive input  */
```

```
        int    maxbytes;  /* maximum number of characters to read */
```

DESCRIPTION

This routine puts a line of input into *string*. The specified input fd 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.

SEE ALSO

fioLib

floatLib

NAME

floatLib - floating-point formatting and scanning library

SYNOPSIS

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

```
VOID floatInit ()
```

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 but rather get connected to call-outs in the *printf*( ) and *scanf*( ) family of functions in fioLib. This is done dynamically by the routine *floatInit*( ) (called in *usrRoot*( ) in usrConfig.c) if the INCLUDE_FLOATING_POINT option is defined in configAll.h. If this option is omitted (i.e., *floatInit*( ) is not called), floating-point format specifications in *printf*( ) and *sscanf*( ) will not be supported.

SEE ALSO

fioLib( )

INCLUDE FILE **floatLib.h**

floatInit( )

NAME

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

SYNOPSIS

```
VOID floatInit ()
```

DESCRIPTION

This routine must be called if floating-point format specifications are to be supported by the *printf*( ) and *scanf*( ) family of routines. If INCLUDE_FLOATING_POINT has been defined in configAll.h, it is called by the root task, *usrRoot*( ), in usrConfig.c.

RETURNS

N/A

SEE ALSO

floatLib

fppALib

NAME

fppALib - floating-point support assembly language routines

SYNOPSIS

fppSave( ) - save the floating-pointing context
fppRestore( ) - restore the floating-point context

```
VOID fppSave (pFpContext)
VOID fppRestore (pFpContext)
```

DESCRIPTION

This library contains routines to support the 80960SB and 80960KB floating-point. The routines *fppSave*( ) and *fppRestore*( ) save and restore all the task floating-point context information.

Higher-level access mechanisms are found in fppLib.

SEE ALSO

fppLib

fppLib

NAME

fppLib - floating-point support library

SYNOPSIS

fppTaskRegsShow( ) - display the floating-point registers of a task
fppTaskRegsGet( ) - get the floating-point registers from a task TCB
fppTaskRegsSet( ) - set the floating-point registers of a task
fppProbe( ) - probe for the presence of a floating-point support

```
VOID fppInit ()
VOID fppTaskRegsShow (task)
STATUS fppTaskRegsGet (task, fpregs, pFpcr, pFpsr, pFpiar)
STATUS fppTaskRegsSet (task, fpregs, fpcr, fpsr, fpiar)
STATUS fppProbe ()
```

DESCRIPTION

This library provides a low-level interface to the floating point for 80960KB and 80960SB. The routines *fppTaskRegsShow*( ), *fppTaskRegsSet*( ), and *fppTaskRegsGet*( ) inspect and set floating point registers on a per task basis. The routine *fppProbe*( ) checks for the presence of the 80960KB or 80960SB. With the exception of *fppProbe*( ), the higher level facilities in **dbgLib** and **usrLib** should be used instead of these routines.

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 **fppALib**. See the manual entry for *intConnect*( ) for more information.

SEE ALSO

fppALib, *intConnect*( )

fppTaskRegsShow( )

NAME

fppTaskRegsShow( ) - display the floating-point registers of a task

SYNOPSIS

```
VOID fppTaskRegsShow (task)
    int  task;  /* task to print registers for */
```

DESCRIPTION

This routine displays the contents of the floating-point registers for *task* in the following format:

```
    fp0 = ...
    fp1 = ...
    fp2 = ...
    fp3 = ...
```

If the floating-point registers are not supported, nothing is printed.

RETURNS

N/A

SEE ALSO

fppLib

fppTaskRegsGet( )

NAME

fppTaskRegsGet( ) - get the floating-point registers from a task TCB

SYNOPSIS

```
STATUS fppTaskRegsGet (task)
    int     task;      /* task to get info about        */
    fppTaskRegsGet(task, pRegs)
    REG_SET *pRegs
```

DESCRIPTION

This routine copies the floating-point registers of a task into the REG_SET structure. The floating-point registers are copied in an array containing four 80-bit 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 (i.e., leftover from the last task switch).

RETURNS

OK, or ERROR if there is no floating-point support

SEE ALSO

fppLib, *fppTaskRegsSet*( )

fppTaskRegsSet( )

NAME

fppTaskRegsSet( ) - set the floating-point registers of a task

SYNOPSIS

```
STATUS fppTaskRegsSet (task, pRegs)

    REG_SET *pRegs;
```

DESCRIPTION

This routine loads the specified values into the specified task TCB. The four 80-bit registers are copied.

RETURNS

OK, or ERROR if there is no floating-point support

SEE ALSO

fppLib, *fppTaskRegsGet*( )

fppProbe( )

NAME

fppProbe( ) - probe for the presence of floating-point support

SYNOPSIS

```
STATUS fppProbe ()
```

DESCRIPTION

This routine reports which library the current code has linked with.

IMPLEMENTATION

When the Vx960 kernel is built, it is linked with appropriate libraries for the target processor. This routine reports whether the library had the ability to use hardware floating point. Software floating point can always be used.

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.

RETURNS

OK if the floating-point support is present, otherwise ERROR.

SEE ALSO

fppLib

ftpLib

NAME

ftpLib - ARPA File Transfer Protocol library

SYNOPSIS

ftpCommand( ) - send an FTP command and get the reply
ftpXfer( ) - initiate a transfer via FTP
ftpReplyGet( ) - get an FTP command reply
ftpHookup( ) - get a control connection to the FTP server on a specified host
ftpLogin( ) - log in to a remote FTP server
ftpDataConnInit( ) - initialize an FTP data connection
ftpDataConnGet( ) - get a completed FTP data connection

```
int ftpCommand (ctrlSock, fmt, arg1, arg2, arg3, arg4, arg5, arg6)
STATUS ftpXfer (host, user, passwd, acct, cmd, dirname, filename, ...
int ftpReplyGet (ctrlSock, expecteof)
int ftpHookup (host)
STATUS ftpLogin (ctrlSock, user, passwd, account)
int ftpDataConnInit (ctrlSock)
int ftpDataConnGet (dataSock)
```

DESCRIPTION

This library provides facilities for transferring files to and from a host via File Transfer Protocol (FTP). This library implements only the "client" side of the FTP facilities.

FTP IN VXWORKS

Vx960 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, familiarity with ftpLib is not necessary.

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") != FTP_COMPLETE)     ||
        (ftpCommand (ctrlSock, "CWD %s", dirname) != FTP_COMPLETE) ||
        ((dataSock = ftpDataConnInit (ctrlSock)) == ERROR)     ||
        (ftpCommand (ctrlSock, "RETR %s", filename) != 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 FILE

ftpLib.h

SEE ALSO

netDrv

ftpCommand( )

NAME

ftpCommand( ) - send an FTP command and get the reply

SYNOPSIS

```
int ftpCommand (ctrlSock, fmt, arg1, arg2, arg3, arg4, arg5, arg6)
    int   ctrlSock;  /* fd of control connection socket */
    char  *fmt;      /* format string of command to send */
    int   arg1;      /* arguments 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 6 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");   /* image-type xfer *
ftpCommand (ctrlSock, "STOR %s", filename); /* 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 read/write error or unexpected EOF.

SEE ALSO

ftpLib, *ftpReplyGet*( )

ftpXfer( )

NAME

ftpXfer( ) - initiate a transfer via FTP

SYNOPSIS

```
STATUS ftpXfer (host, user, passwd, acct, cmd, dirname, filename,
                 pCtrlSock, pDataSock)
    char  *host;       /* name of server host */
    char  *user;       /* user name with which to login to host */
    char  *passwd;     /* password with which to login to host */
    char  *acct;       /* account with which to login to host */
    char  *cmd;        /* command to send to host */
    char  *dirname;    /* directory to 'cd' to before sending command */
    char  *filename;   /* filename to send with command */
    int   *pCtrlSock;  /* where to return control socket fd */
    int   *pDataSock;  /* where to return data socket fd,
                        * (NULL == don't open data connection) */
```

DESCRIPTION

This routine initiates a transfer via a remote FTP server in the following order:

(1) Establishes a connection to the FTP server on the specified host.

(2) Logs in with the specified user name, password, and account, as necessary for the particular host.

(3) Sets the transfer type to image by sending the command "TYPE I".

(4) Changes to the specified directory by sending the command "CWD *dirname*".

(5) Sends the specified transfer command with the specified filename as an argument, and establishs 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 fds 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 fd returned in *pDataSock*. When all incoming data has been read (as indicated by an EOF when reading the data socket) and/or all outgoing data has been written, the data socket fd should be closed. The routine *ftpReplyGet*( ) should then be called to receive the final reply on the control socket, after which the control socket should be closed.

If the FTP command does not involve data transfer (i.e., file delete or rename), *pDataSock* should be NULL, in which case no data connection will be established.

EXAMPLE

The following code fragment reads the file "/usr/fred/myfile" from the host "server", logged in as user "fred", with password "magic" and no account name.

```
int ctrlSock;
int dataSock;
char buf [512];
int nBytes;

if (ftpXfer ("server", "fred", "magic", "",
             "RETR %s", "/usr/fred", "myfile",
             &ctrlSock, &dataSock) == ERROR)
    return (ERROR);

while ((nBytes = read (dataSock, buf, sizeof (buf))) > 0)
```

```
        {
        ...
        }
    close (dataSock);

    if (nBytes < 0)              /* read error? *
        status = ERROR;

    if (ftpReplyGet (ctrlSock) != FTP_COMPLETE)
        status = ERROR;

    if (ftpCommand (ctrlSock, "QUIT") != 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*( )

ftpReplyGet( )

NAME

ftpReplyGet( ) - get an FTP command reply

SYNOPSIS

```
int ftpReplyGet (ctrlSock, expecteof)
    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

ftpHookup( )

NAME

ftpHookup( ) - get a control connection to the FTP server on a specified host

SYNOPSIS

```
int ftpHookup (host)
    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 fd 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 (ctrlSock, user, passwd, account)
    int   ctrlSock;  /* fd of control socket on which to login */
    char  *user;     /* user name with which to login to host */
    char  *passwd;   /* password with which to login to host */
    char  *account;  /* account with which to login to host */
```

DESCRIPTION

This routine logs into 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*( )

ftpDataConnInit( )

NAME

ftpDataConnInit( ) - initialize an FTP data connection

SYNOPSIS

```
int ftpDataConnInit (ctrlSock)
    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 fd of the data socket created, or ERROR.

SEE ALSO

ftpLib, *ftpHookup*( ), *ftpLogin*( ), *ftpCommand*( ), *ftpXfer*( )

ftpDataConnGet( )

NAME

ftpDataConnGet( ) - get a completed FTP data connection

SYNOPSIS

```
int ftpDataConnGet (dataSock)
    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 fd 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 fd of the new data socket, or ERROR if the connection failed.

SEE ALSO

ftpLib, *ftpDataConnInit*( ), *ftpCommand*( )

ftpdLib

NAME

ftpdLib - ARPA File Transfer Protocol server

SYNOPSIS

ftpdTask( ) - FTP server daemon task
ftpdInit( ) - initialize the FTP server task
ftpdDelete( ) - clean up and finalize the FTP server task

```
STATUS ftpdTask ()
STATUS ftpdInit (stackSize)
VOID ftpdDelete ()
```

DESCRIPTION

This library provides File Transfer Protocol (FTP) service to allow an FTP client to store and retrieve files to and from the Vx960 target. The FTP is defined in Requests For Comments (RFC) document 959 and this library implements an extented subset of this specification. This implementation of the FTP server understands the following FTP requests:

USER	- Verify user name
PASS	- Verify password for the user
QUIT	- Quit the session
LIST	- List out contents of a directory
NLST	- List out contents of a directory using 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

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

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

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

SEE ALSO

ftpLib, netDrv, RFC-959 File Transfer Protocol

ftpdTask( )

NAME

ftpdTask( ) - FTP server daemon task

SYNOPSIS

```
STATUS ftpdTask ()
```

DESCRIPTION

This routine processes incoming FTP client requests by spawning a new FTP work task for each connection that is set up.

RETURNS

OK, or ERROR when the task terminates with errors due to socket related I/O problems, lack of resources, or task creation failure.

SEE ALSO

ftpdLib

ftpdInit( )

NAME

ftpdInit( ) - initialize the FTP server task

SYNOPSIS

```
STATUS ftpdInit (stackSize)
    int stackSize; /* stack size for the ftpdTask */
```

DESCRIPTION

This routine will spawn a new FTP server task if one does not already exist. If an existing FTP server task is running already, *ftpdInit*( ) will simply return without creating a new task. It will simply report whether a new FTP task was successfully spawned. An argument *stackSize* can be specified to change the default stack size for the FTP server task. The default size is set in the global variable *ftpdWorkTaskStackSize.*

RETURNS

OK if a new FTP task is created successfully, ERROR otherwise.

SEE ALSO

ftpdLib

ftpdDelete( )

NAME

ftpdDelete( ) - clean up and finalize the FTP server task

SYNOPSIS

```
VOID ftpdDelete ()
```

DESCRIPTION

This routine finalizes and deletes the main FTP server task *ftpdTask*( ), cleans up all active sessions spawned by it, and reclaims all resources used by each active slot in the active session list. All sockets associated with FTP services will be closed, and all memory dynamically allocated will be freed.

RETURNS

N/A

SEE ALSO

ftpdLib

hostLib

NAME

hostLib - host table subroutine library

SYNOPSIS

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

```
VOID hostTblInit ()
STATUS hostAdd (hostName, hostAddr)
STATUS hostDelete (name, addr)
int hostGetByName (name)
STATUS hostGetByAddr (addr, name)
VOID hostShow ()
int sethostname (name, nameLen)
int gethostname (name, nameLen)
```

DESCRIPTION

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

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

SEE ALSO

inetLib, *Programmer's Guide: Network*

hostTblInit( )

NAME

hostTblInit( ) - initialize the network host table

SYNOPSIS

```
VOID hostTblInit ()
```

DESCRIPTION

This routine initializes the host list data structure used by routines throughout this module. It should be called by *usrRoot*( ) in usrConfig.c, before any other routines in this module.

SEE ALSO

hostLib, usrConfig

hostAdd( )

NAME

hostAdd( ) - add a host to the host table

SYNOPSIS

```
STATUS hostAdd (hostName, hostAddr)
    char  *hostName;  /* host name */
    char  *hostAddr;  /* host address 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 "vxhost", "90.2"
-> hostShow
hostname          inet address        aliases
--------          ------------        -------
localhost         127.0.0.1
yuba              90.0.0.3
vxhost            90.0.0.2
value = 12288 = 0x3000 = _bzero + 0x18
```

RETURNS

OK, or ERROR if the host table is full, the host name is already entered, there is an invalid Internet address, or the routine runs out of memory.

SEE ALSO

hostLib, netDrv, nfsDrv

hostDelete( )

NAME

hostDelete( ) - delete a host from the host table.

SYNOPSIS

```
STATUS hostDelete (name, addr)
    char  *name;  /* host name or alias */
    char  *addr;  /* host address in standard internet format */
```

DESCRIPTION

This routine deletes a host name from the local host table. The host entry is deleted if host name is used as *name*. The alias of the host name is deleted if the alias is used as *name*.

RETURNS

OK, or ERROR if the host is unknown.

SEE ALSO

hostLib

hostGetByName( )

NAME

hostGetByName( ) - look up a host in the host table by its name.

SYNOPSIS

```
int hostGetByName (name)
    char  *name;  /* name of host */
```

DESCRIPTION

This routine returns the Internet address of a host that has been added to the host table by *hostAdd*( ).

RETURNS

The Internet address (as an integer), or ERROR if the host is unknown.

SEE ALSO

hostLib

hostGetByAddr( )

NAME

hostGetByAddr( ) - look up a host in the host table by its Internet address

SYNOPSIS

```
STATUS hostGetByAddr (addr, name)
    int   addr;  /* inet address of host */
    char  *name;  /* buffer to hold the name */
```

DESCRIPTION

This routine finds the host name by its Internet address and puts it in *name* which should be preallocated with (MAXHOSTNAMELEN + 1) bytes of memory. The buffer *name* is null-terminated unless insufficient space is provided.

RETURNS

OK, or ERROR if the host is unknown.

WARNING

The routine *hostGetByAddr( )* does not look for aliases. Host names are limited to MAXHOSTNAMELEN (from hostLib.h) characters, currently 64.

SEE ALSO

hostLib, *hostGetByName( )*

hostShow( )

NAME

hostShow( ) - display the host table

SYNOPSIS

```
VOID hostShow ()
```

DESCRIPTION

This routine prints a list of remote hosts along with their Internet addresses and aliases.

RETURNS

N/A

SEE ALSO

hostLib, *hostAdd*( )

sethostname( )

NAME

sethostname( ) - set the symbolic name of this machine

SYNOPSIS

```
int sethostname (name, nameLen)
    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.

SEE ALSO

hostLib

gethostname( )

NAME

gethostname( ) - get the symbolic name of this machine

SYNOPSIS

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

DESCRIPTION

This routine gets the target machine's symbolic name which can be used for identification.

SEE ALSO

hostLib

ifLib

NAME

ifLib - network interface library

SYNOPSIS

ifAddrSet( ) - set an interface address for a network interface

ifAddrGet( ) - get the Internet address of a network interface

ifBroadcastSet( ) - set the broadcast address for a network interface

ifBroadcastGet( ) - get the broadcast address for a network interface

ifDstAddrSet( ) - define an address for the other end of a point-to-point link

ifDstAddrGet( ) - get the Internet address of a point-to-point peer

ifMaskSet( ) - define a subnet for a network interface

ifMaskGet( ) - get the subnet mask for a network interface

ifFlagChange( ) - change the network interface flags

ifFlagSet( ) - specify the flags for a network interface

ifFlagGet( ) - get the network interface flags

ifMetricSet( ) - specify a network interface hop count

ifMetricGet( ) - get the metric for a network interface

ifRouteDelete( ) - delete routes associated with a network interface

ifunit( ) - map an interface name to an interface structure pointer

```
STATUS ifAddrSet (interfaceName, interfaceAddress)
STATUS ifAddrGet (interfaceName, interfaceAddress)
STATUS ifBroadcastSet (interfaceName, broadcastAddress)
STATUS ifBroadcastGet (interfaceName, broadcastAddress)
STATUS ifDstAddrSet (interfaceName, dstAddress)
STATUS ifDstAddrGet (interfaceName, dstAddress)
STATUS ifMaskSet (interfaceName, netMask)
STATUS ifMaskGet (interfaceName, netMask)
STATUS ifFlagChange (interfaceName, flags, on)
STATUS ifFlagSet (interfaceName, flags)
STATUS ifFlagGet (interfaceName, flags)
STATUS ifMetricSet (interfaceName, metric)
STATUS ifMetricGet (interfaceName, pMetric)
int ifRouteDelete (ifName, unit)
struct ifnet *ifunit(name)
```

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.

SEE ALSO

hostLib, *Programmer's Guide: Network* UNIX man ifconfig(8), Stevens section 6.11

ifAddrSet( )

NAME

ifAddrSet( ) - set an interface address for a network interface

SYNOPSIS

```
STATUS ifAddrSet (interfaceName, interfaceAddress)
    char  *interfaceName;      /* name of interface to configure */
    char  *interfaceAddress;   /* Internet address to assign to interface */
```

DESCRIPTION

This routine assigns an Internet address to a specified network interface. The Internet address can be a host name or a standard Internet address format (e.g., 90.0.0.4). If a host name is specified, it should already have been added to the host table with *hostAdd( )*.

RETURNS

OK, or ERROR if the interface cannot be set.

SEE ALSO

ifLib, *ifAddrGet( ), ifDstAddrSet( ), ifDstAddrGet( )*

ifAddrGet( )

NAME

ifAddrGet( ) - get the Internet address of a network interface

SYNOPSIS

```
STATUS ifAddrGet (interfaceName, interfaceAddress)
    char  *interfaceName;      /* name of interface */
    char  *interfaceAddress;   /* buffer for Internet address */
```

DESCRIPTION

This routine gets the Internet address of a specified network interface and copies it to *interfaceAddress*.

RETURNS

OK or ERROR.

SEE ALSO

ifLib, *ifAddrSet*( ), *ifDstAddrSet*( ), *ifDstAddrGet*( )

ifBroadcastSet( )

NAME

ifBroadcastSet( ) - set the broadcast address for a network interface

SYNOPSIS

```
STATUS ifBroadcastSet (interfaceName, broadcastAddress)
    char  *interfaceName;     /* name of interface to assign */
    char  *broadcastAddress;  /* broadcast address to assign to interface */
```

DESCRIPTION

This routine assigns a broadcast address for the specified network interface. The broadcast address must be a string in standard Internet address format (e.g., 90.0.0.0).

An interface's default broadcast address is its Internet address with a host part of all ones (e.g., 90.255.255.255). This conforms to current ARPA specifications. However, some older systems use an Internet address with a host part of all zeros as the broadcast address.

NOTE

Vx960 automatically accepts a host part of all zeros as a broadcast address, in addition to the default or specified broadcast address. But if Vx960 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, *ifBroadcastGet*( )

ifBroadcastGet( )

NAME

ifBroadcastGet( ) - get the broadcast address for a network interface

SYNOPSIS

```
STATUS ifBroadcastGet (interfaceName, broadcastAddress)
    char  *interfaceName;     /* name of interface */
    char  *broadcastAddress;  /* buffer for broadcast address */
```

DESCRIPTION

This routine gets the broadcast address for a specified network interface. The broadcast address is copied to the buffer *broadcastAddress*.

RETURNS

OK or ERROR.

SEE ALSO

ifLib, *ifBroadcastSet*( )

ifDstAddrSet( )

NAME

ifDstAddrSet( ) - define an address for the other end of a point-to-point link

SYNOPSIS

```
STATUS ifDstAddrSet (interfaceName, dstAddress)
    char  *interfaceName;  /* name of interface to configure */
    char  *dstAddress;     /* Internet address to assign to destination */
```

DESCRIPTION

This routine assigns the Internet address of a machine connected to the opposite end of a point-to-point network connection, such as a SLIP connection. Inherently, point-to-point connection-oriented protocols such as SLIP require that addresses for both ends of a connection be specified.

RETURNS

OK or ERROR.

SEE ALSO

ifLib, *ifAddrSet*( ), *ifDstAddrGet*( )

ifDstAddrGet( )

NAME

ifDstAddrGet( ) - get the Internet address of a point-to-point peer

SYNOPSIS

```
STATUS ifDstAddrGet (interfaceName, dstAddress)
    char  *interfaceName;  /* name of interface */
    char  *dstAddress;     /* buffer for destination address */
```

DESCRIPTION

This routine gets the Internet address of a machine connected to the opposite end of a point-to-point network connection. The Internet address is copied to the buffer *dstAddress.*

RETURNS

OK or ERROR.

SEE ALSO

ifLib, *ifDstAddrSet*( ), *ifAddrGet*( )

ifMaskSet( )

NAME

ifMaskSet( ) - define a subnet for a network interface

SYNOPSIS

```
STATUS ifMaskSet (interfaceName, netMask)
    char  *interfaceName;  /* name of interface for which to set mask */
    int   netMask;         /* subnet mask (e.g. 0xff000000) */
```

DESCRIPTION

This routine allocates additional bits to the network portion of an Internet address. The network portion is specified with a mask that must contain ones in all positions that are to be interpreted as the network portion. This includes all the bits that are normally interpreted as the network portion for the given class of address, plus the bits to be added. Note that all bits must be contiguous.

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*( )

ifMaskGet( )

NAME
ifMaskGet( ) - get the subnet mask for a network interface

SYNOPSIS

```
STATUS ifMaskGet (interfaceName, netMask)
    char  *interfaceName;  /* name of interface */
    int   *netMask;        /* buffer for subnet mask */
```

DESCRIPTION
This routine gets the subnet mask for a specified network interface. The subnet mask is copied to the buffer *netMask*.

RETURNS
OK or ERROR.

SEE ALSO
ifLib, *ifAddrGet*( ), *ifFlagGet*( )

ifFlagChange( )

NAME
ifFlagChange( ) - change the network interface flags

SYNOPSIS

```
STATUS ifFlagChange (interfaceName, flags, on)
    char  *interfaceName;  /* name of the network interface */
    int   flags;           /* the flag to be changed */
    BOOL  on;              /* TRUE=turn on, FALSE=turn off */
```

DESCRIPTION
This routine changes the flags for the specified network interfaces. If the parameter *on* is TRUE, the specified flags are turned on; otherwise, they are turned off. The routines *ifFlagGet*( ) and *ifFlagSet*( ) are called to do the

actual work.

RETURNS

OK or ERROR.

SEE ALSO

ifLib, *ifAddrSet( )*, *ifMaskSet( )*, *ifFlagSet( )*, *ifFlagGet( )*

ifFlagSet( )

NAME

ifFlagSet( ) - specify the flags for a network interface

SYNOPSIS

```
STATUS ifFlagSet (interfaceName, flags)
    char  *interfaceName;  /* name of the network interface */
    int   flags;           /* network flags */
```

DESCRIPTION

This routine changes the flags for a given network interface. Any combination of the following flags can be specified:

IFF_UP (0x1)
IFF_DEBUG (0x4)
IFF_LOOPBACK (0x8)
IFF_NOTRAILERS (0x20)
IFF_PROMISC (0x100)
IFF_ALLMULTI (0x200)
IFF_NOARP (0x80)

NOTE

The following flags can only be set at interface initialization time. Specifying these flags will not change any settings in the interface data structure.

IFF_POINTOPOINT (0x10)
IFF_NOTRAILERS (0x20)
IFF_RUNNING (0x40)

RETURNS

OK or ERROR.

SEE ALSO

ifLib, *ifFlagChange( )*, *ifFlagGet( )*

ifFlagGet( )

NAME

ifFlagGet( ) - get the network interface flags

SYNOPSIS

```
STATUS ifFlagGet (interfaceName, flags)
    char  *interfaceName;  /* name of the network interface */
    int   *flags;          /* network flags returned here */
```

DESCRIPTION

This routine gets the flags for a specified network interface. The flags are copied to the buffer *flags*.

RETURNS

OK or ERROR.

SEE ALSO

ifLib, *ifFlagSet*( )

ifMetricSet( )

NAME

ifMetricSet( ) - specify a network interface hop count

SYNOPSIS

```
STATUS ifMetricSet (interfaceName, metric)
    char  *interfaceName;  /* name of the network interface */
    int   metric;          /* metric for this interface */
```

DESCRIPTION

This routine configures *metric* for a network interface from the host machine to the destination network. This information is used primarily by the IP routing algorithm to compute the relative distance for a collection of hosts connected to each interface. For example, a higher metric for SLIP interfaces can be specified to discourage routing a packet to slower serial line connections. Note that when *metric* is zero, the IP routing algorithm will allow directly sending a packet whose IP network address is not necessarily the same as the local network address.

RETURNS

OK or ERROR.

SEE ALSO

ifLib, *ifMetricGet*( )

ifMetricGet( )

NAME

ifMetricGet( ) - get the metric for a network interface

SYNOPSIS

```
STATUS ifMetricGet (interfaceName, pMetric)
    char  *interfaceName;  /* name of the network interface */
    int   *pMetric;        /* returned interface's metric */
```

DESCRIPTION

This routine retrieves the metric for a specified network interface. The metric is copied to the buffer *pMetric.*

RETURNS

OK or ERROR.

SEE ALSO

ifLib, *ifMetricSet*( )

ifRouteDelete( )

NAME

ifRouteDelete( ) - delete routes associated with a network interface

SYNOPSIS

```
int ifRouteDelete (ifName, unit)
    char  *ifName;  /* name of the interface */
    int   unit;     /* unit number for this interface */
```

DESCRIPTION

This routine deletes all routes that are associated with the specified interface.

RETURNS

The number of routes deleted, or ERROR if an interface is not specified.

SEE ALSO

ifLib

ifunit( )

NAME

ifunit( ) - map an interface name to an interface structure pointer

SYNOPSIS

```
struct ifnet *ifunit(name)
    char  *name;
```

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 ("ei0");
```

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.

INCLUDE

etherLib.h

SEE ALSO

ifLib, etherLib

if_bp

NAME

if_bp - Vx960 & SunOS backplane driver

SYNOPSIS

bpattach( ) - attach the *bp* interface to the network
bpInit( ) - initialize the backplane anchor
bpShow( ) - show information about the backplane network

```
STATUS bpattach (unit, pAnchor, procNum, intType, intArg1, intArg2, intArg3)
STATUS bpInit (pAnchor, pMem, memSize, tasOK)
VOID bpShow (bpName, zero)
```

DESCRIPTION

This module implements the Vx960 backplane network driver. The backplane driver allows several CPUs to communicate via shared memory. Usually, the first CPU to boot in a card cage is considered the backplane master. This CPU has either dual-ported memory or an additional memory board which all other CPUs can access. Each CPU must have some way to be interrupted by another CPU and to be able interrupt all other CPUs. There are three interrupt types: polling, mailboxes, VMEbus interrupts. Polling is used when there are no hardware interrupts; each CPU spawns a polling task to manage transfers. Mailbox interrupts are the preferred method because they do not require an interrupt level. Using VMEbus interrupts is much better than polling but may require hardware jumpers.

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

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

MEMORY LAYOUT

The following diagram shows the memory layout of a backplane network. All pointers in shared memory are relative to the start of shared memory, since dual-ported memory may appear in different places for different CPUs.

```
--------------------------- low address (anchor)
|       ready value        |    1234567
|        heartbeat         |    increments 1/sec
|   pointer to bp header   |
|        watchdog          |    for backplane master CPU
 -------------------------
--------------------------

the backplane header may be contiguous or
allocated elsewhere on the master CPU
--------------------------

--------------------------- backplane header
|    backplane header      |    1234567
|       num CPUs           |    unused
|    ethernet address      |    (6 bytes)
|   pointer to free ring   |
|--------------------------|

|--------------------------| cpu descriptor
|         active           |    true/false
|   processor number       |    0-NCPU
|         unit             |    0-NBP
|   pointer to input ring  |
|    interrupt type        |    POLL | MAILBOX    |  BUS
|    interrupt arg1        |    none | addr space | level
|    interrupt arg2        |    none | address    | vector
|    interrupt arg3        |    none | value      | none
|--------------------------|
           ...               repeated MAXCPU times
|--------------------------|
|       free ring          |    contains pointers to buffer nodes
|--------------------------|
|       input ring 1       |    contains pointers to buffer nodes
|--------------------------|
    ...
|--------------------------|
|       input ring n       |
|--------------------------|

|--------------------------| buffer node 1
|     address, length      |
|--------------------------|
|      data buffer 1       |
|--------------------------|
    ...
|--------------------------| buffer node m
|     address, length      |
|--------------------------|
|      data buffer m       |
--------------------------- high address
```

SEE ALSO

Programmer's Guide: Network

bpattach( )

NAME

bpattach( ) - attach the *bp* interface to the network

SYNOPSIS

```
STATUS bpattach (unit, pAnchor, procNum, intType, intArg1, intArg2, intArg3)
    int   unit;      /* backplane unit number */
    char  *pAnchor;  /* bus pointer to bp anchor */
    int   procNum;   /* processor number in backplane */
    int   intType;   /* interrupt type: poll, bus, mailbox */
    int   intArg1;   /* as per interrupt type */
    int   intArg2;   /* as per interrupt type */
    int   intArg3;   /* as per interrupt type */
```

DESCRIPTION

This routine makes the *bp* interface available by filling in the network interface record. The system will initialize the interface when it is ready to accept packets.

RETURN

OK or ERROR.

SEE ALSO

if_bp

bpInit( )

NAME

bpInit( ) - initialize the backplane anchor

SYNOPSIS

```
STATUS bpInit (pAnchor, pMem, memSize, tasOK)
    char  *pAnchor;  /* backplane anchor address */
    char  *pMem;     /* start of backplane shared memory, NONE=alloc */
    int   memSize;   /* num bytes in bp shared memory, 0=0x100000 */
    BOOL  tasOK;     /* TRUE = hardware can do test-and-set */
```

DESCRIPTION

This routine initializes the backplane anchor. Typically, the *pAnchor* and *pMem* parameters both point to the same block of shared memory. If the first processor is dual-porting its memory, then, by convention, the anchor is at 0x600 (only 16 bytes are required) and the start of memory *pMem* is dynamically allocated by using the value NONE (-1). Memory size ranges from a minimum of 64K to a comfortable 512K, although this may be larger. The *tasOK* parameter is provided for CPUs (e.g., Sun-3) that do not support the test-and-set instruction. If any test-and-set deficient CPUs are in the system, then all CPUs must use the software "test-and-set".

RETURNS

OK, or ERROR if data structures cannot be set up, or memory is insufficient.

SEE ALSO

if_bp

bpShow( )

NAME

bpShow( ) - show information about the backplane network

SYNOPSIS

```
VOID bpShow (bpName, zero)
    char  *bpName;   /* backplane interface name (NULL == "bp0") */
    BOOL  zero;      /* TRUE = zap totals */
```

DESCRIPTION

This routine shows information about the different CPUs configured in the backplane network. It also prints error statistics (and zeros).

```
-> bpShow
Anchor at 0x800000
heartbeat = 705, header at 0x800010, free pkts = 237.

cpu int type    arg1       arg2       arg3     queued pkts rd index
--- -------- ---------- ---------- ---------- ----------- --------
 0  poll           0x0        0x0        0x0       0          27
 1  poll           0x0        0x0        0x0       0          11
 2  bus-int        0x3       0xc9        0x0       0          9
 3  mbox-2        0x2d     0x8000        0x0       0          1

input packets = 192     output packets = 164
output errors = 0       collisions = 0
```

```
        value = 1 = 0x1
```

RETURNS

N/A

SEE ALSO

if_bp

if_enp

NAME

if_enp - CMC ENP 10/L Ethernet interface driver

SYNOPSIS

enpattach( ) - attach the *enp* interface to the network

```
STATUS enpattach (unit, addr, ivec, ilevel)
```

DESCRIPTION

This module implements the CMC ENP 10/L network interface driver. There is one user-callable routine, *enpattach*( ).

enpattach( )

NAME

enpattach( ) - attach the *enp* interface to the network

SYNOPSIS

```
STATUS enpattach (unit, addr, ivec, ilevel)
    int   unit;    /* unit number */
    char  *addr;   /* address of enp's shared memory */
    int   ivec;    /* interrupt vector to connect to */
    int   ilevel;  /* interrupt level */
```

DESCRIPTION

This routine attaches an *enp* interface to the network if the device exists. The system will initialize the interface when it is ready to accept packets.

RETURNS

OK or ERROR.

SEE ALSO

if_enp, ifLib, netShow

if_ex

NAME

if_ex - Excelan EXOS 201/202/302 Ethernet interface driver

SYNOPSIS

exattach( ) - attach the *ex* interface to the network

```
STATUS exattach (unit, devAdrs, ivec, ilevel)
```

DESCRIPTION

This module implements the Excelan EXOS 201/202/302 Ethernet interface driver. There is one user-callable routine, *exattach*( ).

exattach( )

NAME

exattach( ) - attach the *ex* interface to the network

SYNOPSIS

```
STATUS exattach (unit, devAdrs, ivec, ilevel)
    int   unit;      /* logical number of this interface */
    char  *devAdrs;  /* bus address */
    int   ivec;      /* interrupt vector */
    int   ilevel;    /* interrupt level */
```

DESCRIPTION

This routine attaches an *ex* Ethernet interface to the network if the interface exists. The 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.

RETURNS

OK or ERROR.

SEE ALSO

if_ex, ifLib, netShow

if_ln

NAME

if_ln - LANCE Ethernet interface driver

SYNOPSIS

lnattach( ) - attach the *ln* interface to the network

```
STATUS lnattach (unit, devAdrs, ivec, ilevel, memAdrs, memSize, memWidth, ...
```

DESCRIPTION

This module implements the LANCE Ethernet interface driver. There is one user-callable routine, *lnattach*( ).

lnattach( )

NAME

lnattach( ) - attach the *ln* interface to the network

SYNOPSIS

```
STATUS lnattach (unit, devAdrs, ivec, ilevel, memAdrs, memSize, memWidth,
                 usePadding, bufSize)
    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 malloced */
    int    memWidth;    /* byte-width of data (-1 == any width)      */
    BOOL   usePadding;  /* use padding when accessing RAP? */
    int    bufSize;     /* size of a buffer in the LANCE ring */
```

DESCRIPTION

This routine attaches an *ln* Ethernet interface to the network if the interface exists. The 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 *memAdrs* parameter specifies the location of the interface memory pool; if NONE, the memory pool will be malloc'ed.

The *memWidth* parameter sets the memory pool's data port width (in bytes); if NONE, any data width will be used. The *bufSize* parameter specifies the size of a buffer in the LANCE ring; if zero, the default value of LN_BUFSIZE is used. If *memWidth* is NONE and *bufSize* is LN_BUFSIZE, the driver may attempt to loan out buffers for increased network performance.

The *usePadding* parameter should be set to TRUE if the board maps RAP on a 4-byte instead of a 2-byte boundary (e.g., the Tadpole TP32V).

RETURNS

OK or ERROR.

BUGS

To zero out LANCE data structures, this routine uses *bzero*( ), which ignores the *memWidth* specification and may use any size data access to write to memory.

SEE ALSO

if_ln, ifLib

if_sl

NAME

if_sl - serial line IP (SLIP) network interface driver

SYNOPSIS

slipInit( ) - initialize the SLIP interface
slipBaudSet( ) - set the baud rate for a SLIP interface
slattach( ) - attach a SLIP interface to the network
slipDelete( ) - delete a SLIP interface

```
STATUS slipInit (unit, devName, myAddr, peerAddr, baud)
STATUS slipBaudSet (unit, baud)
STATUS slattach (unit, fd)
STATUS slipDelete (unit)
```

DESCRIPTION

This module implements the Vx960 Serial Line IP (SLIP) network driver. The SLIP driver allows Vx960 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*( ), specifying the Internet address for both sides of the SLIP point-to-point link and the name of the tty device on the local host. It 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)

END is used to denote the end of an IP packet. The ESC (not to be confused with ASCII ESC) is used to circumvent potential occurrences of the END character within a packet as well as the ESC character itself. When an END character is to be embedded within a packet, SLIP will send a sequence of "ESC TRANS_END" instead, to avoid confusion between a SLIP-specific END and actual data whose value is the END character. To send ESC character itself, "ESC TRANS_ESC" is used.

On the receiving side of the connection, SLIP uses the opposite actions to decode the SLIP packets. Whenever END character is received, SLIP will assume a full IP packet has been received and send it up to the IP layer.

IMPLEMENTATION

The write side of a SLIP connection is an independent task. Each SLIP interface will have its own output task that will send SLIP packets over a particular tty device channel. Whenever a packet is ready to be sent out, the SLIP driver will activate this task by giving a semaphore. When the semaphore is available, the output task will perform packetization (as explained above) and write 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 SLIP packet, it is queued to IP's input queue.

SEE ALSO

tyLib, John Romkey: RFC-1055, entitled *A Nonstandard for Transmission of IP Datagrams Over Serial Lines: SLIP*

ACKNOWLEDGEMENT

This program is based on the original work done by Rick Adams of Center for Seismic Studies and Chris Torek of University of Maryland.

slipInit( )

NAME

slipInit( ) - initialize the SLIP interface

SYNOPSIS

```
STATUS slipInit (unit, devName, myAddr, peerAddr, baud)
    int   unit;       /* SLIP device unit number (0 - NSLIP-1) */
    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 for this SLIP device */
                      /*   0 == don't set baud rate */
```

DESCRIPTION

This routine initializes a SLIP device. Its parameters specify the name of the tty device and 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.

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 NSLIP (20) units may be created.

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 a the address of the remote host is 192.10.1.2. The baud rate will be the default rate for /tyCo/1.

```
    slipInit (0, "/tyCo/1", "192.10.1.1", "192.10.1.2", 0);
```

RETURNS

OK, or ERROR if the device cannot be opened, memory is insufficient, or the route is invalid.

SEE ALSO

if_sl

slipBaudSet( )

NAME

slipBaudSet( ) - set the baud rate for a SLIP interface

SYNOPSIS

```
STATUS slipBaudSet (unit, baud)
    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 not initialized.

SEE ALSO

if_sl

slattach( )

NAME

slattach( ) - attach a SLIP interface to the network

SYNOPSIS

```
STATUS slattach (unit, fd)
    int  unit;  /* SLIP device unit number */
    int  fd;    /* file descriptor of tty device for SLIP interface */
```

DESCRIPTION

This routine attaches an *sl* Ethernet interface to the network. It is usually called by *slipInit*( ). It inserts a pointer to the SLIP interface data structure into a linked list of available network interfaces.

RETURNS

OK or ERROR.

SEE ALSO

if_sl

slipDelete( )

NAME

slipDelete( ) - delete a SLIP interface

SYNOPSIS

```
STATUS slipDelete (unit)
    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

inetLib

NAME

inetLib - Internet address manipulation routines

SYNOPSIS

inet_addr( ) - convert a dot notation Internet address to a long integer
inet_lnaof( ) - get the local address (host number) from the Internet address
inet_makeaddr_b( ) - form an Internet address from the network and host numbers
inet_makeaddr( ) - form an Internet address from the 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 a string to an address
inet_ntoa_b( ) - convert network dot notation address to ASCII
inet_ntoa( ) - convert network dot notation address to ASCII

```
u_long inet_addr (inetString)
int inet_lnaof (inetAddress)
VOID inet_makeaddr_b (netAddr, hostAddr, pInetAddr)
struct in_addr inet_makeaddr (netAddr, hostAddr)
int inet_netof (inetAddress)
VOID inet_netof_string (inetString, netString)
u_long inet_network (inetString)
VOID inet_ntoa_b (inetAddress, pString)
char *inet_ntoa (inetAddress)
```

DESCRIPTION

The library inetLib provides routines for manipulating Internet addresses, including the UNIX BSD 4.3 "inet_" routines. It includes routines for converting between character addresses in Internet standard dot notation and integer addresses, routines for extracting the network and host portions out of an Internet address, and routines for constructing Internet addresses given the network and host address parts.

All Internet addresses are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine format integer values.

INTERNET ADDRESSES

Internet addresses are typically specified in dot notation or as a 4-byte number. Values specified using the dot notation take one of the following forms:

a.b.c.d
a.b.c

a.b
a

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address. Note that when an Internet address is viewed as a 32-bit integer quantity on any i960 family machine, the bytes referred to above appear as "a.b.c.d" are ordered from right to left.

When a three-part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right-most two bytes of the network address. This makes the three-part address format convenient for specifying Class B network addresses as "128.net.host".

When a two-part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right-most three bytes of the network address. This makes the two-part address format convenient for specifying Class A network addresses as "net.host".

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as parts in a dot notation may be decimal, octal, or hexadecimal, as specified in the C language. That is, a leading 0x or 0X implies hexadecimal; otherwise, a leading 0 implies octal. With no leading 0, the number is interpreted as decimal.

INCLUDE FILES

inetLib.h, in.h

SEE ALSO

UNIX BSD 4.3 documentation for inet(3N), *Programmer's Guide: Network*
Stevens, Section 6.5

inet_addr( )

NAME

inet_addr( ) - convert a dot notation Internet address to a long integer

SYNOPSIS

```
u_long inet_addr (inetString)
    char  *inetString;
```

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

```
 inet_addr ("90.0.0.2");
```

Returns 0x5a000002 on a big-endian machine. Returns 0x200005a on a little-endian machine like the i960 microprocessor.

RETURNS

The Internet address, or ERROR.

SEE ALSO

inetLib, *Programmer's Guide: Network*

inet_lnaof( )

NAME

inet_lnaof( ) - get the local address (host number) from the Internet address

SYNOPSIS

```
int inet_lnaof (inetAddress)
    int   inetAddress;   /* inet address 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

```
 inet_lnaof (0x5a0000002);
```

returns 2 on a big-endian machine. Returns 0x5a on little-endian machine like the i960 microprocessor.

RETURNS

The local address portion of *inetAddress*.

SEE ALSO

inetLib, *Programmer's Guide: Network*

inet_makeaddr_b( )

NAME

inet_makeaddr_b( ) - form an Internet address from the network and host numbers

SYNOPSIS

```
VOID inet_makeaddr_b (netAddr, hostAddr, pInetAddr)
    int              netAddr;     /* network part of the inet address */
    int              hostAddr;    /* host part of the inet address */
    struct in_addr   *pInetAddr;  /* where to return the inet address */
```

DESCRIPTION

This routine constructs the Internet address from the network number and local host address. This routine is identical to the UNIX *inet_makeaddr*( ) routine except that a buffer for the resulting value must provided.

EXAMPLE

```
 inet_makeaddr_b (0x5a, 2, pInetAddr);
```

Returns the address 0x5a000002 in the location pointed to by *pInetAddr* on a big-endian machine. Returns 0x200005a on a little-endian machine like the i960 microprocessor.

RETURNS

N/A

SEE ALSO

inetLib, *Programmer's Guide: Network*

inet_makeaddr( )

NAME

inet_makeaddr( ) - form an Internet address from the network and host numbers

SYNOPSIS

```
struct in_addr inet_makeaddr (netAddr, hostAddr)
    int  netAddr;
    int  hostAddr;
```

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

```
 inet_makeaddr (0x5a, 2);
```

returns the address 0x5a000002 to the structure *in_addr*.

RETURNS

The network address in an in_addr structure.

SEE ALSO

inetLib, *inet_makeaddr_b*( ), *Programmer's Guide: Network*

inet_netof( )

NAME

inet_netof( ) - return the network number from an Internet address

SYNOPSIS

```
int inet_netof (inetAddress)
    struct in_addr  inetAddress;  /* the inet address */
```

DESCRIPTION

This routine extracts the network portion of an Internet address.

EXAMPLE

```
 inet_netof (0x5a000002);
```

returns 0x5a.

RETURNS

The network portion of *inetAddress*.

SEE ALSO

inetLib, *Programmer's Guide: Network*

inet_netof_string( )

NAME

inet_netof_string( ) - extract the network address in dot notation

SYNOPSIS

```
VOID inet_netof_string (inetString, netString)
    char  *inetString;  /* Internet address to extract local portion from */
    char  *netString;   /* net Internet address to return */
```

DESCRIPTION

This routine extracts the network Internet address from a host Internet address (specified in dot notation). The routine handles class A, B, and C network addresses. The buffer *netString* should be INET_ADDR_LEN bytes long.

NOTE

This is the only routine in inetLib that handles subnet masks correctly.

EXAMPLE 1

```
    inet_netof_string ("90.0.0.2", netString);
```

Returns “90.0.0.0” in *netString*.

RETURNS

N/A

EXAMPLE 2 - for subnet mask

```
    hostAdd("lshost", "143.185.6.9")
    if MaskSet("ei0", 0xffffff00)
    inet_Netof_string("143.185.6.44", netString)
```

RETURNS

Returns “143.185.6.0” in *netString*.

SEE ALSO

inetLib

inet_network( )

NAME

inet_network( ) - convert an Internet network number from a string to an address

SYNOPSIS

```
u_long inet_network (inetString)
    char  *inetString;  /* string version of inet addrs */
```

DESCRIPTION

This routine forms a network address given an ASCII Internet network number.

EXAMPLE

```
 inet_network ("90");
```

returns 0x5a.

RETURNS

The Internet address version of an ASCII string.

SEE ALSO

inetLib, *Programmer's Guide: Network*

inet_ntoa_b( )

NAME

inet_ntoa_b( ) - convert network dot notation address to ASCII

SYNOPSIS

```
VOID inet_ntoa_b (inetAddress, pString)
    struct in_addr  inetAddress;
    char            *pString;     /* where to return ASCII string */
```

DESCRIPTION

This routine converts an Internet address in network format to dot notation.

This routine is identical to the UNIX *inet_ntoa*( ) routine except that a buffer of size INET_ADDR_LEN must be provided.

EXAMPLE

```
 inet_ntoa_b (0x200005a, pString);
```

returns the string "90.0.0.2" in *pString*.

RETURNS

N/A

SEE ALSO

inetLib

inet_ntoa( )

NAME

inet_ntoa( ) - convert network dot notation address to ASCII

SYNOPSIS

```
char *inet_ntoa (inetAddress)
    struct in_addr  inetAddress;
```

DESCRIPTION

This routine converts an Internet address in network format to dot notation.

WARNING

This routine is supplied for UNIX compatibility only. Each time this routine is called, 18 bytes are allocated from memory. Use *inet_ntoa_b*( ) instead.

EXAMPLE

```
 inet_ntoa (0x200005a);
```

returns a pointer to the string "90.0.0.2".

RETURNS

A pointer to the string version of an Internet address.

SEE ALSO

inetLib, *inet_ntoa_b*( ), *Programmer's Guide: Network*

intALib

NAME

intALib - interrupt library assembly language routines

SYNOPSIS

intLevelSet( ) - set interrupt level

```
int intLevelSet (level)
```

DESCRIPTION

These routines are used to support various functions associated with interrupts from C. The routine intLevelSet (2) changes the current interrupt level of the processor.

SEE ALSO

intLib (1)

intLevelSet( )

NAME

intLevelSet( ) - set interrupt level

SYNOPSIS

```
int intLevelSet (level)
    int level;          /* new interrupt level mask (0-31) */
```

DESCRIPTION

This routine changes the interrupt mask in the status register to take on the value specified by level. It is strongly advised that the level be in the range 0-31.

Setting interrupts involves privileged instructions, thus user-level tasks must trap to supervisor level before executing this routine. This routine should only be called in supervisor mode.

RETURNS

previous interrupt level 0-31.

SEE ALSO

intALib

intLib

NAME

intLib - interrupt subroutine library

SYNOPSIS

intConnect( ) - connect C routine to hardware interrupt
intHandlerCreate( ) - construct interrupt handler for C routine
intLockLevelSet( ) - set the current interrupt lock-out level
intLockLevelGet( ) - get the current interrupt lock-out level
intRestrict( ) - restrict interrupt context from using a routine
intContext( ) - determine if we are in interrupt context or task context
intCount( ) - get current interrupt nesting depth
intLock( ) - lock out interrupts
intUnlock( ) - cancel effect of intLock
intVecBaseSet( ) - set the vector base address
intVecBaseGet( ) - get the vector base address
intVecSet( ) - set CPU vector
intVecGet( ) - get vector

```
STATUS intConnect (vector, routine, parameter)
FUNCPTR intHandlerCreate (routine, parameter)
VOID intLockLevelSet (newLevel)
int intLockLevelGet ()
STATUS intRestrict ()
BOOL intContext ()
int intCount ()
int intLock ()
int intUnlock (level)
VOID intVecBaseSet (baseAddr)
FUNCPTR *intVecBaseGet ()
VOID intVecSet (vector, function)
FUNCPTR intVecGet (vector)
```

DESCRIPTION

This library provides routines to manipulate and connect to hardware interrupts and exceptions. Most importantly, any C language routine can be connected to any exception, interrupt, or trap by calling the routine intConnect. Interrupt vectors can be accessed directly by the routines intVecSet and intVecGet. The vector base register can be accessed by the routines in intVecBaseGet. Tasks can lock and unlock interrupts by calling the routines intLock and intUnlock. The routines intCount and intContext can be used to determine whether the CPU is running in an interrupt context or in a normal task context.

INTERRUPT VECTORS AND NUMBERS

Most of the routines in this library take an interrupt vector as a parameter, which is the byte offset into the vector table. Macros are provided to convert these interrupt vectors to interrupt numbers and vice versa: IVEC_TO_INUM(intVector) changes a vector to a number. INUM_TO_IVEC(intNumber) turns a number into a vector. TRAPNUM_TO_IVEC(trapNumber) converts a trap number to a vector.

EXAMPLE

There are instances where it is desired 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 *
```

intConnect( )

NAME

intConnect( ) - connect C routine to hardware interrupt

SYNOPSIS

```
STATUS intConnect (vector, routine, parameter)
    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 the specified C routine to the specified interrupt vector. When an interrupt occurs that vectors through the specified address, the routine will be called with the specified parameter. The routine will be invoked in supervisor mode at interrupt level. A proper C environment will have been 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.

This routine simply calls intHandlerCreate (2) and intVecSet (2). It is the address of the handler returned by intHandlerCreate (2) that actually gets put in the interrupt vector.

RETURNS

OK or
ERROR if unable to build interrupt handler

SEE ALSO

intLib, intHandlerCreate (2), intVecSet (2)

intHandlerCreate( )

NAME

intHandlerCreate( ) - construct interrupt handler for C routine

SYNOPSIS

```
FUNCPTR intHandlerCreate (routine, parameter)
    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 (2). The routine will be invoked in supervisor mode at interrupt level. A proper C environment will have been 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.

IMPLEMENTATION

This routine builds an interrupt handler of the following form in allocated memory.

```
0x90403000, 0x00000000,   ld      _errno, r8
0x86003000, 0x00000000,   callx   _intEnt
0x5c201e01,               mov     1, r4
0x65210284,               modpc   r4, r4, r4
0x5c281601,               mov     sp, r5
0x8c200040,               lda     0x40,r4
0x59084004,               addo    r4, sp, sp
0xb2815000,               stq      g0,(r5)
0xb2a16010,               stq     g4,0x10(r5)
```

```
0xb2c16020,                 stq     g8,0x20(r5)
0xa2e16030,                 stt     g12,0x30(r5)
0x8c803000, 0x00000000,     lda     parameter, g0
0x5cf01e00,                 mov        0,g14
0x86003000, 0x00000000,     callx   routine
0xb0815000,                 ldq     (r5), g0
0xb0a16010,                 ldq     0x10(r5),g4
0xb0c16020,                 ldq     0x20(r5),g8
0xa0e16030,                 ldt     0x30(r5),g12
0x5c081605,                 mov     r5, sp
0x84003000, 0x00000000      bx      _intExit
```

RETURNS

pointer to new interrupt handler, or
NULL if out of memory

SEE ALSO

intLib

intLockLevelSet( )

NAME

intLockLevelSet( ) - set the current interrupt lock-out level

SYNOPSIS

```
VOID intLockLevelSet (newLevel)
    int newLevel;              /* new interrupt level */
```

DESCRIPTION

This routine prepares the appropriate interrupt mask and stores it in the globally accesible intLockMask.

SEE ALSO

intLib

intLockLevelGet( )

NAME

intLockLevelGet( ) - get the current interrupt lock-out level

SYNOPSIS

```
int intLockLevelGet ()
```

DESCRIPTION

This routine returns the interrupt level currently stored in the interrupt lockout mask.

SEE ALSO

intLib

intRestrict( )

NAME

intRestrict( ) - restrict interrupt context from using a routine

SYNOPSIS

```
STATUS intRestrict ()
```

DESCRIPTION

This routine returns OK if and only if we are executing in a task's context and ERROR if called within an interrupt context.

SEE ALSO

intLib, *INT_RESTRICT*( ) macro in intLib.h.

RETURNS

TRUE or FALSE

intContext( )

NAME

intContext( ) - determine if we are in interrupt context or task context

SYNOPSIS

```
BOOL intContext ()
```

DESCRIPTION

This routine returns true if and only if we are executing in interrupt context and NOT in a meaningful task context.

RETURNS

TRUE or FALSE

SEE ALSO

intLib

intCount( )

NAME

intCount( ) - get current interrupt nesting depth

SYNOPSIS

```
int intCount ()
```

DESCRIPTION

This routine returns the number of interrupts that are currently nested.

CAVEAT

While it may appear that intCount will never return a value greater than 31, remember that ISR's may modify the interrupt mask to allow lower priority interrupts.

RETURNS

number of nested interrupts

SEE ALSO

intLib

intLock( )

NAME

intLock( ) - lock out interrupts

SYNOPSIS

```
int intLock ()
```

DESCRIPTION

This routine is used to disable interrupts. The interrupt level is set to the lock-out level (level 31 by default). The routine returns the previous interrupt level, and this should be restored by a call to intUnlock (2).

IMPORTANT CAVEAT

intLock can be called both from interrupt level and from task level. When called from a task context, the interrupt lock level is part of the task's context. Locking out interrupts does not prevent rescheduling. Thus, if a task locks out interrupts and then invokes kernel services that cause the task to block (e.g. taskSuspend (2) or taskDelay (2)) or causes a higher priority task to be ready (e.g. semGive (2) or taskResume (2)), then rescheduling will occur and interrupts will be unlocked while other tasks run. Rescheduling may be explicitly disabled with taskLock (2).

EXAMPLE

```
oldLevel = intLock ();
/* ...work with interrupts locked out... *
intUnlock (oldLevel);
```

RETURNS

previous interrupt level

SEE ALSO

intLib, intUnlock (2), taskLock (2)

intUnlock( )

NAME

intUnlock( ) - cancel effect of intLock

SYNOPSIS

```
int intUnlock (level)
    int level;     /* level to which to restore interrupts */
```

DESCRIPTION

This routine is used to re-enable interrupts that have been disabled by intLock (2). Use the level obtained from the preceding intLock (2) call.

SEE ALSO

intLib, intLock (2)

intVecBaseSet( )

NAME

intVecBaseSet( ) - set the vector base address

SYNOPSIS

```
VOID intVecBaseSet (baseAddr)
    FUNCPTR *baseAddr;       /* new vector base address */
```

DESCRIPTION

This routine sets the vector base address. The CPU's vector base register is set to the specified value, and subsequent calls to intVecGet/intVecSet will use this base address. The vector base address is fixed, until changed by calls to this routine.

NOTE

The interrupt vector table is located in sysALib.s and moving it by intVBaseSet would require resetting the processor. This is NOT YET IMPLEMENTED.

For the i960 microprocessor the vector base is cached on chip in the PRCB, so it cannot be set from this routine.

SEE ALSO

intLib, intVecBaseGet (2), intVecGet (2), intVecSet (2)

intVecBaseGet( )

NAME

intVecBaseGet( ) - get the vector base address

SYNOPSIS

```
FUNCPTR *intVecBaseGet ()
```

DESCRIPTION

This routine returns the current vector base address that has been set via the intVecBaseSet (2) routine.

RETURNS

current vector base address

SEE ALSO

intLib, intVecBaseSet (2)

intVecSet( )

NAME

intVecSet( ) - set CPU vector

SYNOPSIS

```
VOID intVecSet (vector, function)
    FUNCPTR     *vector;     /* vector offset */
    FUNCPTR     function;    /* address to place in vector */
```

DESCRIPTION

This routine sets the specified exception/interrupt vector to the specified address. The vector is specified as an offset into the CPU's vector table.

NOTE

i960 vectors 0-7 are illeagal vectors, use of them will put the vector in the priorities/pending portion of the table which will yield undesirable actions.

The 80960CA caches the NMI vector in internal ram at system power up. This is where the vector is taken when the NMI occurs. Therefore, this routine checks to see if the vector we are changing is the NMI vector, and writes it to internal ram if it is.

SEE ALSO

intLib, intVecBaseSet (2), intVecGet (2)

intVecGet( )

NAME

intVecGet( ) - get vector

SYNOPSIS

```
FUNCPTR intVecGet (vector)
    FUNCPTR *vector;    /* vector offset */
```

DESCRIPTION

This routine returns the current value of the specified exception/interrupt vector. The vector is specified as an offset into the CPU's vector table.

NOTE

For the i960 the interrupt table location is cached in the prcb on chip. The location is returned by intVecBaseGet (2)

RETURNS

current value of specified vector

SEE ALSO

intLib, intVecSet (2), intVecBaseSet (2)

ioLib

NAME

ioLib - I/O interface library

SYNOPSIS

creat( ) - create a file
remove( ) - delete a file
unlink( ) - delete a file
delete( ) - delete a file
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 a file-specific control function
lseek( ) - set 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
getwd( ) - get the current default path
ioGlobalStdSet( ) - set the fd for global standard input/output/error
ioGlobalStdGet( ) - get the fd for global standard input/output/error
ioTaskStdSet( ) - set the fd for task standard input/output/error
ioTaskStdGet( ) - get the fd for task standard input/output/error

```
int creat (name, flag)
STATUS remove (name)
STATUS unlink (name)
STATUS delete (name)
int open (name, flags, mode)
STATUS close (fd)
STATUS rename (oldname, newname)
int read (fd, buffer, maxbytes)
int write (fd, buffer, nbytes)
int ioctl (fd, function, arg)
int lseek (fd, offset, whence)
STATUS ioDefPathSet (name)
VOID ioDefPathGet (pathname)
STATUS chdir (pathname)
char *getcwd (buffer, size)
char *getwd (pathname)
```

```
VOID ioGlobalStdSet (stdFd, newFd)
int ioGlobalStdGet (stdFd)
VOID ioTaskStdSet (taskId, stdFd, newFd)
int ioTaskStdGet (taskId, stdFd)
```

DESCRIPTION

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

- Interfaces to the seven basic driver-provided functions: *creat*( ), *delete*( ), *open*( ), *close*( ), *read*( ), *write*( ), and *ioctl*( ).
- Interfaces to several file system functions: *rename*( ) and *lseek*( ).
- Routines to set and get the current working directory.
- Routines to assign task and global standard fds.

FILE DESCRIPTORS

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

Three fds are reserved and have special meanings:

0 - standard input
1 - standard output
2 - standard error output

Vx960 allows two levels of redirection. First, there is a global assignment of the three standard fds. By default, tasks will use this global assignment. The global assignment of the three standard fds is controlled by the routines *ioGlobalStdSet*( ) and *ioGlobalStdGet*( ).

Second, individual tasks may override the global assignment of these fds with their own assignments that apply only to that task. The assignment of task-specific standard fds is controlled by the routines *ioTaskStdSet*( ) and *ioTaskStdGet*( ).

INCLUDE FILE

ioLib.h

SEE ALSO

iosLib, *Programmer's Guide: I/O System*

creat( )

NAME

creat( ) - create a file

SYNOPSIS

```
int creat (name, flag)
    char  *name;  /* name of the file to create */
    int   flag;   /* READ, WRITE, or UPDATE     */
```

DESCRIPTION

This routine creates a file called *name* and opens it with a specified *flag*. The routine determines on which device to create the file, and then calls that 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 READ, WRITE, or UPDATE for the duration that 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.

RETURNS

A file descriptor number, or ERROR if a filename is not specified, the device does not exist, no fds are available (see *iosInit*( )), or the driver returns ERROR.

SEE ALSO

ioLib, *open*( ), *Programmer's Guide: I/O System*

remove( )

NAME

remove( ) - delete a file

SYNOPSIS

```
STATUS remove (name)
    char  *name;  /* name of the file to delete */
```

DESCRIPTION

This routine deletes a specified file. It performs the same function as *delete*( ). It is provided for ANSI compatibility.

RETURNS

OK if there is no delete routine for the device or if the driver returns OK; ERROR if there is no such device or if the driver returns ERROR.

SEE ALSO

ioLib, *delete*( ), *unlink*( ), *Programmer's Guide: I/O System*

unlink( )

NAME

unlink( ) - delete a file

SYNOPSIS

```
STATUS unlink (name)
    char  *name;  /* name of the file to delete */
```

DESCRIPTION

This routine deletes a specified file. It performs the same function as *delete*( ), and it 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, *delete*( ), *remove*( ), *Programmer's Guide: I/O System*

delete( )

NAME

delete( ) - delete a file

SYNOPSIS

```
STATUS delete (name)
    char  *name;  /* name of the file to delete */
```

DESCRIPTION

This routine deletes a specified file. The routine 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, *Programmer's Guide: I/O System*

open( )

NAME

open( ) - open a file

SYNOPSIS

```
int open (name, flags, mode)
    char  *name;  /* name of the file to open        */
    int   flags;  /* READ, WRITE, UPDATE, or O_CREAT */
    int   mode;   /* mode of file to be created      */
                  /* (UNIX chmod style)              */
```

DESCRIPTION

This routine opens a file for reading, writing, or updating, and returns an fd for that file. The arguments to *open*( ) are the filename and the type of access:

READ (or O_RDONLY)	- open for reading only
WRITE (or O_WRONLY)	- open for writing only
UPDATE (or O_RDWR)	- open for reading and writing
TRUNC (or O_TRUNC)	- open with 0 bytes

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.

RETURNS

A file descriptor number, or ERROR if a file name is not specified, the device does not exist, no fds are available (see *iosInit*( )), or the driver returns ERROR.

SEE ALSO

ioLib, *creat*( ), *Programmer's Guide: I/O System*

close( )

NAME

close( ) - close a file

SYNOPSIS

```
STATUS close (fd)
    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

Status of the driver close routine, or ERROR if the fd is invalid.

SEE ALSO

ioLib, *Programmer's Guide: I/O System*

rename( )

NAME

rename( ) - change the name of a file

SYNOPSIS

```
STATUS rename (oldname, newname)
    char  *oldname;  /* name of file to rename */
    char  *newname;  /* name with which to rename file */
```

DESCRIPTION

This routine changes the name of a file from *oldfile* to *newfile.*

RETURNS

OK, or ERROR if the file could not be opened or renamed.

SEE ALSO

ioLib

read( )

NAME

read( ) - read bytes from a file or device

SYNOPSIS

```
int read (fd, buffer, maxbytes)
    int   fd;        /* file descriptor from which to read       */
    char  *buffer;   /* pointer to buffer to receive bytes       */
    int   maxbytes;  /* max number of bytes to read into buffer */
```

DESCRIPTION

This routine reads a number of bytes (less than or equal to *maxbytes*) from a specified fd 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 fd does not exist or the driver returns ERROR.

SEE ALSO

ioLib, *Programmer's Guide: I/O System*

write( )

NAME

write( ) - write bytes to a file

SYNOPSIS

```
int write (fd, buffer, nbytes)
    int   fd;       /* file descriptor on which to write      */
    char  *buffer;  /* buffer containing bytes to be written */
    int   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 fd does not exist or the driver returns ERROR.

SEE ALSO

ioLib, *Programmer's Guide: I/O System*

ioctl( )

NAME

ioctl( ) - perform a file-specific control function

SYNOPSIS

```
int ioctl (fd, function, arg)
    int  fd;        /* file descriptor    */
    int  function;  /* function code      */
    int  arg;       /* arbitrary argument */
```

DESCRIPTION

This routine performs a device-specific function on a device. Most requests are passed on to the driver for handling. The following example, which places the filename of the fd in *nameBuf*, is handled at the I/O interface level:

```
    ioctl (fd, FIOGETNAME, &nameBuf)
```

Since the availability of *ioctl*( ) functions is driver specific, these functions are discussed separately in tyLib, pipeDrv, nfsDrv, dosFsLib, rt11FsLib, and rawFsLib.

RETURNS

The return value of the driver, or ERROR if the fd does not exist.

SEE ALSO

ioLib, tyLib, pipeDrv, nfsDrv, dosFsLib, rt11FsLib, rawFsLib, *Programmer's Guide: I/O System, Local File Systems*

lseek( )

NAME

lseek( ) - set file read/write pointer

SYNOPSIS

```
int lseek (fd, offset, whence)
    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:

L_SET	- set to *offset*
L_INCR	- set to current position plus *offset*
L_XTND	- 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

ioDefPathSet( )

NAME

ioDefPathSet( ) - set the current default path

SYNOPSIS

```
STATUS ioDefPathSet (name)
    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*( )

ioDefPathGet( )

NAME

ioDefPathGet( ) - get the current default path

SYNOPSIS

```
VOID ioDefPathGet (pathname)
    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*( )

chdir( )

NAME

chdir( ) - set the current default path

SYNOPSIS

```
STATUS chdir (pathname)
    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*( )

getcwd( )

NAME

getcwd( ) - get the current default path

SYNOPSIS

```
char *getcwd (buffer, size)
    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 compatibility the POSIX specification.

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*( )

getwd( )

NAME

getwd( ) - get the current default path

SYNOPSIS

```
char *getwd (pathname)
    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

ioGlobalStdSet( )

NAME

ioGlobalStdSet( ) - set the fd for global standard input/output/error

SYNOPSIS

```
VOID ioGlobalStdSet (stdFd, newFd)
    int  stdFd;  /* standard input (0), output (1), or error (2) */
    int  newFd;  /* new underlying fd                            */
```

DESCRIPTION

This routine changes the assignment of a specified global standard fd *stdFd* (0, 1, or, 2) to the specified underlying fd *newFd*. *newFd* should be an fd 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 fd (see *ioTaskStdSet*( )). If *stdFd* is not 0, 1, or 2, this routine has no effect.

RETURNS

N/A

SEE ALSO

ioLib, *ioGlobalStdGet*( ), *ioTaskStdSet*( )

ioGlobalStdGet( )

NAME

ioGlobalStdGet( ) - get the fd for global standard input/output/error

SYNOPSIS

```
int ioGlobalStdGet (stdFd)
    int  stdFd;  /* standard input (0), output (1), or error (2) */
```

DESCRIPTION

This routine returns the current underlying fd for global standard input, output, and error.

RETURNS

The underlying global fd, or ERROR if *stdFd* is not 0, 1, or 2.

SEE ALSO

ioLib, *ioGlobalStdSet*( ), *ioTaskStdGet*( )

ioTaskStdSet( )

NAME

ioTaskStdSet( ) - set the fd for task standard input/output/error

SYNOPSIS

```
VOID ioTaskStdSet (taskId, stdFd, newFd)
    int  taskId;  /* id of task whose std fd is to be set (0 = self) */
    int  stdFd;   /* standard input (0), output (1), or error (2) */
    int  newFd;   /* new underlying fd                             */
```

DESCRIPTION

This routine changes the assignment of a specified task-specific standard fd *stdFd* (0, 1, or, 2) to the specified underlying fd *newFd*. *newFd* should be an fd 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.

This routine has no effect if it is called at interrupt level.

RETURNS

N/A

SEE ALSO

ioLib, *ioGlobalStdGet*( ), *ioTaskStdGet*( )

ioTaskStdGet( )

NAME

ioTaskStdGet( ) - get the fd for task standard input/output/error

SYNOPSIS

```
int ioTaskStdGet (taskId, stdFd)
    int  taskId;  /* id of desired task (0 = self) */
```

```
        int  stdFd;    /* standard input (0), output (1), or error (2) */
```

DESCRIPTION

This routine returns the current underlying fd for task-specific standard input, output, and error.

RETURNS

The underlying fd, or ERROR if *stdFd* is not 0, 1, or 2, or the routine is called at interrupt level.

SEE ALSO

ioLib, *ioGlobalStdGet*( ), *ioTaskStdSet*( )

iosLib

NAME

iosLib - I/O system

SYNOPSIS

iosInit( ) - initialize the I/O system
iosDrvInstall( ) - install an I/O driver
iosDrvRemove( ) - remove an I/O driver
iosDrvShow( ) - display a list of system drivers
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
iosDevShow( ) - display the list of devices in the system
iosFdShow( ) - display a list of fd names in the system
iosFdValue( ) - validate an open fd and return the driver-specific value

```
STATUS iosInit (max_drivers, max_files, nullDevName)
int iosDrvInstall (pCreate, pDelete, pOpen, pClose, pRead, pWrite, pIoctl)
STATUS iosDrvRemove (drvnum, forceClose)
VOID iosDrvShow ()
STATUS iosDevAdd (pDevHdr, name, drvnum)
VOID iosDevDelete (pDevHdr)
DEV_HDR *iosDevFind (name, pNameTail)
VOID iosDevShow ()
VOID iosFdShow ()
int iosFdValue (fd)
```

DESCRIPTION

This library is the driver-level interface to the Vx960 I/O system. Its primary purpose is to route user I/O requests to the proper drivers, using the proper parameters. To do this, it keeps tables about the available drivers (e.g., names, open files).

The I/O system should be initialized by calling *iosInit*( ) before 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 chapter "I/O System".

SEE ALSO

intLib, ioLib, *Programmer's Guide: I/O System*

iosInit( )

NAME

iosInit( ) - initialize the I/O system

SYNOPSIS

```
STATUS iosInit (max_drivers, max_files, nullDevName)
    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.

SEE ALSO

iosLib

iosDrvInstall( )

NAME

iosDrvInstall( ) - install an I/O driver

SYNOPSIS

```
int iosDrvInstall (pCreate, pDelete, pOpen, pClose, pRead, pWrite, pIoctl)
    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 (drvnum, forceClose)
    int   drvnum;      /* Number of the driver to remove.
                        * Returned by iosDrvInstall */
    BOOL  forceClose;  /* if TRUE, force closure of all open files */
```

DESCRIPTION

This routine removes an I/O driver from the driver table that was added by *iosDrvInstall*( ).

RETURNS

OK, or ERROR if the driver has open files.

SEE ALSO

iosLib

iosDrvShow( )

NAME

iosDrvShow( ) - display a list of system drivers

SYNOPSIS

```
VOID iosDrvShow ()
```

DESCRIPTION

This routine displays all the drivers in the driver list.

RETURNS

N/A

SEE ALSO

iosLib

iosDevAdd( )

NAME

iosDevAdd( ) - add a device to the I/O system

SYNOPSIS

```
STATUS iosDevAdd (pDevHdr, name, drvnum)
    DEV_HDR  *pDevHdr;  /* Pointer to the device's structure */
    char     *name;     /* Name of the device */
    int      drvnum;    /* Number of the servicing driver.
                         * Returned by iosDrvInstall */
```

DESCRIPTION

This routine adds a device to the I/O system device list, making the device available to subsequent *open*( ) and *creat*( ) calls.

The parameter *pDevHdr* is a pointer to a 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 (pDevHdr)
    DEV_HDR  *pDevHdr;  /* Pointer to the 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 fds open on the device or pending operations are unaffected.

If the device was never added to the device list, unpredictable results may occur.

SEE ALSO
iosLib

iosDevFind( )

NAME
iosDevFind( ) - find an I/O device in the device list

SYNOPSIS
```
DEV_HDR *iosDevFind (name, pNameTail)
    char  *name;        /* Name of the device */
    char  **pNameTail;  /* Where to return ptr to tail of the 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, and returns a pointer to the device. If the routine fails, it returns a pointer to the default device 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, pathLib

iosDevShow( )

NAME
iosDevShow( ) - display the list of devices in the system

SYNOPSIS
```
VOID iosDevShow ()
```

DESCRIPTION
This routine displays a list of all the devices in the device list.

RETURNS
N/A

SEE ALSO
iosLib, *devs*( )

iosFdShow( )

NAME
iosFdShow( ) - display a list of fd names in the system

SYNOPSIS

```
VOID iosFdShow ()
```

DESCRIPTION
This routine displays a list of all fds in the system.

RETURNS
N/A

SEE ALSO
iosLib, *ioctl*( )

iosFdValue( )

NAME
iosFdValue( ) - validate an open fd and return the driver-specific value

SYNOPSIS

```
int iosFdValue (fd)
    int  fd;  /* fd 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 fd is invalid.

SEE ALSO
iosLib

kernelLib

NAME

kernelLib - Vx960 kernel library

SYNOPSIS

kernelInit( ) - initialize the kernel
kernelVersion( ) - return the kernel revision string
kernelTimeSlice( ) - enable round-robin selection

```
VOID kernelInit (rootRtn, rootMemSize, pMemPoolStart, pMemPoolEnd, ...
char *kernelVersion ()
STATUS kernelTimeSlice (ticks)
```

DESCRIPTION

The Vx960 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 Vx960 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 Vx960.

(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. Vx960 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 Vx960 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 Vx960 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 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.

SEE ALSO

taskLib, **intLib**, *Programmer's Guide: Basic OS*

kernelInit( )

NAME

kernelInit( ) - initialize the kernel

SYNOPSIS

```
VOID kernelInit (rootRtn, rootMemSize, pMemPoolStart, pMemPoolEnd,
                             intStackSize, lockOutLevel)
    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 only be called once. The routine *rootRtn* is the user's start-up code that subsequently initializes system facilities (i.e., the I/O system, network). Typically, *rootRtn* is *usrRoot*( ).

Interrupts are enabled for the first time after *kernelInit*( ) exits. Vx960 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 will allocate a portion of memory starting at *pMemPoolStart* of *intStackSize* bytes for this purpose.

RETURNS

N/A

SEE ALSO

kernelLib, *intLockLevelSet*( )

kernelVersion( )

NAME

kernelVersion( ) - return the kernel revision string

SYNOPSIS

```
char *kernelVersion ()
```

DESCRIPTION

This routine returns a string which contains the current revision of the kernel. The string is of the form "Vx960 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 "Vx960 version x.y".

SEE ALSO

kernelLib

kernelTimeSlice( )

NAME

kernelTimeSlice( ) - enable round-robin selection

SYNOPSIS

```
STATUS kernelTimeSlice (ticks)
    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.

See the manual entry on kernelLib for more information on round-robin scheduling.

RETURNS

OK always.

SEE ALSO

kernelLib

ledLib

NAME

ledLib - line-editing library

SYNOPSIS

ledOpen( ) - create a new line-editor ID
ledClose( ) - discard the line-editor ID
ledRead( ) - read a line with line-editing
ledControl( ) - change the line-editor ID parameters

```
int ledOpen (inFd, outFd, histSize)
STATUS ledClose (led_id)
int ledRead (led_id, string, maxBytes)
VOID ledControl (led_id, inFd, outFd, histSize)
```

DESCRIPTION

This library provides a line-editing layer on top of a *tty* device. The shell uses this interface for its history-editing features.

The shell history mechanism is similar to the UNIX Korn shell history facility, with a built-in line-editor similar to UNIX **vi** that allows previously typed commands to be edited. The command *h*( ) displays the 20 most recent commands typed into the shell; old commands fall off the top as new ones are entered.

To edit a command, type ESC to enter edit mode, and use the commands listed below. The ESC key switches the shell to edit mode. The RETURN key always gives the line to the shell from either editing or input mode.

The following list is a summary of the commands available in edit mode.

Movement and search commands:

*n***G**	- Go to command number *n*.
/*s*	- Search for string *s* backward in history.
?*s*	- Search for string *s* forward in history.
n	- Repeat last search.
N	- Repeat last search in opposite direction.
*n***k**	- Get *n*th previous shell command in history.
*n***-**	- Same as "k".
*n***j**	- Get *n*th next shell command in history.
*n***+**	- Same as "j".
*n***h**	- Move left *n* characters.
^H	- Same as "h".

*n*l	- 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:

`^U`	- Delete line and leave edit mode.
`^L`	- Redraw line.
`^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, the library **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.

SEE ALSO

Programmer's Guide: Shell

ledOpen( )

NAME

ledOpen( ) - create a new line-editor ID

SYNOPSIS

```
int ledOpen (inFd, outFd, histSize)
    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

ledClose( )

NAME

ledClose( ) - discard the line-editor ID

SYNOPSIS

```
STATUS ledClose (led_id)
    int  led_id;  /* ID returned by ledOpen */
```

DESCRIPTION

This routine frees resources allocated by *ledOpen*( ). The low-level input/output fds are not closed.

SEE ALSO

ledLib

ledRead( )

NAME

ledRead( ) - read a line with line-editing

SYNOPSIS

```
int ledRead (led_id, string, maxBytes)
    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

ledControl( )

NAME

ledControl( ) - change the line-editor ID parameters

SYNOPSIS

```
VOID ledControl (led_id, inFd, outFd, histSize)
    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

loadLib

NAME

loadLib - object module loader

SYNOPSIS

loadModule( ) - load object module into memory
loadModuleAt( ) - load object module into memory

```
STATUS loadModule (fd, symFlag)
STATUS loadModuleAt (fd, symFlag, ppText, ppData, ppBss)
ldPrintSegInfo (pSeg)
```

DESCRIPTION

This library provides an object module loading facility. GNU/960 and b.out 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, anywhere in memory.

EXAMPLE

```
    fdX = open ("/devX/objFile", READ);
    loadModule (fdX, ALL_SYMBOLS);
    close (fdX);
```

This code fragment would load the b.out 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

Programmer's Guide: Architecture usrLib, symLib, memLib

loadModule( )

NAME

loadModule( ) - load object module into memory

SYNOPSIS

```
STATUS loadModule (fd, symFlag)
    int fd;            /* fd of file to load */
    int symFlag;       /* symbols to be added to table
                        * ([NO | GLOBAL | ALL]_SYMBOLS) */
```

DESCRIPTION

This routine loads an b.out format object module from the specified file, and placing the code, data and bss into memory allocated from the system memory pool.

See *loadModuleAt*( ) for more detail.

RETURNS

OK, or ERROR if the file cannot be read, there is not enough memory, or the file format is illegal.

SEE ALSO

loadLib, *loadModuleAt*( )

loadModuleAt( )

NAME

loadModuleAt( ) - load object module into memory

SYNOPSIS

```
STATUS loadModuleAt (fd, symFlag, ppText, ppData, ppBss)
    int fd;       /* fd from which to read module */
    int symFlag;  /* symbols to be added to table
                   *   ([NO | GLOBAL | ALL]_SYMBOLS) */
    INT8 **ppText; /* load text segment at adress pointed to by this
                    * pointer, return load address via this pointer */
    INT8 **ppData; /* load data segment at adress pointed to by this
                    * pointer, return load address via this pointer */
    INT8 **ppBss;  /* load bss segment at adress pointed to by this
                    * pointer, return load address via this pointer */
```

DESCRIPTION

This routine reads a b.out format object module from the specified fd, and loads the code, data and bss into memory at the specified load addresses, or into allocated memory as described below. The module is properly relocated as per 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 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 can not 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, ERROR will be returned after the module is loaded.

ADDING SYMBOLS TO THE SYMBOL TABLE

The symbols defined in the module being loaded may optionally be added to the system symbol table, depending on the value of *symFlag*:

NO_SYMBOLS	- add no symbols to the system symbol table
GLOBAL_SYMBOLS	- add external symbols only
ALL_SYMBOLS	- add all symbols

In addition, the following symbols are also added to the symbol table to indicate the 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 malloc'ed for the segment from the system memory pool. 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
ptr to LD_NO_ADDRESS	- no load address is specified, return address via ptr
ptr to address	- load address is specified

The ppText, ppData, and ppBss parameters are devious. For each one, if the pointer is NULL, the caller doesn't care where the segment gets loaded, and doesn't want to be told. If the pointer is not NULL, then the pointer points to a second pointer, which points to where the segment should be loaded. If that second pointer has the value LD_NO_ADDRESS, then the caller doesn't care where the segment gets loaded, but needs to be told where it was put. In that case, the LD_NO_ADDRESS gets replaced with a pointer to where the segment got loaded. (LD_NO_ADDRESS is used, rather than NULL, so that a segment may be loaded at the beginning of memory.)

When either don't care method is used, the corresponding segment is placed following the preceding segment (where the ordering of segments is text, data, bss). Thus, if ppText is don't care, ppData indicates an actual address, and ppBss is don't care, then space will be allocated for text, and bss will be placed following data. The caller is responsible for ensuring that the area indicated by ppData is large enough to contain both data and bss.

COMMON

Some host compiler/linker combinations internally use another storage class called "common". In C, uninitialized global variables are eventually put in the BSS segment. However, in partially linked object modules they are flagged internally as "common" and the linker resolves these and places them in BSS as a final step in creating a fully linked object module. However, the Vx960 loader is most often used to load object modules which are partially linked. When the Vx960 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 some 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 "-dr").

NOTE 80960CA

This function reads a b.out format file generated by the GNU/960 toolchain loader. (see b_out.h and GNU/960 Reference Manual). The b.out header fields are used as follows;

a_tload - text runtime load address

a_dload - data runtime load address

a_talign - Alignment of text segment

a_dalign - alingment of data segment

a_balign - alignment of bss segment

If tload and dload are NULL, talign and dalign are checked and the segments are located accordingly. If tload and dload are specified, it is the callers responsibilty to have them in aggreement with the given alignments. In effect, dalign and talign are ignored and only a_balign will be used to align the BSS segment accordingly. The BSS segment will be contiguous with the data segemnt, and padded by the appropriate number of bytes to make the alignment correct.

EXAMPLE

Load a module into allocated memory; do not return segment addresses:

```
status = loadModuleAt (fd, GLOBAL_SYMBOLS, NULL, NULL, NULL);
```

Load a module into allocated memory; return segment addresses:

```
pText = pData = pBss = LD_NO_ADDRESS;
status = loadModuleAt (fd, GLOBAL_SYMBOLS, &pText, &pData, &pBss);
```

Load a module at a specified address:

```
pText = 0x800000;                /* address of text segment */
pData = pBss = LD_NO_ADDRESS     /* other segments follow by default */
status = loadModuleAt (fd, GLOBAL_SYMBOLS, &pText, &pData, &pBss);
```

RETURNS

OK, or ERROR if the file cannot be read, there is not enough memory, or the file format is illegal.

SEE ALSO

loadLib, *Programmer's Guide: Architecture* GNU/960 Reference Manual

logLib

NAME

logLib - message logging library

SYNOPSIS

logInit( ) - initialize message logging library
logMsg( ) - log a formatted error message
logFdSet( ) - set the primary logging file descriptor
logFdAdd( ) - add a logging file descriptor
logFdDelete( ) - delete a logging file descriptor
logTask( ) - message-logging support task

```
STATUS logInit (fd, maxMsgs)
int logMsg (fmt, arg1, arg2, arg3, arg4, arg5, arg6)
VOID logFdSet (fd)
STATUS logFdAdd (fd)
STATUS logFdDelete (fd)
VOID logTask ()
```

DESCRIPTION

This module 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 routine logMsg 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 fds 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 any other routines in this module. This is done by the root task, *usrRoot*( ), in **usrConfig.c**.

SEE ALSO

fioLib, **pipeDrv**, *Programmer's Guide: I/O System*

logInit( )

NAME

logInit( ) - initialize message logging library

SYNOPSIS

```
STATUS logInit (fd, maxMsgs)
    int  fd;        /* file descriptor to use as logging device */
    int  maxMsgs;   /* maximum number of messages allowed in *
                     * the log queue */
```

DESCRIPTION

This routine specifies the fd 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

logMsg( )

NAME

logMsg( ) - log a formatted error message

SYNOPSIS

```
int logMsg (fmt, arg1, arg2, arg3, arg4, arg5, arg6)
    char  *fmt;  /* format string for print */
    int   arg1;  /* optional arguments for fmt */
    int   arg2;
    int   arg3;
    int   arg4;
    int   arg5;
    int   arg6;
```

DESCRIPTION

This routine logs the specified message via the logging task. Its syntax is identical to *printf*( ) — a format string is followed by arguments to be filled in the format. However, the *logMsg*( ) routine is restricted to a maximum of 6 arguments following the format string.

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).

See the manual entry for **logLib** for more detailed information about the use of *logMsg*( ).

EXAMPLE

If the following code were executed by task 20:

```
{
name = "GROMK";
num = 123;

logMsg ("ERROR - name = %s, num = %d.\n", name, num);
}
```

the following error message would appear on the system log:

```
t20: ERROR - name = GROMK, 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*( )

logFdSet( )

NAME

logFdSet( ) - set the primary logging file descriptor

SYNOPSIS

```
VOID logFdSet (fd)
    int  fd;  /* file descriptor to use as logging device */
```

DESCRIPTION

This routine changes the fd where messages from *logMsg*( ) are written, allowing the log device to be changed from the default specified by *logInit*( ). It first removes the old fd (if one had been previously set) from the log fd list, then adds the new *fd.*

The old logging fd 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*( )

logFdAdd( )

NAME

logFdAdd( ) - add a logging file descriptor

SYNOPSIS

```
STATUS logFdAdd (fd)
    int  fd;  /* file descriptor for additional logging device */
```

DESCRIPTION

This routine adds another fd to which messages will be logged to the log fd list. The fd must be a valid open fd.

RETURNS

OK, or ERROR if the allowable number of additional logging fds (5) is exceeded.

SEE ALSO

logLib

logFdDelete( )

NAME

logFdDelete( ) - delete a logging file descriptor

SYNOPSIS

```
STATUS logFdDelete (fd)
    int  fd;  /* file descriptor to stop using as logging device */
```

DESCRIPTION

This routine removes a logging fd added by *logFdAdd*( ) from the log *fd* list. The fd is not closed; it is simply no longer used by the logging facilities.

RETURNS

OK, or ERROR if the fd was not added with *logFdAdd*( ).

SEE ALSO

logLib, *logFdAdd*( )

logTask( )

NAME

logTask( ) - message-logging support task

SYNOPSIS

```
VOID logTask ()
```

DESCRIPTION

This task prints the messages logged with *logMsg*( ). It continually reads a pipe and prints any messages that come through the pipe in log format on the fd that was specified by ***logInit***( ) (or a subsequent call to *logFdSet*( ) or *logFdAdd*( )).

This task is spawned by ***logInit***( ).

RETURNS

N/A

SEE ALSO

logLib

loginLib

NAME

loginLib - user login/password subroutine library

SYNOPSIS

loginInit( ) - initialize the login table
loginUserAdd( ) - add a user to the login table
loginUserDelete( ) - delete a user entry from the login table
loginUserVerify( ) - verify a user name and password in the login table
loginUserShow( ) - display the user login table
loginPrompt( ) - display a login prompt and validate a user entry
loginStringSet( ) - change the login string
loginEncryptInstall( ) - install an encryption routine
loginDefaultEncrypt( ) - default password encryption routine

```
VOID loginInit ()
STATUS loginUserAdd (name, passwd)
STATUS loginUserDelete (name, passwd)  ...
STATUS loginUserVerify (name, passwd)
VOID loginUserShow ()
STATUS loginPrompt (userName)    ...
VOID loginStringSet (newString)
VOID loginEncryptInstall (rtn, var)
STATUS loginDefaultEncrypt (in, out)
```

DESCRIPTION

This library provides a login/password facility for network access to the Vx960 shell. When installed, it will require a user name and password match to gain access to the Vx960 shell from rlogin or telnet. This permits Vx960 to be used in secure environments that must restrict access to a Vx960 system.

A routine is provided to prompt for the user name and password, and to 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 remotely login to a Vx960 shell. 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 calling *loginInit*( ). The login feature must then be connected to the shell by calling *shellLoginInstall*( ).

```
loginInit ();                          /* initialize login table *
shellLoginInstall (loginPrompt, NULL); /* install security program *
```

If INCLUDE_SECURITY is defined in **configAll.h**, these routines are called by *usrRoot*( ) in usrConfig.c.

The name/password pairs are added to the table by calling *loginUserAdd*( ), which takes the name and an encrypted password as arguments. The Vx960 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 Vx960:

```
loginUserAdd ("fred", "ScebRezb9c");
```

This can be done from the shell, a start-up script, or application code.

In the standard Vx960 configuration, the login security feature is included in a Vx960 system by defining INCLUDE_SECURITY in configAll.h. This will enable the calls to *loginInit*( ) and *shellLoginInstall*( ), in the Vx960 configuration module usrConfig.c. It also adds a single default user to the login table. The default user and password are defined in **configAll.h** as LOGIN_USER_NAME and LOGIN_PASSWORD. These can be set to any desired name and password. Additional users can be added by adding additional calls to *loginUserAdd*( ).

LOGGING IN

When the login security facility is installed, every attempt to rlogin or telnet to the Vx960 shell will first prompt for user name and password.

```
% rlogin target

Vx960 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

loginInit( )

NAME

loginInit( ) - initialize the login table

SYNOPSIS

```
VOID loginInit ()
```

DESCRIPTION

This routine must be called to initialize the login data structure used by routines throughout this module. If INCLUDE_SECURITY is defined in configAll.h, it is called by *usrRoot*( ) in usrConfig.c, before any other routines in this module.

RETURNS

N/A

SEE ALSO

loginLib

loginUserAdd( )

NAME

loginUserAdd( ) - add a user to the login table

SYNOPSIS

```
STATUS loginUserAdd (name, passwd)
    char  name[MAX_LOGIN_NAME_LEN+1];  /* user name */
    char  passwd[80];                  /* user password */
```

DESCRIPTION

This routine adds a user name and password entry to the login table.

The length of user names should not exceed MAX_LOGIN_NAME_LEN, while the length of passwords depends on the encryption routine used. For the default encryption routine, passwords should be at least 8 characters long and no more than 40 characters.

The procedure for adding a new user to login table is as follows:

(1) Generate the encrypted password by invoking **vxencrypt** in **vw/bin.**

(2) Add a user by invoking *loginUserAdd*( ) in the Vx960 shell with the user name and the encrypted password.

The password of a user can be changed by first deleting the user entry and 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 (name, passwd)
    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 supplied to remove an entry from the login table.

RETURNS

OK, or ERROR if there is no such user or the password is incorrect.

SEE ALSO

loginLib

loginUserVerify( )

NAME

loginUserVerify( ) - verify a user name and password in the login table

SYNOPSIS

```
STATUS loginUserVerify (name, passwd)
    char  *name;    /* name of user */
    char  *passwd;  /* password of user */
```

RETURNS

OK, or ERROR if the user name or password is not found.

SEE ALSO

loginLib

loginUserShow( )

NAME

loginUserShow( ) - display the user login table

SYNOPSIS

```
VOID loginUserShow ()
```

DESCRIPTION

This routine displays valid user names.

EXAMPLE

```
    -> loginUserShow ()

      User Name
```

```
          =========
          peter
          robin
       value = 0 = 0x0
       ->
```

RETURNS

N/A

SEE ALSO

loginLib

loginPrompt( )

NAME

loginPrompt( ) - display a login prompt and validate a user entry

SYNOPSIS

```
STATUS loginPrompt (userName)
    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 Vx960 system. Otherwise, it will prompt the user again.

All control characters are disabled during authentication except ^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 (newString)
    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

loginEncryptInstall( )

NAME

loginEncryptInstall( ) - install an encryption routine

SYNOPSIS

```
VOID loginEncryptInstall (rtn, var)
    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 (see the manual entry for *loginDefaultEncrypt*( )):

```
STATUS encryptRoutine (password, encryptedPassword)
       char *password;                /* string to encrypt *
       char *encryptedPassword;       /* resulting encryption *
```

When a custom encryption routine is installed, a UNIX version of this routine should also replace the tool vxencrypt in vw/bin. When a user logs in to the system, *rtn* is invoked to encrypt the requested password. The encrypted password is then compared against the encryption that was generated by the custom version of vxencrypt and added to Vx960 via *loginUserAdd*( ).

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 encryption
                                                        routine *
#endif
```

RETURNS

N/A

SEE ALSO

loginLib, *loginDefaultEncrypt*( ), vxencrypt

loginDefaultEncrypt( )

NAME

loginDefaultEncrypt( ) - default password encryption routine

SYNOPSIS

```
STATUS loginDefaultEncrypt (in, out)
    char  *in;   /* input string */
    char  *out;  /* encrypted string */
```

DESCRIPTION

This is the default encryption routine. It employs a simple encryption algorithm. As arguments, it takes 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 vw/bin should also be replaced by a UNIX 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**

lstLib

NAME

lstLib - doubly linked list subroutine library

SYNOPSIS

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

```
VOID lstInit (pList)
VOID lstAdd (pList, pNode)
VOID lstConcat (pDstList, pAddList)
int lstCount (pList)
VOID lstDelete (pList, pNode)
VOID lstExtract (pSrcList, pStartNode, pEndNode, pDstList)
NODE *lstFirst (pList)
NODE *lstGet (pList)
VOID lstInsert (pList, pPrev, pNode)
NODE *lstLast (pList)
NODE *lstNext (pNode)
NODE *lstNth (pList, nodenum)
NODE *lstPrevious (pNode)
NODE *lstNStep (pNode, nStep)
int lstFind (pList, pNode)
VOID lstFree (pList)
```

DESCRIPTION

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

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

NON-EMPTY LIST

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

EMPTY LIST

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

INCLUDE FILE

lstLib.h

lstInit( )

NAME

lstInit( ) - initialize a list descriptor

SYNOPSIS

```
VOID lstInit (pList)
    LIST  *pList;  /* pointer to list descriptor to be initialized */
```

DESCRIPTION

This routine initializes a specified list to an empty list.

RETURNS

N/A

SEE ALSO

lstLib

lstAdd( )

NAME

lstAdd( ) - add a node to the end of a list

SYNOPSIS

```
VOID lstAdd (pList, pNode)
    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 (pDstList, pAddList)
    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 (pList)
    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 (pList, pNode)
    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 (pSrcList, pStartNode, pEndNode, pDstList)
    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;     /* pointer 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

lstFirst( )

NAME

lstFirst( ) - find first node in list

SYNOPSIS

```
NODE *lstFirst (pList)
    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

lstGet( )

NAME

lstGet( ) - delete and return the first node from a list

SYNOPSIS

```
NODE *lstGet (pList)
    LIST  *pList;  /* pointer 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

lstInsert( )

NAME

lstInsert( ) - insert a node in a list after a specified node

SYNOPSIS

```
VOID lstInsert (pList, pPrev, pNode)
    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 (pList)
    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 (pNode)
    NODE  *pNode;  /* pointer 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

lstNth( )

NAME

lstNth( ) - find the Nth node in a list

SYNOPSIS

```
NODE *lstNth (pList, nodenum)
    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. 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 (pNode)
    NODE  *pNode;  /* pointer 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

lstNStep( )

NAME

lstNStep( ) - find a list node *nStep* nodes away from a specified node

SYNOPSIS

```
NODE *lstNStep (pNode, nStep)
    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, a NULL is returned.

RETURNS

A pointer to the node *nStep* steps away, or NULL if the node is out of range.

SEE ALSO

lstLib

lstFind( )

NAME

lstFind( ) - find a node in a list

SYNOPSIS

```
int lstFind (pList, pNode)
    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

lstFree( )

NAME

lstFree( ) - free up a list

SYNOPSIS

```
VOID lstFree (pList)
    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*( )

memLib

NAME

memLib - the Vx960 memory manager

SYNOPSIS

memPartCreate( ) - create a memory partition
memPartAddToPool( ) - add memory to a memory partition
memPartOptionsSet( ) - set the debug options for a memory partition
memPartAlloc( ) - allocate a block of memory from a specified partition
memPartRealloc( ) - reallocate a block of memory in a specified partition
memPartFree( ) - free a block of memory in a specified partition
memPartFindMax( ) - find the size of the largest available free block
memPartShow( ) - show the partition blocks and statistics
memAddToPool( ) - add memory to the system memory partition
memOptionsSet( ) - set the debug options for the system memory partition
malloc( ) - allocate a block of memory from the system memory partition
calloc( ) - allocate space for an array
realloc( ) - reallocate a block of memory
free( ) - free a block of memory
cfree( ) - free a block of memory
memFindMax( ) - find the largest free block in the system memory partition
memShow( ) - show the system memory partition blocks and statistics

```
PART_ID memPartCreate (pPool, poolSize)
STATUS memPartAddToPool (partId, pPool, poolSize)
STATUS memPartOptionsSet (partId, options)
VOID *memPartAlloc (partId, nBytes)
VOID *memPartRealloc (partId, pBlock, nBytes)
STATUS memPartFree (partId, pBlock)
int memPartFindMax (partId)
STATUS memPartShow (partId, type)
VOID memAddToPool (pPool, poolSize)
VOID memOptionsSet (options)
VOID *malloc (nBytes)
VOID *calloc (elemNum, elemSize)
VOID *realloc (pBlock, newSize)
STATUS free (pBlock)
STATUS cfree (pBlock)
int memFindMax ()
VOID memShow (type)
```

DESCRIPTION

This library provides facilities for managing the allocation of blocks of memory from ranges of memory called memory partitions. This library consists of two sets of routines. The first set, *memPart...*( ), comprises a general facility for the creation and deletion of memory partitions, and the allocation and deallocation of blocks from those partitions. The second set provides a traditional C *malloc*( )/*free*( ) interface to one particular partition, 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 *memPartAlloc*( ) in the general case and *malloc*( ) for the system memory partition, is done with a first-fit algorithm. Adjacent blocks of memory are coalesced when they are freed, using *memPartFree*( ) and *free*( ).

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, 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 *free*( ) will make a consistency check of various pointers and values in the header of the block being freed.

MEM_BLOCK_CHECK
- check each block freed.

One of the following options can be specified to determine the action to be taken when a bad block is detected when freed. These options apply only if the MEM_BLOCK_CHECK option is selected.

MEM_BLOCK_ERROR_RETURN
- just return the status to the calling task.

MEM_BLOCK_ERROR_LOG_MSG
- log an error message and return the status to the calling task.

MEM_BLOCK_ERROR_LOG_AND_SUSPEND
- log an error message and suspend the calling task.

The default options when a partition is created are:

MEM_ALLOC_ERROR_LOG_MSG
MEM_BLOCK_CHECK
MEM_BLOCK_ERROR_LOG_AND_SUSPEND

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_MSG |
                             MEM_BLOCK_CHECK |
                             MEM_BLOCK_ERROR_LOG_MSG);
```

CAVEATS

The routine *malloc*( ) always rounds up to a 4-byte boundary and there is an 8-byte overhead for every block allocated.

INCLUDE FILE

memLib.h

memPartCreate( )

NAME

memPartCreate( ) - create a memory partition

SYNOPSIS

```
PART_ID memPartCreate (pPool, poolSize)
    char      *pPool;    /* pointer to memory area */
    unsigned  poolSize;  /* size in bytes */
```

DESCRIPTION

This routine creates a new memory partition containing the 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

memLib

memPartAddToPool( )

NAME

memPartAddToPool( ) - add memory to a memory partition

SYNOPSIS

```
STATUS memPartAddToPool (partId, pPool, poolSize)
    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 memory partition after the initial call to *memPartCreate*( ). The memory added need not be contiguous with memory previously assigned to the partition.

RETURNS

OK or ERROR.

SEE ALSO

memLib, *memPartCreate*( )

memPartOptionsSet( )

NAME

memPartOptionsSet( ) - set the debug options for a memory partition

SYNOPSIS

```
STATUS memPartOptionsSet (partId, options)
    PART_ID   partId;   /* partition for which to set option */
    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 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

OK or ERROR.

SEE ALSO

memLib

memPartAlloc( )

NAME

memPartAlloc( ) - allocate a block of memory from a specified partition

SYNOPSIS

```
VOID *memPartAlloc (partId, nBytes)
    PART_ID   partId;  /* memory partition to allocate out of */
    unsigned  nBytes;  /* number of bytes to allocate */
```

DESCRIPTION

From a specified partition, this routine allocates a block of memory whose size is equal to or greater than *nBytes*. The partition must have been previously initialized with *memPartInit*( ).

RETURNS

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

SEE ALSO

memLib, *memPartInit*( )

memPartRealloc( )

NAME

memPartRealloc( ) - reallocate a block of memory in a specified partition

SYNOPSIS

```
VOID *memPartRealloc (partId, pBlock, nBytes)
    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 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.

RETURNS

A pointer to the new block of memory, or NULL if the call fails.

SEE ALSO

memLib

memPartFree( )

NAME

memPartFree( ) - free a block of memory in a specified partition

SYNOPSIS

```
STATUS memPartFree (partId, pBlock)
    PART_ID partId;   /* memory partition to add the block to */
    char    *pBlock;  /* pointer to block of memory to be freed */
```

DESCRIPTION

This routine takes a block of memory previously allocated with *memPartAlloc*( ) and returns it to the partition's free memory list.

RETURNS

OK or ERROR if there is an invalid block.

SEE ALSO

memLib

memPartFindMax( )

NAME

memPartFindMax( ) - find the size of the largest available free block

SYNOPSIS

```
int memPartFindMax (partId)
    PART_ID  partId;  /* partition ID */
```

DESCRIPTION

This routine searches for the largest block in the memory partition free list, and returns its size.

RETURNS

The size (in bytes) of the largest available block.

SEE ALSO

memLib

memPartShow( )

NAME

memPartShow( ) - show the partition blocks and statistics

SYNOPSIS

```
STATUS memPartShow (partId, type)
    PART_ID  partId;  /* partition ID */
    int      type;    /* 0 = statistics, 1 = statistics & list */
```

DESCRIPTION

For a specified partition, this routine displays the total amount of free space in the partition, 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.

In addition, if *type* is 1, this routine displays a list of all the blocks in the free list of the specified partition.

RETURNS

OK or ERROR.

SEE ALSO

memLib, *memShow*( )

memAddToPool( )

NAME

memAddToPool( ) - add memory to the system memory partition

SYNOPSIS

```
VOID memAddToPool (pPool, poolSize)
    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

memLib, *memPartAddToPool*( )

memOptionsSet( )

NAME

memOptionsSet( ) - set the debug options for the system memory partition

SYNOPSIS

```
VOID memOptionsSet (options)
    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*( )

malloc( )

NAME

malloc( ) - allocate a block of memory from the system memory partition

SYNOPSIS

```
VOID *malloc (nBytes)
    unsigned  nBytes;  /* number of bytes to allocate */
```

DESCRIPTION

This routine allocates a block of memory from the free list whose size is equal to or greater than *nBytes*.

RETURNS

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

SEE ALSO

memLib

calloc( )

NAME

calloc( ) - allocate space for an array

SYNOPSIS

```
VOID *calloc (elemNum, elemSize)
    unsigned  elemNum;   /* number of elements */
    unsigned  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 calls fails.

SEE ALSO

memLib

realloc( )

NAME

realloc( ) - reallocate a block of memory

SYNOPSIS

```
VOID *realloc (pBlock, newSize)
    char      *pBlock;  /* block to be reallocated */
    unsigned  newSize;  /* new block size */
```

DESCRIPTION

This routine changes the size of a given block 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.

RETURNS

A pointer to the new block of memory, or NULL if the call fails.

SEE ALSO

memLib

free( )

NAME

free( ) - free a block of memory

SYNOPSIS

```
STATUS free (pBlock)
    char  *pBlock;  /* pointer to block of memory to be freed */
```

DESCRIPTION

This routine takes a block of memory previously allocated with *malloc( )* or *calloc( )* and returns it to the free memory pool.

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

RETURNS

OK, or ERROR if the the block is invalid.

SEE ALSO

memLib

cfree( )

NAME

cfree( ) - free a block of memory

SYNOPSIS

```
STATUS cfree (pBlock)
    char  *pBlock;  /* pointer to block of memory to be freed */
```

DESCRIPTION

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

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

memFindMax( )

NAME

memFindMax( ) - find the largest free block in the system memory partition

SYNOPSIS

```
int memFindMax ()
```

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*( )

memShow( )

NAME

memShow( ) - show the system memory partition blocks and statistics

SYNOPSIS

```
VOID memShow (type)
    int  type;
```

DESCRIPTION

This routine displays the total amount of free space in the system memory partition, 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, a list of all the blocks in the free list of the system partition is displayed.

EXAMPLE

```
-> memShow 1

FREE LIST:
  num     addr       size
  --- ---------- ----------
    1  0x3fee18         16
    2  0x3b1434         20
    3   0x4d188    2909400

SUMMARY:
 status   bytes    blocks   ave block  max block
 ------ --------- -------- ---------- ----------
current
   free   2909436        3     969812    2909400
  alloc    969060    16102         60          -
cumulative
  alloc   1143340    16365         69          -
value = 12288 = 0x3000 = cFwd + 0x20
```

RETURNS

N/A

SEE ALSO

memLib, *memPartShow*( )

msgQLib

NAME

msgQLib - message queue library

SYNOPSIS

msgQCreate( ) - create and initialize a message queue
msgQDelete( ) - delete a message queue
msgQSend( ) - send a message to a message queue
msgQReceive( ) - receive a message from a message queue
msgQNumMsgs( ) - get the number of messages queued to a message queue
msgQInfoGet( ) - get information about a message queue
msgQShow( ) - show information about a message queue

```
MSG_Q_ID msgQCreate (maxMsgs, maxMsgLength, options)
STATUS msgQDelete (msgQId)
STATUS msgQSend (msgQId, buffer, nBytes, timeout, priority)
int msgQReceive (msgQId, buffer, maxNBytes, timeout)
int msgQNumMsgs (msgQId)
STATUS msgQInfoGet (msgQId, pInfo)
STATUS msgQShow (msgQId, level)
```

DESCRIPTION

In Vx960, message queues are 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 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 require 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 will be allowed to 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 value of the timeout parameter may have the special values of NO_WAIT (0) meaning to always return immediately or WAIT_FOREVER (-1) meaning to never time out.

URGENT MESSAGES

The *msgQSend*( ) function 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.

SEE ALSO

pipeDrv, *Programmer's Guide: Basic OS*

msgQCreate( )

NAME

msgQCreate( ) - create and initialize a message queue

SYNOPSIS

```
MSG_Q_ID msgQCreate (maxMsgs, maxMsgLength, options)
    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 of up to *maxMsgLength* bytes long. It returns a message queue ID used to identify the created message queue in all subsequent calls to routines in this library. The queue can be created with the following options:

MSG_Q_FIFO	- queue pended tasks in FIFO order.
MSG_Q_PRIORITY	- queue pended tasks in priority order.

RETURNS

MSG_Q_ID, or NULL if error.

ERRNO

S_memLib_NOT_ENOUGH_MEMORY
unable to allocate memory for message queue and message buffers.

S_intLib_NOT_ISR_CALLABLE
called from an interrupt service routine.

SEE ALSO

msgQLib

msgQDelete( )

NAME

msgQDelete( ) - delete a message queue

SYNOPSIS

```
STATUS msgQDelete (msgQId)
    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. Subsequently, *msgQId* will no longer be a valid message queue ID.

RETURNS

OK, or ERROR.

ERRNO

S_objLib_OBJ_ID_ERROR
the *msgQId* is invalid.

S_intLib_NOT_ISR_CALLABLE
called from an interrupt service routine.

SEE ALSO

msgQLib

msgQSend( )

NAME

msgQSend( ) - send a message to a message queue

SYNOPSIS

```
STATUS msgQSend (msgQId, buffer, nBytes, timeout, priority)
    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. *timeout* can have the special value NO_WAIT that indicates *msgQSend( )* will return immediately even if the message has not been sent. Another value for *timeout* is WAIT_FOREVER that indicates that *msgQSend( )* will never timeout.

The *priority* parameter specifies the priority of the message being sent. Normal priority messages (MSG_PRI_NORMAL) are added to the tail of the list of queued messages. Urgent priority messages (MSG_PRI_URGENT) are added to the head of the list.

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 specified as NO_WAIT.

RETURNS

OK, or ERROR.

ERRNO

S_objLib_OBJ_ID_ERROR
  the *msgQId* is invalid.

S_objLib_OBJ_DELETED
  the message queue was deleted while waiting to a send message.

S_objLib_OBJ_UNAVAILABLE
timeout is set to NO_WAIT and the queue is full.

S_objLib_OBJ_TIMEOUT
the queue is full for *timeout* ticks.

S_msgQLib_INVALID_MSG_LENGTH
nBytes is larger than the *maxMsgLength* set for the message queue.

S_msgQLib_NON_ZERO_TIMEOUT_AT_INT_LEVEL
called from an ISR with *timeout* not set to NO_WAIT.

SEE ALSO

msgQLib

msgQReceive( )

NAME

msgQReceive( ) - receive a message from a message queue

SYNOPSIS

```
int msgQReceive (msgQId, buffer, maxNBytes, timeout)
   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. *timeout* can have the special value NO_WAIT that indicates *msgQReceive( )* will return immediately even if a message is available. Another value for *timeout* is WAIT_FOREVER that indicates that *msgQReceive( )* will never timeout.

This routine *must not* be called by interrupt service routines.

RETURNS

The number of bytes copied to *buffer*, or ERROR.

ERRNO

S_objLib_OBJ_ID_ERP.OR
the *msgQId* is invalid.

S_objLib_OBJ_DELETED
the message queue was deleted while waiting to receive a message.

S_objLib_OBJ_UNAVAILABLE
timeout is set to NO_WAIT and no messages are available.

S_objLib_OBJ_TIMEOUT
no messages were received in *timeout* ticks.

S_msgQLib_INVALID_MSG_LENGTH
nBytes is less than 0.

SEE ALSO

msgQLib

msgQNumMsgs( )

NAME

msgQNumMsgs( ) - get the number of messages queued to a message queue

SYNOPSIS

```
int msgQNumMsgs (msgQId)
    MSG_Q_ID  msgQId;  /* message queue to examine */
```

DESCRIPTION

This routine returns the number of messages currently queued to a specified message queue.

RETURNS

The number of messages queued, or ERROR.

ERRNO

S_objLib_OBJ_ID_ERROR
the *msgQId* is invalid.

SEE ALSO

msgQLib

msgQInfoGet( )

NAME

msgQInfoGet( ) - get information about a message queue

SYNOPSIS

```
STATUS msgQInfoGet (msgQId, pInfo)
    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*, *numTasks* will be 0.

A list of pointers to the messages queued and their lengths can be obtained by setting *msgPtrList* and *msgLenList* to the addresses of arrays to receive the respective lists, and setting *msgListMax* to the maximum number of elements in those arrays. If either list pointer is NULL, no data will be returned for that array.

No more than *msgListMax* message pointers and lengths are returned, although *numMsgs* will always be returned with the actual number of messages queued.

For example, if the caller supplies a *msgPtrList* and *msgLenList* with room for 10 messages and sets *msgListMax* to 10, but there are 20 messages queued, then the pointers and lengths of the first 10 messages in the queue are returned in *msgPtrList* and *msgLenList*, but *numMsgs* will be returned with the value 20.

A list of the task IDs of tasks blocked on the message queue can be obtained by setting *taskIdList* to the address of an array to receive the list, and setting *taskIdListMax* to the maximum number of elements in that array. If *taskIdList* is NULL, then no task IDs are returned. No more than *taskIdListMax* task IDs are returned, although *numTasks* will always be returned with the actual number of tasks blocked.

For example, if the caller supplies a *taskIdList* with room for 10 task IDs and sets *taskIdListMax* to 10, but there are 20 tasks blocked on the message queue, then the IDs of the first 10 tasks in the blocked queue will be returned in *taskIdList*, but *numTasks* will be returned with the value 20.

Note that the tasks returned in *taskIdList* may be blocked for either send or receive. As noted above this can be determined by examining *numMsgs*.

The variables *sendTimeouts* and *recvTimeouts* are the counts of the number of times *msgQSend*( ) and *msgQReceive*( ) respectively returned with a timeout.

The variables *options*, *maxMsgs*, and *maxMsgLength* are the parameters with which the message queue was created.

WARNING

The information returned by this routine is not static and may be obsolete by the time it is examined. In particular, the lists of task IDs and/or message pointers may no longer be valid. However, the information is obtained atomically, thus it will be an accurate snapshot of the state of the message queue at the time of the call. This information is generally used for debugging purposes only.

WARNING

The current implementation of this routine locks out interrupts while obtaining the information. This can compromise the overall interrupt latency of the system. Generally this routine is used for debugging purposes only.

RETURNS

OK, or ERROR.

ERRNO

S_objLib_OBJ_ID_ERROR
  the *msgQId* is invalid.

SEE ALSO

msgQLib

msgQShow( )

NAME

msgQShow( ) - show information about a message queue

SYNOPSIS

```
STATUS msgQShow (msgQId, level)
    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 0x12345678:
  PRIORITY Task queuing
       100 Message length max (bytes)
        10 Messages max
         0 Messages queued
         0 Receivers blocked
  12345678 Send timeouts
  12345678 Receive timeouts
```

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:
  0x3e134 (tExcTask)
```

SEE ALSO

msgQLib

netLib

NAME

netLib - network interface library

SYNOPSIS

netLibInit( ) - initialize the network package
netTask( ) - network task entry point

```
STATUS netLibInit ()
VOID netTask ()
```

DESCRIPTION

This module 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 done at task level.

The routine *netHelp*( ) in **usrLib** displays a summary of the network facilities available from the Vx960 shell.

SEE ALSO

routeLib, **hostLib**, **netDrv**, *netHelp*( ), *Programmer's Guide: Network*

netLibInit( )

NAME

netLibInit( ) - initialize the network package

SYNOPSIS

```
STATUS netLibInit ()
```

DESCRIPTION

This routine installs the socket driver, creates the network task job queue, and spawns the network task *netTask*( ). It should be called once to initialize the network. If INCLUDE_NETWORK is defined in **configAll.h**, this is done by *usrRoot*( ) in **usrConfig.c**.

RETURNS

OK, or ERROR if network support cannot be initialized.

SEE ALSO

netLib, **usrConfig**

netTask( )

NAME

netTask( ) - network task entry point

SYNOPSIS

```
VOID netTask ()
```

DESCRIPTION

This routine is the Vx960 network support task. Most of the Vx960 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

netLib

netShow

NAME

netShow - network related information display routines

SYNOPSIS

ifShow( ) - display the attached network interfaces
icmpstatShow( ) - display statistics for ICMP
inetstatShow( ) - display all active connections for Internet protocol sockets
ipstatShow( ) - display IP statistics
mbufShow( ) - report mbuf statistics
netShowInit( ) - initialize network show routines
tcpstatShow( ) - display all statistics for the TCP protocol
udpstatShow( ) - display statistics for the UDP protocol
arptabShow( ) - display the known ARP entries

```
VOID ifShow (ifName)
VOID icmpstatShow ()
VOID inetstatShow ()
VOID ipstatShow (zero)
VOID mbufShow ()
VOID netShowInit ()  ...
VOID tcpstatShow ()
VOID udpstatShow ()
VOID arptabShow ()
```

DESCRIPTION

This library provides routines to show various network-related statistics, such as configuration parameters for network interfaces, protocol statistics, socket statistics, and so forth.

SEE ALSO

ifLib, *Programmer's Guide: Network* routeShow, hostShow

ifShow( )

NAME

ifShow( ) - display the attached network interfaces

SYNOPSIS

```
VOID ifShow (ifName)
    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 Internet address, point-to-point peer address (if using SLIP), broadcast address, netmask, subnet mask, Ethernet address, route metric, maximum transfer unit, number of packets sent and received on this interface, number of input and output errors, and flags (e.g., loopback, point-to-point, broadcast, promiscuous, arp, running, debug) are shown.

Note that *routeShow* can also be used to determine the number of interfaces.

EXAMPLE

The following call displays all interfaces whose names begin with "ei", (e.g., "ei0", "ei1", "ei2", etc.):

```
-> ifShow "ei"
```

The following call displays just the interface "ei0":

```
-> ifShow "ei0"
```

RETURNS

N/A

SEE ALSO

netShow, *routeShow*( ), ifLib, *Programmer's Guide: Network*

icmpstatShow( )

NAME

icmpstatShow( ) - display statistics for ICMP

SYNOPSIS

```
VOID icmpstatShow ()
```

DESCRIPTION

This routine displays statistics for the ICMP protocol.

RETURNS

N/A

SEE ALSO

netShow, *Programmer's Guide: Network*

inetstatShow( )

NAME

inetstatShow( ) - display all active connections for Internet protocol sockets

SYNOPSIS

```
VOID inetstatShow ()
```

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, *Programmer's Guide: Network*

ipstatShow( )

NAME

ipstatShow( ) - display IP statistics

SYNOPSIS

```
VOID ipstatShow (zero)
    BOOL  zero;  /* TRUE = reset statistics to 0 */
```

DESCRIPTION

This routine displays detailed statistics for the IP protocol.

RETURNS

N/A

SEE ALSO

netShow

mbufShow( )

NAME

mbufShow( ) - report mbuf statistics

SYNOPSIS

```
VOID mbufShow ()
```

DESCRIPTION

This routine displays the distribution of mbufs in the network.

RETURNS

N/A

SEE ALSO

netShow

netShowInit( )

NAME

netShowInit( ) - initialize network show routines

SYNOPSIS

```
VOID netShowInit ()
```

DESCRIPTION

This routine links the network show routines into the Vx960 system. These routines are included automatically by defining INCLUDE_NETWORK_SHOW in **configAll.h**.

RETURNS

N/A

SEE ALSO

netShow

tcpstatShow( )

NAME

tcpstatShow( ) - display all statistics for the TCP protocol

SYNOPSIS

```
VOID tcpstatShow ()
```

DESCRIPTION

This routine displays detailed statistics for the TCP protocol.

RETURNS

N/A

SEE ALSO

netShow, *Programmer's Guide: Network*

udpstatShow( )

NAME

udpstatShow( ) - display statistics for the UDP protocol

SYNOPSIS

```
VOID udpstatShow ()
```

DESCRIPTION

This routine displays statistics for the UDP protocol.

RETURNS

N/A

SEE ALSO

netShow

arptabShow( )

NAME

arptabShow( ) - display the known ARP entries

SYNOPSIS

```
VOID arptabShow ()
```

DESCRIPTION

This routine displays current IP-to-Ethernet mappings in the ARP table.

RETURNS

N/A

SEE ALSO

netShow

nfsLib

NAME

nfsLib - Network File System library

SYNOPSIS

nfsHelp( ) - display the NFS help menu
nfsExportShow( ) - display the exported file systems of a remote host
nfsAuthUnixPrompt( ) - modify the NFS UNIX authentication parameters
nfsAuthUnixShow( ) - display the NFS UNIX authentication parameters
nfsAuthUnixSet( ) - set the NFS UNIX authentication parameters
nfsAuthUnixGet( ) - get the NFS UNIX authentication parameters
nfsIdSet( ) - set the ID number of the NFS UNIX authentication parameters

```
VOID nfsHelp ()
STATUS nfsExportShow (hostName)
VOID nfsAuthUnixPrompt ()
VOID nfsAuthUnixShow ()
VOID nfsAuthUnixSet (machname, uid, gid, ngids, aup_gids)
VOID nfsAuthUnixGet (machname, pUid, pGid, pNgids, gids)
VOID nfsIdSet (uid)
```

DESCRIPTION

This library provides the client side of services for Network File System (NFS) 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.

Vx960 is delivered with NFS enabled. NFS is enabled by defining the constant INCLUDE_NFS in the Vx960 configuration header file **config/all/configAll.h**:

```
#define INCLUDE_NFS
```

In the same file, the parameters NFS_USER_ID and NFS_GROUP_ID should be defined to set the default user ID and group ID at system start-up. See the "Network" chapter in the *Programmer's Guide* for information on creating NFS devices.

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 that the user belongs to. On UNIX systems, a user ID is specified in the file /etc/passwd. The list of groups that a user belongs to 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.

TASK STACK SIZE

In previous versions of Vx960, the recursive nature of the underlying XDR (eXternal Data Representation) routines for RPC required large task stacks (> 7000 bytes). In Vx960 5.0, these stack requirements have been reduced considerably. Normal use of NFS requires no more than 2000 bytes of stack.

INCLUDE FILE

nfsLib.h

SEE ALSO

rpcLib, ioLib, nfsDrv, *Programmer's Guide: Network*

nfsHelp( )

NAME

nfsHelp( ) - display the NFS help menu

SYNOPSIS

```
VOID nfsHelp ()
```

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/ *
```

SEE ALSO

nfsLib

nfsExportShow( )

NAME

nfsExportShow( ) - display the exported file systems of a remote host

SYNOPSIS

```
STATUS nfsExportShow (hostName)
    char  *hostName;  /* host machine for which to show exports */
```

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

nfsAuthUnixPrompt( )

NAME

nfsAuthUnixPrompt( ) - modify the NFS UNIX authentication parameters

SYNOPSIS

```
VOID nfsAuthUnixPrompt ()
```

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, *nfsAuthUnixGet( )*, *nfsAuthUnixSet( )*, *nfsAuthUnixShow( )*, *nfsIdSet( )*

nfsAuthUnixShow( )

NAME

nfsAuthUnixShow( ) - display the NFS UNIX authentication parameters

SYNOPSIS

```
VOID nfsAuthUnixShow ()
```

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
```

SEE ALSO

nfsLib, *nfsAuthUnixGet*( ), *nfsAuthUnixSet*( ), *nfsAuthUnixPrompt*( ), *nfsIdSet*( )

nfsAuthUnixSet( )

NAME

nfsAuthUnixSet( ) - set the NFS UNIX authentication parameters

SYNOPSIS

```
VOID nfsAuthUnixSet (machname, uid, gid, ngids, aup_gids)
    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 in *usrNetInit*( ) in usrConfig.

SEE ALSO

nfsLib, usrConfig, *nfsAuthUnixPrompt*( ), *nfsIdSet*( ), *nfsAuthUnixGet*( ), *nfsAuthUnixShow*( )

nfsAuthUnixGet( )

NAME

nfsAuthUnixGet( ) - get the NFS UNIX authentication parameters

SYNOPSIS

```
VOID nfsAuthUnixGet (machname, pUid, pGid, pNgids, gids)
    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.

SEE ALSO

nfsLib, *nfsAuthUnixPrompt*( ), *nfsAuthUnixSet*( ), *nfsAuthUnixShow*( ), *nfsIdSet*( )

nfsIdSet( )

NAME

nfsIdSet( ) - set the ID number of the NFS UNIX authentication parameters

SYNOPSIS

```
VOID nfsIdSet (uid)
    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.

SEE ALSO

nfsLib, *nfsAuthUnixPrompt*( ), *nfsAuthUnixGet*( ), *nfsAuthUnixSet*( ), *nfsAuthUnixShow*( )

rawFsLib

NAME

rawFsLib - raw block device file system library

SYNOPSIS

rawFsDevInit( ) - associate a block device with raw volume functions
rawFsInit( ) - prepare to use the raw volume library
rawFsModeChange( ) - modify mode of raw device volume
rawFsReadyChange( ) - notify **rawFsLib** of a change in ready status
rawFsVolUnmount( ) - disable a raw device volume

```
RAW_VOL_DESC *rawFsDevInit (volName, pBlkDev)
STATUS rawFsInit (maxFiles)
VOID rawFsModeChange (vdptr, newMode)
VOID rawFsReadyChange (vdptr)
STATUS rawFsVolUnmount (vdptr)
```

DESCRIPTION

This library provides basic services for disk devices that do not use a standard file or directory structure. The disk volume is treated much like a large file. Portions of it may be read, written, or the current position within the disk may be changed. However, there is no high-level organization of the disk into files or directories.

USING THIS LIBRARY

The various functions provided by the Vx960 rawFs file system may be separated into three broad groups: general initialization, device initialization, and file system operation.

The *rawFsInit*( ) function is the principal initialization function; it need only be called once, regardless of how many rawFs devices will be used.

A separate rawFs function is used for device initialization. For each rawFs device, *rawFsDevInit*( ) must be called to install the device.

Lastly, several functions are provided to inform the file system of changes in the system environment. The *rawFsModeChange*( ) call may be used to modify the readability or writability of a particular device. The *rawFsReadyChange*( ) function 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, the *rawFsVolUnmount*( ) function 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 functions is discussed in the following sections.

INITIALIZATION

Before any other routines in **rawFsLib** can be used, the *rawFsInit*( ) routine must be called to initialize the library. This call specifies the maximum number of raw device file descriptors that can be open simultaneously and allocates memory for that many raw file descriptors. Any attempt to open more raw device file descriptors than the specified maximum will result in errors from *open*( ) or *creat*( ).

During the *rawFsInit*( ) call, the raw device library is installed as a driver in the I/O system driver table. The driver number associated with it is then placed in a global variable, *rawFsDrvNum*.

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

DEFINING A RAW DEVICE

To use this library for a particular device, the device structure used by the device driver must contain, as the very first item, a block device description structure (BLK_DEV). This must be initialized before calling *rawFsDevInit*( ). In the BLK_DEV structure, the driver includes the addresses of five routines it must supply: one that reads one or more blocks, one that writes one or more blocks, one that performs I/O control (*ioctl*( )) on the device, one that checks the status of the the device, and one that resets the device. The BLK_DEV structure also contains fields that describe the physical configuration of the device. See the "I/O System" chapter in the *Programmer's Guide* for more information on defining block devices.

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 the *iosDevShow*( ) routine.

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 (volName, pBlkDev);
char   *volName; /* name to be used for volume   *
BLK_DEV  *pBlkDev; /* pointer to device descriptor *
```

Unlike the Vx960 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 ***dosFsMkfs***( ) and ***rt11Mkfs***( ) routines).

After ***rawFsDevInit***( ) has been called, when **rawFsLib** 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.

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 is 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 explicitly synchronize the disk first. In all other circumstances, failure to synchronize the volume before unmounting may result in lost data.

IOCTL FUNCTIONS

The Vx960 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:", WRITE);
status = ioctl (fd, FIODISKFORMAT);
```

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 Vx960 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);
```

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);
```

FIOGETNAME- Gets the file name of the fd 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);
```

FIOFLUSH - Writes all modified file descriptor buffers to the physical device.

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

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 FILE

rawFsLib.h

SEE ALSO

ioLib, iosLib, dosFsLib, rt11FsLib, ramDrv,
Programmer's Guide: I/O System, Local File Systems

rawFsDevInit( )

NAME

rawFsDevInit( ) - associate a block device with raw volume functions

SYNOPSIS

```
RAW_VOL_DESC *rawFsDevInit (volName, pBlkDev)
    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 (e.g. *open*( ), *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 Vx960 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 (maxFiles)
    int  maxFiles;  /* max # 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 Vx960 I/O system driver table. The driver number assigned to rawFsLib is placed in the global variable *rawFsDrvNum*. This number will later be associated with system file descriptors opened to rawFs devices.

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

RETURNS

OK, or ERROR.

SEE ALSO

rawFsLib

rawFsModeChange( )

NAME

rawFsModeChange( ) - modify mode of raw device volume

SYNOPSIS

```
VOID rawFsModeChange (vdptr, newMode)
    RAW_VOL_DESC  *vdptr;    /* pointer to volume descriptor */
    int           newMode;   /* READ/WRITE/UPDATE (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. See the manual entry for *rawFsReadyChange*( ).

The driver's device initialization routine should initially set the mode to UPDATE (i.e., both READ and WRITE).

RETURNS

N/A

SEE ALSO

rawFsLib

rawFsReadyChange( )

NAME

rawFsReadyChange( ) - notify **rawFsLib** of a change in ready status

SYNOPSIS

```
VOID rawFsReadyChange (vdptr)
    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 (vdptr)
    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

rebootLib

NAME

rebootLib - reboot support library

SYNOPSIS

reboot( ) - reset network devices and transfer control to boot ROMs
rebootHookAdd( ) - add a routine to be called at reboot

```
VOID reboot (startType)
STATUS rebootHookAdd (rebootHook)
```

DESCRIPTION

This library provides reboot support. To restart Vx960, the routine *reboot*( ) can be called at any time by typing ^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. 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*( ).

INCLUDE FILE

sysLib.h

SEE ALSO

sysLib, bootConfig, bootInit

reboot( )

NAME

reboot( ) - reset network devices and transfer control to boot ROMs

SYNOPSIS

```
VOID reboot (startType)
    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.

The bit values for *startType* are defined in **sysLib.h**:

BOOT_NORMAL - causes the system to go through the countdown sequence and try to reboot Vx960 automatically. Memory is not cleared.

BOOT_NO_AUTOBOOT - causes the system to display the Vx960 boot prompt and wait for user input to the boot ROM monitor. Memory is not cleared.

BOOT_CLEAR - the same as BOOT_NORMAL, except that memory is cleared.

BOOT_QUICK_AUTOBOOT - the same as BOOT_NORMAL, except the countdown is shorter.

RETURNS

N/A

SEE ALSO

rebootLib, *sysToMonitor*( ), *rebootHookAdd*( )

rebootHookAdd( )

NAME

rebootHookAdd( ) - add a routine to be called at reboot

SYNOPSIS

```
STATUS rebootHookAdd (rebootHook)
    FUNCPTR  rebootHook;  /* routine to be called at reboot */
```

DESCRIPTION

This routine adds the specified routine to a list of routines to be called when Vx960 is rebooted. The specified routine should be declared as follows:

```
VOID rebootHook (startType)
int startType;   /* startType is passed to all hooks *
```

RETURNS

OK, or ERROR if memory is insufficient.

SEE ALSO

rebootLib, *reboot*( )

remLib

NAME

remLib - remote command library

SYNOPSIS

rcmd( ) - execute a shell command on a remote machine
rresvport( ) - open a socket with a privileged port bound to it
remCurIdGet( ) - get the current user name and password
remCurIdSet( ) - set the remote user name and password
iam( ) - set the remote user name and password
whoami( ) - display the current remote identity
bindresvport( ) - bind a socket to a privileged IP port

```
int rcmd (host, remotePort, localUser, remoteUser, cmd, fd2p)
int rresvport (alport)
VOID remCurIdGet (user, passwd)
STATUS remCurIdSet (newUser, newPasswd)
STATUS iam (newUser, newPasswd)
VOID whoami ()
STATUS bindresvport (sd, sin)
```

DESCRIPTION

This library supplies routines found under BSD 4.2 (rcmd and rresvport), and the routines that provide authorization for network file access via netDrv and remote command execution via rcmd.

INCLUDE FILE

remLib.h

SEE ALSO

inetLib, *Programmer's Guide: Network*

rcmd( )

NAME

rcmd( ) - execute a shell command on a remote machine

SYNOPSIS

```
int rcmd (host, remotePort, localUser, remoteUser, cmd, fd2p)
    char  *host;        /* host name or inet address */
    int   remotePort;   /* remote port to connect to (rshd) */
    char   *localUser;     /* local user name */
    char   *remoteUser;    /* remote user name */
    char   *cmd;           /* command */
    int    *fd2p;          /* if this pointer is non-zero, stderr socket
                            * is opened and socket descriptor is filled in */
```

DESCRIPTION

This routine executes a command on a remote machine, using the remote shell daemon (*rshd*) on the remote system. It is analogous to the UNIX command rcmd.

RETURNS

A socket descriptor if the remote shell daemon accepts, or ERROR if the remote command fails.

SEE ALSO

remLib, BSD 4.2 manual entry for rcmd

rresvport( )

NAME

rresvport( ) - open a socket with a privileged port bound to it

SYNOPSIS

```
int rresvport (alport)
    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 command *rresvport*.

RETURNS

A socket descriptor, or ERROR if either the socket cannot be opened or all ports are in use.

SEE ALSO

remLib, BSD 4.2 manual entry for *rresvport*

remCurIdGet( )

NAME

remCurIdGet( ) - get the current user name and password

SYNOPSIS

```
VOID remCurIdGet (user, passwd)
    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 being 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 (newUser, newPasswd)
    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**).

RETURNS

OK, or ERROR if the name or password is too long.

SEE ALSO

remLib, *iam*( ), *whoami*( )

iam( )

NAME

iam( ) - set the remote user name and password

SYNOPSIS

```
STATUS iam (newUser, newPasswd)
    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**).

RETURNS

OK, or ERROR if the call fails.

SEE ALSO

remLib, *whoami*( ), *remCurIdGet*( ), *remCurIdSet*( )

whoami( )

NAME

whoami( ) - display the current remote identity

SYNOPSIS

```
VOID whoami ()
```

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*( )

bindresvport( )

NAME

bindresvport( ) - bind a socket to a privileged IP port

SYNOPSIS

```
STATUS bindresvport (sd, sin)
    int                sd;    /* socket to be bound */
    struct sockaddr_in *sin;  /* socket address -- value/result */
```

DESCRIPTION

Privileged IP ports (numbers between and including 0 and 1023) are reserved for privileged programs. The *bindresvport*( ) routine will pick a port number between 600 and 1023 that is not being used by any other programs and bind the socket passed as *sd* to that port.

RETURNS

OK, or ERROR if the address family specified in *sin* is not supported or the call fails.

SEE ALSO

remLib

rlogLib

NAME

rlogLib - remote login library

SYNOPSIS

rlogInit( ) - initialize the remote login facility
rlogind( ) - the Vx960 remote login daemon
rlogin( ) - log in to a remote host

```
STATUS rlogInit ()
VOID rlogind ()
STATUS rlogin (host)
```

DESCRIPTION

This library provides a remote login facility for Vx960 that uses the UNIX *rlogin* protocol as implemented in UNIX BSD 4.2 to allow users at a Vx960 terminal to login to remote systems via the network; and allow users at remote systems to log in to Vx960 via the network.

A Vx960 user may log in to any other remote Vx960 or UNIX system via the network, by calling ***rlogin***( ) from the shell.

The remote login daemon, ***rlogind***( ), allows remote users to log into Vx960. The daemon is started by calling ***rlogInit***( ); if INCLUDE_RLOGIN is defined in **configAll.h**, it is called by the root task, *usrRoot*( ), in **usrConfig.c**. The remote login daemon accepts remote login requests from another Vx960 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 Vx960 pseudo-terminal driver **ptyDrv**.

SEE ALSO

ptyDrv, **telnetLib**, UNIX documentation for *rlogin*, *rlogind*, and *pty*

rlogInit( )

NAME

rlogInit( ) - initialize the remote login facility

SYNOPSIS

```
STATUS rlogInit ()
```

DESCRIPTION

This routine initializes the remote login facility. It creates a pty device, and spawns ***rlogind*( )**. If INCLUDE_RLOGIN is defined in **configAll.h**, it is called from ***usrRoot*( )** in **usrConfig.c**, before any other system tries to log in to this Vx960 system via the UNIX *rlogin* protocol.

RETURNS

OK, or ERROR.

SEE ALSO

rlogLib

rlogind( )

NAME

***rlogind*( )** - the Vx960 remote login daemon

SYNOPSIS

```
VOID rlogind ()
```

DESCRIPTION

This routine provides a facility for remote users to log into Vx960 over the network. If INCLUDE_RLOGIN is defined in **configAll.h**, it is spawned by ***rlogInit*( )**, which is called by the root task, ***usrRoot*( )**, in **usrConfig.c**.

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. ***rlogind*( )** 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 Vx960 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. ***rlogind*( )** creates two child processes, *rlogInTask* and *rlogOutTask*, whenever a remote user is logged in. These processes exit when the remote connection is terminated.

RETURNS

N/A

SEE ALSO

rlogLib, *iam*( )

rlogin( )

NAME

rlogin( ) - log in to a remote host

SYNOPSIS

```
STATUS rlogin (host)
    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 Vx960 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*( )

rngLib

NAME

rngLib - ring buffer subroutine library

SYNOPSIS

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

```
RING_ID rngCreate (nbytes)
VOID rngDelete (ringId)
VOID rngFlush (ringId)
int rngBufGet (rngId, buffer, maxbytes)
int rngBufPut (rngId, buffer, nbytes)
BOOL rngIsEmpty (ringId)
BOOL rngIsFull (ringId)
int rngFreeBytes (ringId)
int rngNBytes (ringId)
VOID rngPutAhead (ringId, byte, offset)
VOID rngMoveAhead (ringId, n)
```

DESCRIPTION

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

However, the ring buffer pointers are manipulated in such a way that a reader task (invoking *rngBufGet*( )) and a writer task (invoking *rngBugPut*( )) 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 semephore 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 FILE

rngLib.h

rngCreate( )

NAME

rngCreate( ) - create an empty ring buffer

SYNOPSIS

```
RING_ID rngCreate (nbytes)
    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

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 (ringId)
    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 (ringId)
    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

rngBufGet( )

NAME

rngBufGet( ) - get characters from a ring buffer

SYNOPSIS

```
int rngBufGet (rngId, buffer, maxbytes)
    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 (rngId, buffer, nbytes)
    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

rngIsEmpty( )

NAME

rngIsEmpty( ) - test if a ring buffer is empty

SYNOPSIS

```
BOOL rngIsEmpty (ringId)
    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 (ringId)
    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

rngFreeBytes( )

NAME

rngFreeBytes( ) - determine the number of free bytes in a ring buffer

SYNOPSIS

```
int rngFreeBytes (ringId)
    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

rngNBytes( )

NAME

rngNBytes( ) - determine the number of bytes in a ring buffer

SYNOPSIS

```
int rngNBytes (ringId)
    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 (ringId, byte, offset)
    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

rngMoveAhead( )

NAME

rngMoveAhead( ) - advance a ring pointer by *n* bytes

SYNOPSIS

```
VOID rngMoveAhead (ringId, n)
    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

routeLib

NAME

routeLib - network route manipulation library

SYNOPSIS

routeAdd( ) - add a route
routeDelete( ) - delete a route
routeShow( ) - display host and network routing tables

```
STATUS routeAdd (destination, gateway)
STATUS routeDelete (destination, gateway)
VOID routeShow ()
```

DESCRIPTION

This library contains the routines *routeAdd*( ) and *routeDelete*( ) for changing and examining the network routing tables. Routines are provided for adding and deleting routes that go through a passive gateway. The routine *routeShow*( ) displays the routing tables. Vx960 has no routing daemon; therefore, the tables must be maintained manually.

SEE ALSO

hostLib, *Programmer's Guide: Network*

routeAdd( )

NAME

routeAdd( ) - add a route

SYNOPSIS

```
STATUS routeAdd (destination, gateway)
    char  *destination;  /* inet address or name of route destination     */
    char  *gateway;      /* inet address or name of gateway to destination */
```

DESCRIPTION

This routine adds gateways to the network routing tables. It is called from a Vx960 machine that needs to establish a gateway to a destination network (or machine).

Both *destination* and *gateway* can be specified in standard Internet address format (e.g., 90.0.0.2) or by their host names, as specified by *hostAdd*( ).

EXAMPLE

The call:

```
routeAdd ("90.0.0.0", "gate")
```

tells Vx960 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.

The call:

```
routeAdd ("90.0.0.0", "91.0.0.3")
```

tells Vx960 that the machine with the Internet address 91.0.0.3 is the gateway to network 90.0.0.0.

The call:

```
routeAdd ("destination", "gate")
```

tells Vx960 that the machine with the host name "gate" is the gateway to the machine named "destination". The host names "gate" and "destination" must have already been created by *hostAdd*( ).

The call:

```
routeAdd ("0", "gate")
```

tells Vx960 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 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, *Programmer's Guide: Network*

routeDelete( )

NAME

routeDelete( ) - delete a route

SYNOPSIS

```
STATUS routeDelete (destination, gateway)
    char  *destination;  /* inet address or name of route destination      */
    char  *gateway;      /* inet address 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*( ), *Programmer's Guide: Network*

routeShow( )

NAME

routeShow( ) - display host and network routing tables

SYNOPSIS

```
VOID routeShow ()
```

DESCRIPTION

This routine displays the current routing information in the routing table.

EXAMPLE

```
-> routeShow

ROUTE NET TABLE
destination      gateway          flags  Refcnt  Use            Interface
------------------------------------------------------------------------
90.0.0.0         90.0.0.63        1      1       142            enp0
------------------------------------------------------------------------

ROUTE HOST TABLE
destination      gateway          flags  Refcnt  Use            Interface
------------------------------------------------------------------------
127.0.0.1        127.0.0.1        5      0       82             lo0
------------------------------------------------------------------------
```

The flags field represents a decimal value of the flags specified for a given route. The following is a list of currently available flag values:

0x1	- route is usable (i.e., "up")

0x2	- destination is a gateway
0x4	- host specific routing entry
0x10	- created dynamically (by redirect)
0x20	- modified dynamically (by redirect)

In the above 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

routeLib, *Programmer's Guide: Network*

rpcLib

NAME

rpcLib - RPC support library

SYNOPSIS

rpcInit( ) - initialize RPC package
rpcTaskInit( ) - initialize a task's access to the RPC package

```
STATUS rpcInit ()
STATUS rpcTaskInit ()
```

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's *Remote Procedure Call Programming Guide.*

The RPC facility is enabled by defining INCLUDE_RPC in **configAll.h** or in the **config.h** file for the target CPU.

Vx960 supports Network File System (NFS) and gdb960, which are built on top of RPC. If NFS or INCLUDE_RDB (remote debug) is configured into the Vx960 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.

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. The task-specific data structures are automatically deleted when the task exits.

INCLUDE FILE

rpc.h

SEE ALSO

nfsLib, **nfsDrv**, **dbxLib**, Sun's RPC Programming Guide

rpcInit( )

NAME

rpcInit( ) - initialize RPC package

SYNOPSIS

```
STATUS rpcInit ()
```

DESCRIPTION

This routine must be called before any task can use RPC; it spawns the portmap daemon. If INCLUDE_RPC is defined in **configAll.h**, it is called by the root task *usrRoot*( ) in **usrConfig.c**.

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 ()
```

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

rt11FsLib

NAME

rt11FsLib - RT-11 media-compatible file system library

SYNOPSIS

rt11FsDevInit( ) - initialize the RT-11 device descriptor
rt11FsInit( ) - prepare to use the RT-11 library
rt11FsMkfs( ) - initialize a device and create an RT-11 file system
rt11FsDateSet( ) - set the current date
rt11FsReadyChange( ) - notify **rt11FsLib** of a change in ready status
rt11FsModeChange( ) - modify the mode of an RT-11 volume

```
RT_VOL_DESC *rt11FsDevInit (devName, pBlkDev, rt11Fmt, nEntries, changeNoWarn)
STATUS rt11FsInit (maxFiles)
RT_VOL_DESC *rt11FsMkfs (volName, pBlkDev)
VOID rt11FsDateSet (year, month, day)
VOID rt11FsReadyChange (vdptr)
VOID rt11FsModeChange (vdptr, newMode)
```

DESCRIPTION

This library provides services for file-oriented device drivers which use the RT-11 file standard. This module takes care of all the buffering, directory maintenance, and RT-11-specific details necessary.

USING THIS LIBRARY

The various functions provided by the Vx960 rt11Fs file system may be separated into three broad groups: general initialization, device initialization, and file system operation.

The *rt11FsInit*( ) function is the principal initialization function; it need only be called once, regardless of how many rt11Fs devices will be used.

Other rt11Fs functions are used for device initialization. For each rt11Fs device, either *rt11FsDevInit*( ) or *rt11FsMkfs*( ) must be called to install the device and define its configuration.

Lastly, several functions are provided to inform the file system of changes in the system environment. The *rt11FsDateSet*( ) function is used to set the current date. The *rt11FsModeChange*( ) call may be used to modify the readability or writability of a particular device. Finally, the *rt11FsReadyChange*( ) function 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.

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

INITIALIZING RT11LIB

Before any other routines in **rt11FsLib** can be used, the routine *rt11FsInit*( ) must be called to initialize this library. This call specifies the maximum number of RT-11 files that can be open simultaneously and allocates memory for that many RT-11 file descriptors. Attempts to open more RT-11 files than the specified maximum will result in errors from *open*( ) or *creat*( ).

To enable this initialization, define INCLUDE_RT11FS in **configAll.h**; *rt11FsInit*( ) will then be called from the root task, *usrRoot*( ), in **usrConfig.c**.

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). See *Programmer's Guide: I/O System* for more information on defining block devices.

The device is associated with the RT-11 file system via the *rt11FsDevInit*( ) call. The arguments to *rt11FsDevInit*( ) include the name to be used for the RT-11 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 Vx960-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 (except to add the track offset as described above) to maintain RT-11 compatibility.

Note that if the number of files specified is different than

RT_FILES_FOR_2_BLOCK_SEG under either a Vx960 system or an RT-11 system, compatibility is lost because Vx960 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 for you.

In the Vx960 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 a six-character filename, followed by a period (.), followed by an optional three-character extension.

DIRECTORY ENTRIES

An *ioctl*( ) call with the FIODIRENTRY function returns information about a particular directory entry. A pointer to a REQ_DIR_ENTRY structure is passed as the parameter. The field *entryNum* in the REQ_DIR_ENTRY structure must be set to the desired entry number. The name of the file, its size (in bytes), and its creation date are returned in the structure. If the specified entry is empty (i.e., if it represents an unallocated section of the disk), the name will be an empty string, the size will be the size of the available disk section, and the date will be meaningless. Typically, the entries are accessed sequentially, starting with *entryNum* = 0, until the terminating entry is reached, indicated by a return code of ERROR.

DIRECTORIES IN MEMORY

A copy of the directory for each volume is kept in memory (in the RT_VOL_DESC structure). This speeds up directory accesses, but requires that **rt11FsLib** be notified when disks are changed (i.e., floppies are swapped). If the driver can find this out (by interrogating controller status or by receiving an interrupt) the driver simply calls ***rt11FsReadyChange*( )** when a disk is inserted or removed. The library **rt11FsLib** will automatically try to remount the device next time it needs it.

If the driver does not have access to the information that disk volumes have been changed, the *changeNoWarn* parameter should be set to TRUE when the device is defined using ***rt11FsDevInit*( )**. This will cause the disk to be automatically remounted before each ***open*( )**, ***creat*( )**, ***delete*( )**, and directory listing.

The routine *rt11FsReadyChange*( ) can also be called by user tasks by issuing an *ioctl*( ) call with FIODISKCHANGE as the function code.

ACCESSING THE RAW DISK

As a special case in *open*( ) and *creat*( ) calls, **rt11FsLib** recognizes a null file name to indicate access to the entire "raw" disk, as opposed to a file on the disk. Access in raw mode is useful for a disk that has no file system. For example, to initialize a new file system on the disk, use an *ioctl*( ) call with FIODISKINIT. To read the directory of a disk for which no file names are known, open the raw disk and use an *ioctl*( ) call with the function FIODIRENTRY.

RT-11 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 Vx960 RT-11 file system supports the following *ioctl*( ) functions. The functions listed are defined in the header **ioLib.h**. Unless stated otherwise, the fd used for these functions may be any fd which is 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:", WRITE);
status = ioctl (fd, FIODISKFORMAT);
```

FIODISKINIT - Initializes an RT-11 file system on the disk volume. This routine does not format the disk; formatting must be done by the driver. The fd should be obtained by opening the entire volume in raw mode:

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

FIODISKCHANGE

- Announces a media change. It performs the same function as *rt11FsReadyChange*( ). This function may be called from interrupt level:

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

FIOGETNAME- Gets the file name of the fd 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);
```

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);
```

FIOSQUEEZE - Coalesces fragmented free space on an RT-11 volume:

```
status = ioctl (fd, FIOSQUEEZE);
```

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, the *readdir*( ) routine is used to read a directory, rather than using the FIOREADDIR function directly. See **dirLib**.

```
DIR dirStruct;
fd = open ("directory", READ);
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

rt11FsDevInit( )

NAME

rt11FsDevInit( ) - initialize the RT-11 device descriptor

SYNOPSIS

```
RT_VOL_DESC *rt11FsDevInit (devName, pBlkDev, rt11Fmt, nEntries, changeNoWarn)
    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 will be 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 RT-11 library

SYNOPSIS

```
STATUS rt11FsInit (maxFiles)
    int  maxFiles;  /* maximum number of simultaneously */
                    /* open RT-11 files    */
```

DESCRIPTION

This routine initializes the RT-11 library. It must be called exactly once, before any other routine in the library. The *maxFiles* parameter specifies the number of RT-11 files that may be open at once. This routine initializes the necessary memory structures and semaphores.

To enable this initialization, define INCLUDE_RT11FS in **configAll.h**; *rt11FsInit*( ) will then be called from the root task, *usrRoot*( ), in **usrConfig.c**.

RETURNS

OK, or ERROR if memory is insufficient.

SEE ALSO

rt11FsLib

rt11FsMkfs( )

NAME

rt11FsMkfs( ) - initialize a device and create an RT-11 file system

SYNOPSIS

```
RT_VOL_DESC *rt11FsMkfs (volName, pBlkDev)
    char     *volName;  /* volume name to use */
    BLK_DEV  *pBlkDev;  /* pointer to block device struct */
```

DESCRIPTION

This routine provides a quick method of creating an RT-11 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 RT-11 parameters expected by *rt11FsDevInit*( ). The directory size is set to RT_FILES_FOR_2_BLOCK_SEG (defined in **rt11FsLib.h**). No standard RT-11 disk format is assumed; this allows use of RT-11 on block devices with an odd number of sectors per track. The "changeNoWarn" flag is defined as false, indicating that that the disk will not be replaced without an *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 RT-11 volume descriptor (RT_VOL_DESC), or NULL if there is an error.

SEE ALSO

rt11FsLib

rt11FsDateSet( )

NAME

rt11FsDateSet( ) - set the current date

SYNOPSIS

```
VOID rt11FsDateSet (year, month, day)
    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 current date for the RT-11 file system. All files created will have this creation date.

To set a blank date, invoke the command:

```
rt11FsDateSet (72, 0, 0); /* a date outside RT-11's epoch *
```

RETURNS

N/A

SEE ALSO

rt11FsLib

rt11FsReadyChange( )

NAME

rt11FsReadyChange( ) - notify **rt11FsLib** of a change in ready status

SYNOPSIS

```
VOID rt11FsReadyChange (vdptr)
    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

rt11FsModeChange( )

NAME

rt11FsModeChange( ) - modify the mode of an RT-11 volume

SYNOPSIS

```
VOID rt11FsModeChange (vdptr, newMode)
    RT_VOL_DESC  *vdptr;    /* pointer to volume descriptor */
    int          newMode;   /* READ/WRITE/UPDATE (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 UPDATE, (e.g., both READ and WRITE).

RETURNS

N/A

SEE ALSO

rt11FsLib

scsiLib

NAME

scsiLib - Small Computer System Interface (SCSI) library

SYNOPSIS

scsiPhysDevDelete( ) - delete a SCSI physical device structure
scsiPhysDevCreate( ) - create a SCSI physical device structure
scsiPhysDevIdGet( ) - return a pointer to a SCSI physical device 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
scsiBusReset( ) - pulse the reset signal on the SCSI bus
scsiTestUnitRdy( ) - issue a TEST_UNIT_READY command to a SCSI device
scsiFormatUnit( ) - issue a FORMAT_UNIT command to a SCSI device
scsiInquiry( ) - issue an INQUIRY 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 sectors from an SCSI block device
scsiWrtSecs( ) - write sectors to an SCSI block device
scsiReqSense( ) - issue a REQUEST SENSE command to a device and read the result
scsiIoctl( ) - perform a device-specific control function

```
STATUS scsiPhysDevDelete (pScsiPhysDev)
SCSI_PHYS_DEV *scsiPhysDevCreate (pScsiCtrl, devBusId, devLUN, selTimeOut,  ...
SCSI_PHYS_DEV * scsiPhysDevIdGet (pScsiCtrl, devBusId, devLUN)  ...
STATUS scsiAutoConfig (pScsiCtrl, selTimeOut)  ...
STATUS scsiShow (pScsiCtrl)
BLK_DEV *scsiBlkDevCreate (pScsiPhysDev, numBlocks, blockOffset)
VOID scsiBlkDevInit (pScsiBlkDev, blksPerTrack, nHeads)
STATUS scsiBusReset (pScsiCtrl)
STATUS scsiTestUnitRdy (pScsiPhysDev)
STATUS scsiFormatUnit (pScsiPhysDev, cmpDefectList, defListFormat, vendorUnique,
STATUS scsiInquiry (pScsiPhysDev, buffer, bufLength)
STATUS scsiModeSelect (pScsiPhysDev, pageFormat, saveParams, buffer, bufLength)
STATUS scsiModeSense (pScsiPhysDev, pageControl, pageCode, buffer, bufLength)
STATUS scsiReadCapacity (pScsiPhysDev, pLastLBA, pBlkLength)
STATUS scsiRdSecs (pScsiBlkDev, sector, numSecs, buffer)
STATUS scsiWrtSecs (pScsiBlkDev, sector, numSecs, buffer)
STATUS scsiReqSense (pScsiPhysDev, buffer, bufLength)
STATUS scsiIoctl (pScsiPhysDev, function, arg)
```

DESCRIPTION

This library implements the SCSI protocol in a controller-independent manner. It implements only the SCSI initiator function, so that the concept of a Vx960 target acting as a SCSI target is not currently supported. Furthermore, in the current implementation, a Vx960 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. Normal completion ends with a "Command Complete" message from the target, followed by disconnection from the SCSI bus. In addition, if the status from the target is "Check Condition", the transaction continues with the initiator issuing a "Request Sense" command to gain more information on the exceptional condition reported.

Many of the subroutines in **scsiLib** 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, are 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." Typically, for off-board SCSI controllers this routine would rearrange the fields of the SCSI_TRANSACTION structure into a similar structure for the given hardware, which would then carry out the transaction through firmware control. Drivers for on-board SCSI controller chips can use the ***scsiTransact*( )** routine in **scsiLib**, as long as they provide the other functions specified in the SCSI_CTRL structure. The subject of interfaces between **scsiLib** and drivers for SCSI controllers will be addressed in a separate application note.

NOTE: Disconnect/reconnect is not currently supported.

SUPPORTED SCSI DEVICES

SCSI peripherals conforming to the standards specified in "Common Command Set (CCS) of the SCSI," Rev. 4.B. are strongly recommended. Most SCSI peripherals currently offered support CCS. While an attempt has been made to have **scsiLib** 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.

In theory, all classes of SCSI devices are supported. The **scsiLib** library provides the capability to transact any SCSI command on any SCSI device through the FIOSCSICOMMAND function of the *scsiIoctl*( ) routine.

Only direct-access devices (disks) are supported by a file system. For other types of devices, additional, higher-level software is necessary to map user-level commands to SCSI transactions.

CONFIGURING SCSI CONTROLLERS

The routines to create and initialize a given SCSI controller are specific to the controller and normally will be 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 (e.g., clock rate). *xxCtrlInit*( ) initializes the remaining fields in the XX_SCSI_CTRL structure and writes any necessary registers on the SCSI controller to effect the desired initialization. There is no reason this routine could not be called multiple times, although in practice this would probably occur infrequently. For example, one might wish to change the bus ID of the SCSI controller without rebooting the Vx960 system.

CONFIGURING PHYSICAL SCSI DEVICES

Before a device can be used, it must be "created" (i.e., declared). This occurs with a call to *scsiPhysDevCreate*( ), and can only be done after a SCSI_CTRL structure exists and is properly initialized.

```
SCSI_PHYS_DEV *scsiPhysDevCreate (pScsiCtrl, devBusId, devLUN, selTimeOut,
     devType, removable, numBlocks, blockSize)
   SCSI_CTRL *pScsiCtrl; /* ptr to SCSI controller info *
   int devBusId;  /* device's SCSI bus ID *
   int devLUN;    /* device's logical unit number *
   UINT selTimeOut;  /* time-out for selecting device (usec) *
   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 may be left unspecified as follows:

selTimeOut
- If 0 , use the default (250 msec)

devType
- If -1, issue an INQUIRY to determine

numBlocks,
- If 0, issue a READ_CAPACITY to determine

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 simply an array of contiguously-addressed blocks with a given starting block address and number of blocks. The routine *scsiBlkDevCreate*( ) should be called once for each block device partition. Under normal usage, logical partitions should not overlap.

```
SCSI_BLK_DEV *scsiBlkDevCreate (pScsiPhysDev, numBlocks, blockOffset)
    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: If *numBlocks* is 0 the rest of device is used.

ATTACHING FILE SYSTEMS TO LOGICAL PARTITIONS

Before files can be read or written to a disk partition, a file system (e.g., RT-11, MS-DOS) must be initialized on it. See the manual entries for **rt11FsLib** or **dosFsLib**.

TRANSACTING ARBITRARY COMMANDS TO SCSI DEVICES

The **scsiLib** library provides routines which implement many common SCSI commands. Still, there are situations that require commands that are not supported by **scsiLib** (e.g., control devices that are not direct access devices). Arbitrary commands are handled with the FIOSCSICOMMAND option to *scsiIoctl*( ). The *arg* parameter for this option is a pointer to a valid SCSI_TRANSACTION structure.

Typically, a call to *scsiIoctl*( ) is written as a subroutine of the form:

```
STATUS myScsiCommand (pScsiPhysDev, buffer, bufLength, someParam)
    SCSI_PHYS_DEV *pScsiPhysDev;/* ptr to SCSI physical device *
    char *buffer;  /* ptr to data buffer *
    int bufLength;  /* length of buffer in bytes *
    int someParam;  /* parameter 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 = <READ (0) or WRITE (1)>;
   myScsiXaction.dataLength    = bufLength;

   /* if dataDirection is READ, 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

SEE ALSO

rt11FsLib, dosFsLib, *Programmer's Guide: I/O System*

scsiPhysDevDelete( )

NAME

scsiPhysDevDelete( ) - delete a SCSI physical device structure

SYNOPSIS

```
STATUS scsiPhysDevDelete (pScsiPhysDev)
    SCSI_PHYS_DEV  *pScsiPhysDev;  /* ptr to SCSI physical device info */
```

RETURNS

OK, or ERROR if *pScsiPhysDev* is NULL, or if SCSI block devices have been created on the device.

SEE ALSO

scsiLib

scsiPhysDevCreate( )

NAME

scsiPhysDevCreate( ) - create a SCSI physical device structure

SYNOPSIS

```
SCSI_PHYS_DEV *scsiPhysDevCreate (pScsiCtrl, devBusId, devLUN, selTimeOut,
                                  devType, removable, numBlocks, blockSize)
    SCSI_CTRL  *pScsiCtrl;   /* ptr to SCSI controller info */
    int        devBusId;     /* device's SCSI bus ID */
    int        devLUN;       /* device's logical unit number */
    UINT       selTimeOut;   /* time-out for selecting device (usec) */
    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 must be invoked before a SCSI device can be accessed. It should be called once for each physical device on the SCSI bus.

If *devType* is specified as NONE (-1), an INQUIRY command is issued to determine the device type, with the added benefit of acquiring the make and model number (*scsiShow*( ) displays this information). Common values of *devType* can be found in **scsiLib.h** or the SCSI specification itself.

NOTE

If a SCSI device does not support the READ CAPACITY command, then *numBlocks* and *blockSize* must be non-zero.

RETURNS

A pointer to the created SCSI_PHYS_DEV, or NULL if the routine is unable to create a physical device.

SEE ALSO

scsiLib

scsiPhysDevIdGet( )

NAME

scsiPhysDevIdGet( ) - return a pointer to a SCSI physical device structure

SYNOPSIS

```
SCSI_PHYS_DEV * scsiPhysDevIdGet (pScsiCtrl, devBusId, devLUN)
    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 structure SCSI_PHYS_DEV of the SCSI physical device at the given bus ID, logical unit number, and SCSI controller.

RETURNS

A pointer to the structure SCSI_PHYS_DEV or NULL if the structure does not exist.

SEE ALSO

scsiLib

scsiAutoConfig( )

NAME

scsiAutoConfig( ) - configure all devices connected to a SCSI controller

SYNOPSIS

```
STATUS scsiAutoConfig (pScsiCtrl, selTimeOut)
    SCSI_CTRL  *pScsiCtrl;  /* ptr to SCSI controller info */
```

```
        UINT        selTimeOut;  /* time-out for selecting devices (usec) */
```

DESCRIPTION

This routine cycles through all legal SCSI bus IDs (and LUNs) and calls *scsiPhysDevCreate*( ) with default parameters for each device. All devices that support the INQUIRY command should be successfully configured. The *scsiShow*( ) command can be used to find the system's table of the SCSI physical devices attached to a given SCSI controller. In addition, *scsiPhysDevIdGet*( ) can fetch a pointer to the SCSI_PHYS_DEV structure for the device at a given SCSI bus ID and LUN.

RETURNS

OK, or ERROR if *pScsiCtrl* and *pSysScsiCtrl* are both NULL.

SEE ALSO

scsiLib

scsiShow( )

NAME

scsiShow( ) - list the physical devices attached to a SCSI controller

SYNOPSIS

```
STATUS scsiShow (pScsiCtrl)
    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 (bytes), and a pointer to the associated SCSI_PHYS_DEV structure for each physical SCSI device known to be attached to a given SCSI controller.

NOTE: If *pScsiCtrl* is NULL, the value of the global variable *pSysScsiCtrl* will be used, unless it is also NULL.

RETURNS

OK, or ERROR if the routine is not successful.

SEE ALSO

scsiLib

scsiBlkDevCreate( )

NAME

scsiBlkDevCreate( ) - define a logical partition on a SCSI block device

SYNOPSIS

```
BLK_DEV *scsiBlkDevCreate (pScsiPhysDev, numBlocks, blockOffset)
    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 a partition is not created.

SEE ALSO

scsiLib

scsiBlkDevInit( )

NAME

scsiBlkDevInit( ) - initialize fields in a SCSI logical partition

SYNOPSIS

```
VOID scsiBlkDevInit (pScsiBlkDev, blksPerTrack, nHeads)
    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 that are required by some file systems (e.g., dosFs). It should be called after a SCSI_BLK_DEV structure is created via *scsiBlkDevCreate*( ), but before any file system initialization routine. It is generally required only for removable media devices.

RETURNS

N/A

SEE ALSO

scsiLib

scsiBusReset( )

NAME

scsiBusReset( ) - pulse the reset signal on the SCSI bus

SYNOPSIS

```
STATUS scsiBusReset (pScsiCtrl)
    SCSI_CTRL  *pScsiCtrl;
```

DESCRIPTION

This routine calls a controller-specific routine to perform a SCSI bus reset. If no controller is specified (*pScsiCtrl* is 0), the value in *pSysScsiCtrl* is used.

RETURNS

OK if a controller-specific routine exists, or ERROR otherwise.

SEE ALSO

scsiLib

scsiTestUnitRdy( )

NAME

scsiTestUnitRdy( ) - issue a TEST_UNIT_READY command to a SCSI device.

SYNOPSIS

```
STATUS scsiTestUnitRdy (pScsiPhysDev)
    SCSI_PHYS_DEV  *pScsiPhysDev;  /* ptr to SCSI physical device */
```

RETURNS

OK, or ERROR if the routine is not successful.

SEE ALSO

scsiLib

scsiFormatUnit( )

NAME

scsiFormatUnit( ) - issue a FORMAT_UNIT command to a SCSI device

SYNOPSIS

```
STATUS scsiFormatUnit (pScsiPhysDev, cmpDefectList, defListFormat, vendorUnique,
                                      interleave, buffer, bufLength)
    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 */
```

RETURNS

OK, or ERROR if the routine is not successful.

SEE ALSO

scsiLib

scsiInquiry( )

NAME

scsiInquiry( ) - issue an INQUIRY command to a SCSI device

SYNOPSIS

```
STATUS scsiInquiry (pScsiPhysDev, buffer, bufLength)
    SCSI_PHYS_DEV   *pScsiPhysDev;   /* ptr to SCSI physical device */
    char            *buffer;         /* ptr to input data buffer */
    int             bufLength;       /* length of buffer in bytes */
```

RETURNS

OK, or ERROR if the routine is not successful.

SEE ALSO

scsiLib

scsiModeSelect( )

NAME

scsiModeSelect( ) - issue a MODE SELECT command to a SCSI device

SYNOPSIS

```
STATUS scsiModeSelect (pScsiPhysDev, pageFormat, saveParams, buffer, bufLength)
    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 */
```

RETURNS

OK, or ERROR if the routine is not successful.

SEE ALSO

scsiLib

scsiModeSense( )

NAME

scsiModeSense ( ) - issue a MODE SENSE command to a SCSI device

SYNOPSIS

```
STATUS scsiModeSense (pScsiPhysDev, pageControl, pageCode, buffer, bufLength)
    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 */
```

RETURNS

OK, or ERROR if the routine is not successful.

SEE ALSO

scsiLib

scsiReadCapacity( )

NAME

scsiReadCapacity( ) - issue a READ_CAPACITY command to a SCSI device

SYNOPSIS

```
STATUS scsiReadCapacity (pScsiPhysDev, pLastLBA, pBlkLength)
    SCSI_PHYS_DEV  *pScsiPhysDev;  /* ptr to SCSI physical device */
    int            *pLastLBA;      /* where to return last logical block address
    int            *pBlkLength;    /* where to return block length */
```

RETURNS

OK, or ERROR if the routine is not successful.

SEE ALSO

scsiLib

scsiRdSecs( )

NAME

scsiRdSecs( ) - read sectors from an SCSI block device

SYNOPSIS

```
STATUS scsiRdSecs (pScsiBlkDev, sector, numSecs, buffer)
    SCSI_BLK_DEV  *pScsiBlkDev;  /* ptr to SCSI block device info */
    int           sector;        /* sector number to be read */
    int           numSecs;       /* sector number to be read */
    char          *buffer;       /* ptr to input data buffer */
```

DESCRIPTION

This routine reads the specified physical sectors from the specified physical device.

RETURNS

OK, or ERROR if the routine is not successful.

SEE ALSO

scsiLib

scsiWrtSecs( )

NAME

scsiWrtSecs( ) - write sectors to an SCSI block device

SYNOPSIS

```
STATUS scsiWrtSecs (pScsiBlkDev, sector, numSecs, buffer)
    SCSI_BLK_DEV  *pScsiBlkDev;  /* ptr to SCSI block device info */
    int           sector;        /* sector number to be read */
    int           numSecs;       /* sector number to be read */
    char          *buffer;       /* ptr to input data buffer */
```

DESCRIPTION

This routine writes the specified physical sectors to the specified physical device.

RETURNS

OK, or ERROR if the routine is not successful.

SEE ALSO

scsiLib

scsiReqSense( )

NAME

scsiReqSense( ) - issue a REQUEST-SENSE command to a device and read the results

SYNOPSIS

```
STATUS scsiReqSense (pScsiPhysDev, buffer, bufLength)
    SCSI_PHYS_DEV  *pScsiPhysDev;  /* ptr to SCSI physical device */
    char           *buffer;        /* ptr to input data buffer */
    int            bufLength;      /* length of buffer in bytes */
```

RETURNS

OK, or ERROR if the routine is not successful.

SEE ALSO

scsiLib

scsiIoctl( )

NAME

scsiIoctl( ) - perform a device-specific control function

SYNOPSIS

```
STATUS scsiIoctl (pScsiPhysDev, function, arg)
    SCSI_PHYS_DEV  *pScsiPhysDev;  /* ptr to SCSI block device info */
    int            function;       /* function code */
    int            arg;            /* argument to pass called function */
```

RETURNS

The status of the request, or ERROR if the request is unsupported.

SEE ALSO

scsiLib

selectLib

NAME

selectLib - UNIX BSD 4.3 select library

SYNOPSIS

selectInit( ) - initialize the *select*( ) library
select( ) - pend on a set of file descriptors
selWakeup( ) - wake up a task pended in *select*( )
selWakeupAll( ) - wake up all tasks on a *select*( ) wake-up list
selNodeAdd( ) - add a wake-up node to the *select*( ) wake-up list
selNodeDelete( ) - find and delete a node on a wake-up list
selWakeupListInit( ) - initialize a *select*( ) wake-up list
selWakeupListLen( ) - get the number of nodes on a *select*( ) wake-up list
selWakeupType( ) - get the type of the given SEL_WAKEUP_NODE

```
VOID selectInit ()
int select (width, pReadFds, pWriteFds, pExceptFds, pTimeOut)
VOID selWakeup (pWakeupNode)
VOID selWakeupAll (pWakeupList, type)
STATUS selNodeAdd (pWakeupList, pWakeupNode)
STATUS selNodeDelete (pWakeupList, pWakeupNode)
VOID selWakeupListInit (pWakeupList)
int selWakeupListLen (pWakeupList)
SELECT_TYPE selWakeupType (pWakeupNode)
```

DESCRIPTION

This library provides a BSD 4.3 compatible select facility to wait for activity on a set of file descriptors. The *select*( ) routine uses a set of routines that permits tasks pended on device activity to be detected by the device's driver. Thus, the driver's interrupt service routine directly wakes up such tasks, eliminating the need for polling.

Applications can use *select*( ) with pipes and serial devices in addition to sockets. Also, *select*( ) examines *write* file descriptors in addition to *read* file descriptors; however, exception file descriptors are still unsupported.

Typically, application developers need only concern themselves 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.

SEE ALSO

Programmer's Guide: I/O System

selectInit()

NAME

selectInit( ) - initialize the *select*( ) library

SYNOPSIS

```
VOID selectInit ()
```

DESCRIPTION

This routine initializes the *select*( ) library. It should be called only once, and is typically 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

select( )

NAME

select( ) - pend on a set of file descriptors

SYNOPSIS

```
int select (width, pReadFds, pWriteFds, pExceptFds, pTimeOut)
    int            width;        /* number of bits to examine from 0 */
    fd_set         *pReadFds;    /* read file descriptors */
    fd_set         *pWriteFds;   /* write file descriptors */
    fd_set         *pExceptFds;  /* exception file descriptors */
    struct timeval *pTimeOut;    /* maximum time to wait for activity;
                                  * NULL to wait 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 fd sets in which each bit corresponds to a particular file descriptor. Bits set in the read fd set (*pReadFds*) will cause *select*( ) to pend until data is available on the any of the corresponding file descriptors, while bits set in the write fd set (*pWriteFds*) will cause *select*( ) to pend until any of the corresponding file descriptors become writable. (*pExceptFds* is currently unused, but is provided for UNIX call compatibility).

The following macros are available for setting the appropriate bits in the fd 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 fd sets, and should be set to either the maximum fd value in use, or simply FD_SETSIZE. When *select*( ) returns, it zeros out the fd 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 fd sets will be polled, and the results returned immediately. If the effective time value is greater than zero, *select*( ) will return after the given 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 are still unsupported.

Driver developers should consult the "I/O System" chapter of the *Vx960 Programmer's Guide* 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, *Programmer's Guide: I/O System*

selWakeup( )

NAME

selWakeup( ) - wake up a task pended in *select*( )

SYNOPSIS

```
VOID selWakeup (pWakeupNode)
    SEL_WAKEUP_NODE  *pWakeupNode;  /* node to awaken */
```

DESCRIPTION

After a driver's FIOSELECT function installs a wake-up node in a device's wake-up list (with *selNodeAdd*( )), it should check to see if the device is actually ready. If so, it should call *selWakeup*( ) to ensure that the *select*( ) call does not pend.

RETURNS

N/A

SEE ALSO

selectLib

selWakeupAll( )

NAME

selWakeupAll( ) - wake up all tasks on a *select*( ) wake-up list

SYNOPSIS

```
VOID selWakeupAll (pWakeupList, type)
    SEL_WAKEUP_LIST  *pWakeupList;  /* list of tasks to wakeup */
    SELECT_TYPE      type;          /* wakeup readers (SELREAD),
                                     * or writers (SELWRITE)  */
```

DESCRIPTION

This routine is called by a driver when a device becomes ready. It wakes up all tasks pended in *select*( ) that are waiting for this device.

RETURNS

N/A

SEE ALSO

selectLib

selNodeAdd( )

NAME

selNodeAdd( ) - add a wake-up node to the *select*( ) wake-up list

SYNOPSIS

```
STATUS selNodeAdd (pWakeupList, pWakeupNode)
    SEL_WAKEUP_LIST  *pWakeupList;
    SEL_WAKEUP_NODE  *pWakeupNode;
```

DESCRIPTION

This routine should be called from a driver's FIOSELECT function to add a wake-up node to a device's wake-up list.

RETURNS

OK, or ERROR if memory is insufficient.

SEE ALSO

selectLib

selNodeDelete( )

NAME

selNodeDelete( ) - find and delete a node on a wake-up list

SYNOPSIS

```
STATUS selNodeDelete (pWakeupList, pWakeupNode)
    SEL_WAKEUP_LIST  *pWakeupList;
    SEL_WAKEUP_NODE  *pWakeupNode;
```

DESCRIPTION

This routine deletes the specified wake-up node from the given wake-up list. It is typically 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

selWakeupListInit( )

NAME

selWakeupListInit( ) - initialize a *select( )* wake-up list

SYNOPSIS

```
VOID selWakeupListInit (pWakeupList)
    SEL_WAKEUP_LIST  *pWakeupList;
```

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 on a *select*( ) wake-up list

SYNOPSIS

```
int selWakeupListLen (pWakeupList)
    SEL_WAKEUP_LIST  *pWakeupList;
```

DESCRIPTION

This routine 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 on a *select*( ) wake-up list, or ERROR otherwise.

SEE ALSO

selectLib

selWakeupType( )

NAME

selWakeupType( ) - get the type of the given SEL_WAKEUP_NODE

SYNOPSIS

```
SELECT_TYPE selWakeupType (pWakeupNode)
    SEL_WAKEUP_NODE  *pWakeupNode;
```

DESCRIPTION

This routine 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

semBLib

NAME

semBLib - binary semaphore library

SYNOPSIS

semBCreate( ) - create and initialize a binary semaphore

```
SEM_ID semBCreate (options, initialState)
```

DESCRIPTION

This library provides the interface to Vx960 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 Vx960 facilities.

A binary semaphore can be viewed as a cell in memory whose contents are in one of two states, full or empty. When a task takes a binary semaphore, using *semTake*( ), subsequent action depends on the state of the semaphore:

(1) If the semaphore is full, the semaphore is made empty, and the calling task continues executing.

(2) If the semaphore is empty, the task will be blocked, pending the availability of the semaphore. If a timeout is specified and the timeout expires, the pended task will be removed from the queue of pended tasks and enter the ready state with an ERROR status. A pended task is ineligible for CPU allocation. Any number of tasks may be pended simultaneously on the same binary semaphore.

When a task gives a binary semaphore, using *semGive*( ), the next available task in the pend queue is unblocked. If no task is pending on this semaphore, the semaphore becomes full. Note that if a semaphore is given, and a task is unblocked that is of higher priority than the task that called *semGive*( ), the unblocked task will preempt the calling task.

MUTUAL EXCLUSION

To use a binary semaphore as a means of mutual exclusion, first create it with an initial state of full. For example:

```
SEM_ID semMutex;

/* create a binary semaphore that is initially full *
semMutex = semBCreate (SEM_Q_PRIORITY, SEM_FULL);
```

Then guard any critical section or resource by taking the semaphore via *semTake*( ), and exit the section or release the resource by giving the semaphore via *semGive*( ). For example:

```
semTake (semMutex, WAIT_FOREVER);
.
.    /* critical region, only accessible by a single 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 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 will block by taking a semaphore at a synchronization point and remain 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 via *semTake*( ), and an interrupt service routine can unblock that task via *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. Task1() 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, task1() 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 to occur *
...     /* 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.

SEE ALSO

semLib, **semCLib**, **semMLib**, *Programmer's Guide: Basic OS*

semBCreate( )

NAME

semBCreate( ) - create and initialize a binary semaphore

SYNOPSIS

```
SEM_ID semBCreate (options, initialState)
    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 and SEM_Q_FIFO respectively.

RETURNS

The semaphore ID, or NULL if memory cannot be allocated.

SEE ALSO

semBLib

semCLib

NAME

semCLib - counting semaphore library

SYNOPSIS

semCCreate( ) - create and initialize a counting semaphore

```
SEM_ID semCCreate (options, initialCount)
```

DESCRIPTION

This library provides the interface to Vx960 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.

SEE ALSO

semLib, **semBLib**, **semMLib**, *Programmer's Guide: Basic OS*

semCCreate( )

NAME

semCCreate( ) - create and initialize a counting semaphore

SYNOPSIS

```
SEM_ID semCCreate (options, initialCount)
    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 and SEM_Q_FIFO respectively.

RETURNS

The semaphore ID, or NULL if memory cannot be allocated.

SEE ALSO

semCLib

semLib

NAME

semLib - general semaphore library

SYNOPSIS

semGive( ) - give a semaphore
semTake( ) - take a semaphore
semFlush( ) - unblock every task pended on a semaphore
semDelete( ) - delete a semaphore
semInfo( ) - get list of task IDs that are blocked on semaphore

```
STATUS semGive (semId)
STATUS semTake (semId, timeout)
STATUS semFlush (semId)
STATUS semDelete (semId)
int semInfo (semId, idList, maxTasks)
```

DESCRIPTION

Semaphores are the basis for synchronization and mutual exclusion in Vx960. They are powerful in their simplicity and form the foundation for numerous Vx960 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 Vx960 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 type:

semBLib - binary semaphores
semCLib - counting semaphores
semMLib - mutual exclusion 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 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 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 will unblock and return ERROR to all pended tasks. Care must be taken when deleting semaphores, particularly when used for mutual exclusion, to avoid pulling the rug out from under a task which has already taken the 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 FILE

semLib.h

SEE ALSO

taskLib, semBLib, semCLib, semMLib, *Programmer's Guide: Basic OS*

semGive( )

NAME

semGive( ) - give a semaphore

SYNOPSIS

```
STATUS semGive (semId)
    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 semaphore type being used.

RETURNS

OK, or ERROR if the semaphore ID is invalid.

SEE ALSO

semLib, semBLib, semCLib, semMLib

semTake( )

NAME

semTake( ) - take a semaphore

SYNOPSIS

```
STATUS semTake (semId, timeout)
    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 semaphore type being used.

A timeout in ticks may be specified. If a task times out, *semTake*( ) will return ERROR. Timeouts of WAIT_FOREVER and NO_WAIT indicate to wait indefinitely or not to wait at all.

semTake( ) is not callable from interrupt service routines.

RETURNS

OK, or ERROR if the semaphore ID is invalid, or the task timed out.

SEE ALSO

semLib, semBLib, semCLib, semMLib

semFlush( )

NAME

semFlush( ) - unblock every task pended on a semaphore

SYNOPSIS

```
STATUS semFlush (semId)
    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.

SEE ALSO

semLib, semBLib, semCLib, semMLib

semDelete( )

NAME

semDelete( ) - delete a semaphore

SYNOPSIS

```
STATUS semDelete (semId)
    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.

RETURNS

OK, or ERROR if the semaphore ID is invalid.

SEE ALSO

semLib, semBLib, semCLib, semMLib

semInfo( )

NAME

semInfo( ) - get list of task IDs that are blocked on semaphore

SYNOPSIS

```
int semInfo (semId, idList, maxTasks)
    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 that are 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 the tasks are still valid or that no new tasks have blocked by the time *semInfo*( ) returns.

RETURNS

The number of blocked tasks placed in *idList*.

SEE ALSO

semLib

semMLib

NAME

semMLib - mutual-exclusion semaphore library

SYNOPSIS

semMCreate( ) - create and initialize a mutual-exclusion semaphore
semMGiveForce( ) - give a mutual-exclusion semaphore without restrictions

```
SEM_ID semMCreate (options)
STATUS semMGiveForce (semId)
```

DESCRIPTION

This library provides the interface to Vx960 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 Vx960 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 made possible by the fact that 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. 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 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.

TASK-DELETION SAFETY

If the option SEM_DELETE_SAFE is selected, the task owning the semaphore will be protected from deletion as long as it owns the semaphore. This solves another problem endemic to mutual exclusion. Deleting a task executing in a critical region can be catastrophic. The resource could be left in a corrupted state and the semaphore guarding the resource would be unavailable, effectively shutting off all access to the resource.

As discussed in **taskLib**, the primitives *taskSafe*( ) and *taskUnsafe*( ) offer one solution, but as this type of protection goes hand in hand with mutual exclusion, the mutual-exclusion semaphore provides the option SEM_DELETE_SAFE, which enables an implicit *taskSafe*( ) with each *semTake*( ), and a *taskUnsafe*( ) with each *semGive*( ). This convenience is also more efficient, as the resulting code requires fewer entrances to the kernel.

CAVEATS

There is no mechanism to give back or reclaim semaphores automatically when tasks are suspended or deleted. Such a mechanism, though desirable, is not currently feasible. Without explicit knowledge of the state of the guarded resource or region, reckless automatic reclamation of a semaphore could leave the resource in a partial state. Thus if a task ceases execution unexpectedly, as with a bus error, currently owned semaphores will not be given back, effectively leaving a resource permanently unavailable. The SEM_DELETE_SAFE option partially protects an application, to the extent that unexpected deletions will be deferred until the resource is released.

SEE ALSO

semLib, semBLib, semCLib, *Programmer's Guide: Basic OS*

semMCreate( )

NAME

semMCreate( ) - create and initialize a mutual-exclusion semaphore

SYNOPSIS

```
SEM_ID semMCreate (options)
    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
- 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 *semTake*( ), and an implicit *taskUnsafe*( ) for each *semGive*( ).

SEM_INVERSION_SAFE
- 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 (semId)
    SEM_ID  semId;  /* semaphore ID to give */
```

DESCRIPTION

This routine gives a mutual-exclusion semaphore not owned by the calling task. 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 only be used as a debugging aid, when the condition of the semaphore is known. It can only be called by a task that does not own the semaphore.

RETURNS

OK, or ERROR if the semaphore ID is invalid, or the routine is called by the task owning the semaphore.

SEE ALSO

semMLib, *semGive*( )

semOLib

NAME

semOLib - version 4.x binary semaphore library

SYNOPSIS

semCreate( ) - create and initialize a version 4.x binary semaphore
semInit( ) - initialize a static binary semaphore
semClear( ) - take a version 4.x semaphore if the semaphore is available

```
SEM_ID semCreate ()
STATUS semInit (pSemaphore)
STATUS semClear (semId)
```

DESCRIPTION

This library is provided for backward compatibility with Vx960 version 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.

A fuller discussion of the behavior of binary semaphores may be found in **semBLib**.

SEE ALSO

semLib, **semBLib**, *Programmer's Guide: Basic OS*

semCreate( )

NAME

semCreate( ) - create and initialize a version 4.x binary semaphore

SYNOPSIS

```
SEM_ID semCreate ()
```

DESCRIPTION

This routine allocates a version 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*( )

semInit( )

NAME

semInit( ) - initialize a static binary semaphore

SYNOPSIS

```
STATUS semInit (pSemaphore)
    SEMAPHORE  *pSemaphore;  /* 4.x semaphore to initialize */
```

DESCRIPTION

In some instances, a semaphore cannot be created with *semCreate*( ) but is a static object. This routine will initialize static version 4.x semaphores.

RETURNS

OK, or ERROR if the semaphore cannot be initialized.

SEE ALSO

semOLib, *semCreate*( )

semClear( )

NAME

semClear( ) - take a version 4.x semaphore if the semaphore is available

SYNOPSIS

```
STATUS semClear (semId)
    SEM_ID  semId;  /* semaphore ID to empty */
```

DESCRIPTION

This routine takes a 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

shellLib

NAME

shellLib - shell execution routines

SYNOPSIS

shellInit( ) - start the shell
shell( ) - the shell entry point
shellScriptAbort( ) - signal shell to stop processing a script
shellHistory( ) - display (or set) shell history
shellPromptSet( ) - change the shell prompt
shellOrigStdSet( ) - set the shell's default input/output/error fds
shellLock( ) - lock access to shell

```
STATUS shellInit (stackSize, arg)
VOID shell (interactive)
VOID shellScriptAbort ()
VOID shellHistory (size)
VOID shellPromptSet (newPrompt)
VOID shellOrigStdSet (which, fd)
BOOL shellLock (request)
```

DESCRIPTION

This module contains the execution support routines for the Vx960 shell. It provides the basic programmer's interface to Vx960. It is a C-expression interpreter, containing no built-in commands.

The nature, use, and syntax of the shell is more fully described in the "Shell" chapter of the *Vx960 Programmer's Guide.*

SEE ALSO

ledLib, *Programmer's Guide: Shell*

shellInit( )

NAME

shellInit( ) - start the shell

SYNOPSIS

```
STATUS shellInit (stackSize, arg)
    int  stackSize;  /* shell stack (0 = previous/default value) */
    int  arg;        /* argument to shell task */
```

DESCRIPTION

This routine starts the shell task. If INCLUDE_SHELL is defined in **configAll.h**, this is done by the root task, *usrRoot*( ), in **usrConfig.c**.

RETURNS

OK, or ERROR.

SEE ALSO

shellLib

shell( )

NAME

shell( ) - the shell entry point

SYNOPSIS

```
VOID shell (interactive)
    BOOL  interactive;  /* should be TRUE, except for a script */
```

DESCRIPTION

This routine is the shell task. It is started with a single parameter that indicates whether this is an interactive shell to be used from a terminal or a socket, or a shell that executes a script.

Normally, the shell is spawned in interactive mode by the root task, *usrRoot*( ), when Vx960 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

shellScriptAbort( )

NAME

shellScriptAbort( ) - signal shell to stop processing a script

SYNOPSIS

```
VOID shellScriptAbort ()
```

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

shellHistory( )

NAME

shellHistory( ) - display (or set) shell history

SYNOPSIS

```
VOID shellHistory (size)
    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*( )

shellPromptSet( )

NAME

shellPromptSet( ) - change the shell prompt

SYNOPSIS

```
VOID shellPromptSet (newPrompt)
    char *newPrompt; /* string to become new shell prompt */
```

DESCRIPTION

This routine changes the shell prompt string to *newPrompt*.

RETURNS

N/A

SEE ALSO

shellLib

shellOrigStdSet( )

NAME

shellOrigStdSet( ) - set the shell's default input/output/error fds

SYNOPSIS

```
VOID shellOrigStdSet (which, fd)
    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 fd. 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**. Any fd for a file or device can be used for *fd*.

RETURNS

N/A

SEE ALSO

shellLib

shellLock( )

NAME

shellLock( ) - lock access to shell

SYNOPSIS

```
BOOL shellLock (request)
    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

When *request* is "lock", the return will be TRUE if the call locks the shell, or FALSE if the call fails. When *request* is "unlock", the return will be TRUE if the call unlocks the shell, or FALSE if the call fails.

SEE ALSO

shellLib

sigLib

NAME

sigLib - software signal facility library

SYNOPSIS

sigInit( ) - initialize the signal facilities
sigvec( ) - install a signal handler
sigstack( ) - install a separate signal stack
sigsetmask( ) - set the signal mask
sigblock( ) - add to set of blocked signals
pause( ) - sleep until the occurrence of a signal
kill( ) - send a signal to a task
sigRaise( ) - send a signal to a task

```
STATUS sigInit ()
STATUS sigvec (sig, pVec, pOvec)
STATUS sigstack (pSs, pOss)
int sigsetmask (mask)
int sigblock (mask)
STATUS pause ()
STATUS kill (tid, signal)
STATUS sigRaise (tid, signal, code)
```

DESCRIPTION

This library provides a UNIX BSD 4.3-compatible software signal facility. 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.

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*( ) or *sigRaise*( ), which is analogous to an interrupt or hardware exception. A signal handler is bound to a particular signal with *sigvec*( ) 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 ***sigblock*( )** and ***sigsetmask*( )**, 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.

Signal handlers are passed several parameters and should be defined as:

```
VOID sigHandler (sig, code, sigContext)
    int sig;    /* signal number *
    int code;    /* additional code *
    SIGCONTEXT *sigContext; /* context of task before signal *
    {
        . . .
    }
```

The parameter *code* distinguishes signal variants. For example, both numeric overflow and zero divide raise SIGFPE (floating-point exception) but have different values for *code.*

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 and its onstack flag. If a user-defined handler is not installed for the given signal, the default action is to log a message to the console and suspend the task.

Of the 31 signals defined by UNIX BSD 4.3, only a few are meaningful in the Vx960 environment. The following can be raised by Vx960:

i960 version:

Signal	Code	Exception
SIGBUS	BUS_BUSERR	address error or parity error (NMI)
SIGBUS	NULL	sigsegv (NMI)
SIGILL	ILL_INVALID_OPCODE	invalid opcode
SIGILL	ILL_UNIMPLEMENTED	unimplemented instruction
SIGILL	BUS_ALIGN	unaligned instruction (80960CA only)
SIGILL	ILL_INVALID_OPERAND	invalid instruction operand
SIGILL	ILL_PRIVVIO_FAULT	constraint privileged instruction
SIGILL	ILL_CONSTR_RANGE_FAULT	constraint range fault
SIGILL	ILL_PROT_LENGTH	protection length fault

SIGILL	ILL_TYPE_MISMATCH	type mismatch fault
SIGTRAP	FST_INSTRUCTION_TRACE	instruction trace
SIGTRAP	FST_BRANCH_TRACE	branch trace
SIGTRAP	FST_CALL_TRACE	call trace
SIGTRAP	FST_RETURN_TRACE	return trace
SIGTRAP	FST_PRERETURN_TRACE	prereturn trace
SIGTRAP	FST_SUPERVISOR_TRACE	supervisor trace
SIGTRAP	FST_BREAKPOINT_TRACE	breakpoint trace
SIGFPE	FPE_INTDIV_TRAP	zero divide
SIGFPE	FPE_INTOVF_TRAP	integer overflow

80960SB/80960KB only:

Signal	Code	Exception
SIGFPE	FPE_FLTOVF_TRAP	floating overflow
SIGFPE	FPE_FLTUND_TRAP	floating underflow
SIGFPE	FPE_FLTINV_TRAP	floating invalid operation
SIGFPE	FPE_FLTDIV_TRAP	floating zero divide
SIGFPE	FPE_FLTINEX_TRAP	floating inexact
SIGFPE	FPE_FLTOPERR_TRAP	floating reserved encoding

Other signals can be used by a Vx960 application.

CAVEATS

If a signal is directed at a task that is pended, the signal handler will not be invoked until the task becomes runnable.

INCLUDE FILE

sigLib.h

SEE ALSO

intLib, UNIX BSD 4.3 documentation

sigInit( )

NAME

sigInit( ) - initialize the signal facilities

SYNOPSIS

```
STATUS sigInit ()
```

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 create/delete hooks could not be installed.

SEE ALSO

sigLib

sigvec( )

NAME

sigvec( ) - install a signal handler

SYNOPSIS

```
STATUS sigvec (sig, pVec, pOvec)
    int      sig;      /* signal to which the handler will be attached */
    SIGVEC   *pVec;    /* handler information if != NULL */
    SIGVEC   *pOvec;   /* previous handler info is copyed here if != NULL */
```

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*( ) will copy current signal handler information for *sig* to *pOvec* and install no signal handler if *pVec* is set to null (0).

Both *pVec* and *pOvec* are pointers to a structure of type SIGVEC. The information passed includes not only the signal handler routine, but also the signal mask and additional option bits. SIGVEC and the available options are defined in **sigLib.h**.

RETURNS

OK, or ERROR if the signal number is invalid.

SEE ALSO

sigLib

sigstack( )

NAME

sigstack( ) - install a separate signal stack

SYNOPSIS

```
STATUS sigstack (pSs, pOss)
    SIGSTACK  *pSs;   /* new signal stack info if != NULL */
    SIGSTACK  *pOss;  /* copy old signal stack info here if != NULL */
```

DESCRIPTION

This routine specifies an alternate stack, to be used for the duration of signal handling. When a signal handler is installed with *sigvec*( ), the *sv_flags* element of the SIGVEC structure must have the SV_ONSTACK bit turned on for the signal stack to be switched in.

RETURNS

OK (always).

SEE ALSO

sigLib

sigsetmask( )

NAME

sigsetmask( ) - set the signal mask

SYNOPSIS

```
int sigsetmask (mask)
    int  mask;  /* new signal mask */
```

DESCRIPTION

This routine sets the calling task's signal mask to the given value. A one (1) in the bit mask indicates that the given signal is blocked from delivery. Use the macro SIGMASK to construct the mask for a given signal number.

RETURNS

The previous value of the signal mask.

SEE ALSO

sigLib, *sigblock*( )

sigblock( )

NAME

sigblock( ) - add to set of blocked signals

SYNOPSIS

```
int sigblock (mask)
    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 given signal is blocked from delivery. Use the macro SIGMASK to construct the mask for a given signal number.

RETURNS

The previous value of the signal mask.

SEE ALSO

sigLib, *sigsetmask*( )

pause( )

NAME

pause( ) - sleep until the occurrence of a signal

SYNOPSIS

```
STATUS pause ()
```

DESCRIPTION

This routine blocks task execution until the occurrence of a signal. After the signal handler has executed, *pause*( ) returns.

RETURNS

ERROR (always).

SEE ALSO

sigLib

kill( )

NAME

kill( ) - send a signal to a task

SYNOPSIS

```
STATUS kill (tid, signal)
    int  tid;     /* task to which the signal is directed */
    int  signal;  /* the signal to send the task */
```

DESCRIPTION

This routine sends a signal to the specified task. The task must have previously installed a signal handler with *sigvec*( ).

CAVEAT

If the task is pended, the signal will not be delivered until the task is ready to run.

RETURNS

OK, or ERROR if the task is not found, the task has no signal handler for *signal,* or the signal cannot be delivered.

SEE ALSO

sigLib, *sigvec*( ), *sigRaise*( )

sigRaise( )

NAME

sigRaise( ) - send a signal to a task

SYNOPSIS

```
STATUS sigRaise (tid, signal, code)
    int  tid;     /* task to which the signal is directed */
    int  signal;  /* the signal to send the task */
    int  code;    /* additional code */
```

DESCRIPTION

This routine sends a signal to the specified task. It provides the same mechanism as *kill*( ), but allows an additional code identifying signal variants to be passed to the signal handler.

RETURNS

OK, or ERROR if the task is not found, the task has no signal handler for *signal,* or the signal cannot be delivered.

SEE ALSO

sigLib, *sigvec*( ), *kill*( )

sockLib

NAME

sockLib - UNIX BSD 4.3 compatible socket library

SYNOPSIS

socket( ) - open a socket
bind( ) - bind a name to a socket
listen( ) - enable connections to a socket
accept( ) - accept a connection from a socket
connect( ) - initiate a connection to a socket
sendto( ) - send a message to a socket
send( ) - send data to a socket
sendmsg( ) - send a message to a socket
recvfrom( ) - receive a message from a socket
recv( ) - receive data from a socket
recvmsg( ) - receive a message from a socket
setsockopt( ) - set socket options
getsockopt( ) - get socket options
getsockname( ) - get a socket name
getpeername( ) - get the name of a connected peer
shutdown( ) - shut down a network connection

```
int socket (domain, type, protocol)
STATUS bind (s, name, namelen)
STATUS listen (s, backlog)
int accept (s, addr, addrlen)
STATUS connect (s, name, namelen)
int sendto (s, buf, bufLen, flags, to, tolen)
int send (s, buf, bufLen, flags)
int sendmsg (sd, mp, flags)
int recvfrom (s, buf, bufLen, flags, from, pFromLen)
int recv (s, buf, bufLen, flags)
int recvmsg (sd, mp, flags)
STATUS setsockopt (s, level, optname, optval, optlen)
STATUS getsockopt (s, level, optname, optval, optlen)
STATUS getsockname (s, name, namelen)
STATUS getpeername (s, name, namelen)
STATUS shutdown (s, how)
```

DESCRIPTION

This library provides UNIX BSD 4.3 compatible socket calls. These calls may be used to open, close, read, and write sockets, either on the same CPU or over a network. The calling sequences of these routines are identical to

UNIX BSD 4.3.

INCLUDE FILES

types.h, mbuf.h, socket.h, socketvar.h

SEE ALSO

UNIX BSD 4.3 documentation, **netLib** *Programmer's Guide: Network*

socket( )

NAME

socket( ) - open a socket

SYNOPSIS

```
int socket (domain, type, protocol)
    int  domain;    /* address family (e.g., AF_INET) */
    int  type;      /* socket type (e.g., SOCK_STREAM) */
    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.

RETURNS

A socket descriptor, or ERROR.

SEE ALSO

sockLib

bind( )

NAME

bind( ) - bind a name to a socket

SYNOPSIS

```
STATUS bind (s, name, namelen)
    int              s;        /* socket descriptor */
    struct sockaddr  *name;    /* name to be bound */
    int              namelen;  /* length of name */
```

DESCRIPTION

This routine associates a network address (alco referred to as its "name") with a given 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

listen( )

NAME

listen( ) - enable connections to a socket

SYNOPSIS

```
STATUS listen (s, backlog)
    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

accept( )

NAME

accept( ) - accept a connection from a socket

SYNOPSIS

```
int accept (s, addr, addrlen)
    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.

RETURNS

A socket descriptor, or ERROR if the call fails.

SEE ALSO

sockLib

connect( )

NAME

connect( ) - initiate a connection to a socket

SYNOPSIS

```
STATUS connect (s, name, namelen)
    int             s;         /* socket descriptor */
    struct sockaddr *name;     /* address of the 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 the type SOCK_DGRAM, it permanently specifies the peer to which messages are sent. The *name* specifies the address of the other socket.

RETURNS

OK if the connection is successful, or ERROR if the call fails.

SEE ALSO

sockLib

sendto( )

NAME

sendto( ) - send a message to a socket

SYNOPSIS

```
int sendto (s, buf, bufLen, flags, to, tolen)
    int             s;       /* socket on which to send data */
    caddr_t         buf;     /* pointer to data buffer */
    int             bufLen;  /* length of buffer */
    int             flags;   /* flags to underlying protocols */
    struct sockaddr *to;     /* recipient's address */
    int             tolen;   /* length of to sockaddr */
```

DESCRIPTION

This routine sends a message to the datagram socket named by *to*. The socket *s* will be received by the receiver as the sending socket.

RETURNS

The number of bytes sent, or ERROR if the call fails.

SEE ALSO

sockLib

send( )

NAME

send( ) - send data to a socket

SYNOPSIS

```
int send (s, buf, bufLen, flags)
    int   s;       /* socket on which to send */
    char  *buf;    /* pointer to buffer to transmit */
    int   bufLen;  /* length of buffer */
    int   flags;   /* flags to underlying protocols */
```

DESCRIPTION

This routine transmits data on a previously established connection-based socket.

RETURNS

The number of bytes sent, or ERROR if the call fails.

SEE ALSO

sockLib

sendmsg( )

NAME

sendmsg( ) - send a message to a socket

SYNOPSIS

```
int sendmsg (sd, mp, flags)
    int            sd;     /* socket on which to transmit */
    struct msghdr  *mp;    /* scatter-gather message header */
    int            flags;  /* flags to underlying protocols */
```

DESCRIPTION

This routine sends a message to a socket. It may be used in place of *sendto*( ) to decrease the overhead of reconstructing the message-header structure (*msghdr*) for each message.

RETURNS

The number of bytes sent, or ERROR if the call fails.

SEE ALSO

sockLib

recvfrom( )

NAME

recvfrom( ) - receive a message from a socket

SYNOPSIS

```
int recvfrom (s, buf, bufLen, flags, from, pFromLen)
    int            s;          /* socket on which to receive data */
    char           *buf;       /* pointer to data buffer */
    int              bufLen;     /* length of buffer */
    int              flags;      /* flags to underlying protocols */
    struct sockaddr  *from;      /* gets filled in with sender's address */
    int              *pFromLen;  /* value/result length of 'from' */
```

DESCRIPTION

This routine receives data from a 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*.

RETURNS

The number of number of bytes received, or ERROR if the call fails.

SEE ALSO

sockLib

recv( )

NAME

recv( ) - receive data from a socket

SYNOPSIS

```
int recv (s, buf, bufLen, flags)
    int   s;       /* socket on which to receive data */
    char  *buf;    /* pointer to buffer where data will be written */
    int   bufLen;  /* length of buffer */
    int   flags;   /* flags to underlying protocols */
```

DESCRIPTION

This routine receives data from a connected socket.

RETURNS

The number of bytes received, or ERROR if the call fails.

SEE ALSO

sockLib

recvmsg( )

NAME

recvmsg( ) - receive a message from a socket

SYNOPSIS

```
int recvmsg (sd, mp, flags)
    int            sd;     /* socket on which to receive */
    struct msghdr  *mp;    /* scatter-gather message header */
    int            flags;  /* flags to underlying protocols */
```

DESCRIPTION

This routine receives message on a socket. It may be used in place of *recvfrom*( ) to decrease the overhead of breaking down the message-header structure (*msghdr*) for each message.

RETURNS

The number of bytes received, or ERROR if the call fails.

SEE ALSO

sockLib

setsockopt( )

NAME

setsockopt( ) - set socket options

SYNOPSIS

```
STATUS setsockopt (s, level, optname, optval, optlen)
    int   s;        /* target socket */
    int   level;    /* protocol level at which option resides */
    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.

RETURNS

OK, or ERROR if there is an invalid socket, an unknown option, an option length greater than MLEN, insufficient mbufs, or the routine is unable to set the specified option.

SEE ALSO

sockLib

getsockopt( )

NAME

getsockopt( ) - get socket options

SYNOPSIS

```
STATUS getsockopt (s, level, optname, optval, optlen)
    int   s;        /* socket */
    int   level;    /* protocol level for the options */
    int   optname;  /* name of the option */
    char  *optval;  /* where option is to be returned */
    int   *optlen;  /* where option length is to be returned */
```

DESCRIPTION

This routine returns relevant option values associated with a socket. To manipulate options at the "socket" level, *level* should be SOL_SOCKET. Any other levels should use the appropriate protocol number.

RETURNS

OK, or ERROR if there is an invalid socket, an unknown option, or the routine is unable to get the specified option.

SEE ALSO

sockLib

getsockname( )

NAME

getsockname( ) - get a socket name

SYNOPSIS

```
STATUS getsockname (s, name, namelen)
    int              s;         /* socket descriptor */
    struct sockaddr  *name;     /* where to return name */
    int              *namelen;  /* amount of space available in name, */
                                /* later filled in with actual size of name */
```

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 actual size of the socket name is copied to *namelen*.

RETURNS

OK, or ERROR if the socket is invalid or not connected.

SEE ALSO

sockLib

getpeername( )

NAME

getpeername( ) - get the name of a connected peer

SYNOPSIS

```
STATUS getpeername (s, name, namelen)
    int              s;          /* socket descriptor */
    struct sockaddr  *name;      /* where to return name */
    int              *namelen;   /* amount of space available in name, */
                                 /* later filled in with actual size of name */
```

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 actual size of the socket name is copied to *namelen*.

RETURNS

OK, or ERROR if the socket is invalid or not connected.

SEE ALSO

sockLib

shutdown( )

NAME

shutdown( ) - shut down a network connection

SYNOPSIS

```
STATUS shutdown (s, how)
    int  s;    /* the socket to shutdown */
    int  how;  /* 0 = receives disallowed */
               /* 1 = sends disallowed */
               /* 2 = sends and receives disallowed */
```

DESCRIPTION

This routine shuts down all, or part, of a connection-based socket *s*. If the value of *how* is 0, receives will be disallowed. If *how* is 1, sends will be disallowed. If *how* is 2, both sends and receives are disallowed.

RETURNS

OK, or ERROR if the socket is invalid or not connected.

SEE ALSO

sockLib

spyLib

NAME

spyLib - spy CPU activity library

SYNOPSIS

spyClkStart( ) - start collecting task activity data
spyClkStop( ) - stop collecting task activity data
spyReport( ) - display task activity data
spyTask( ) - run periodic task activity reports
spyStop( ) - stop spying and reporting
spy( ) - begin periodic task activity reports
spyHelp( ) - display task monitoring help menu

```
STATUS spyClkStart (intsPerSec)
VOID spyClkStop ()
VOID spyReport ()
VOID spyTask (freq)
VOID spyStop ()
VOID spy (freq, ticksPerSec)
VOID spyHelp ()
```

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.

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)
tDbxTask      _dbxTask      edba4   20     0% (       0)    0% (       0)
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
```

spyClkStart( )

NAME

spyClkStart( ) - start collecting task activity data

SYNOPSIS

```
STATUS spyClkStart (intsPerSec)
    int  intsPerSec;  /* timer interrupt frequency, */
                      /* 0 = use 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, or that rate closest to 100 ticks per second (whether faster or slower) that the auxiliary clock cn support (see the module **sysLib.c**). Data from previous collections is cleared.

For systems that cannot run the auxiliary clock at 100 interrupts per second, the default is the rate closest to 100 that the auxiliary clock can support. This value is determined by the auxiliary clock implementation in **sysLib.c**.

RETURNS

OK, or ERROR if the CPU has no auxiliary clock, or if task create and delete hooks cannot be installed.

SEE ALSO

spyLib, *sysAuxClkConnect*( )

spyClkStop( )

NAME

spyClkStop( ) - stop collecting task activity data

SYNOPSIS

```
VOID spyClkStop ()
```

DESCRIPTION

This routine disables the auxiliary clock interrupts. Data collected remains valid until the next *spyClkStart*( ) call.

RETURNS

N/A

SEE ALSO

spyLib

spyReport( )

NAME

spyReport( ) - display task activity data

SYNOPSIS

```
VOID spyReport ()
```

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 in 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

spyLib

spyTask( )

NAME

spyTask( ) - run periodic task activity reports

SYNOPSIS

```
VOID spyTask (freq)
    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

spyLib

spyStop( )

NAME

spyStop( ) - stop spying and reporting

SYNOPSIS

```
VOID spyStop ()
```

DESCRIPTION

This routine calls *spyClkStop*( ). Any periodic reporting by *spyTask*( ) is terminated.

RETURNS

N/A

SEE ALSO

spyLib

spy( )

NAME

spy( ) - begin periodic task activity reports

SYNOPSIS

```
VOID spy (freq, ticksPerSec)
    int  freq;        /* reporting frequency, in seconds */
                      /* 0 = use default of 5            */
    int  ticksPerSec; /* interrupt clock frequency       */
                      /* 0 = use 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, or a value as close to 100 as the auxiliary clock can support. This value is determined by the implementation of the auxiliary clock in **sysLib.c**.

This routine spawns *spyTask*( ) to do the actual reporting.

It is not necessary to call *spyClkStart*( ) before running *spy*( ).

RETURNS

N/A

SEE ALSO

spyLib

spyHelp( )

NAME

spyHelp( ) - display task monitoring help menu

SYNOPSIS

```
VOID spyHelp ()
```

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

spyLib

stdioLib

NAME

stdioLib - standard I/O library

SYNOPSIS

stdioInit( ) - initialize stdioLib support
isatty( ) - return whether the underlying driver is a tty device
fclose( ) - empty stream buffers and close a file
fdopen( ) - associate a stream with a file descriptor
fgetc( ) - return the next character in an input stream
fgets( ) - read a string from an input stream
fflush( ) - write out any buffers on an output stream
fopen( ) - open a stream on a file
fprintf( ) - print a formatted string to a stream
fputc( ) - append a character to an output stream
fputs( ) - copy a NULL-terminated string to an output stream
fread( ) - perform a buffered read
freopen( ) - substitute a named file in place of an open stream
fseek( ) - reposition a stream
ftell( ) - return the current offset in a stream
fwrite( ) - perform a buffered write
gets( ) - read a string from the standard input stream
getw( ) - read the next word (32-bit integer) from a stream
puts( ) - copy a NULL-terminated string to the output stream
putw( ) - append a word (32-bit integer) to an output stream
rewind( ) - position a stream at the beginning
scanf( ) - read and convert characters from the standard input stream
fscanf( ) - read and convert characters from an input stream
setbuf( ) - specify a buffer to be used on a stream
setbuffer( ) - set a buffer to be used on a stream
setlinebuf( ) - set the line buffering for either *stdout* or *stderr*
ungetc( ) - push a character back into an input stream
clearerr( ) - reset the error and end-of-file indicators
feof( ) - determine if an end-of-file has been read
ferror( ) - determine if an error has occurred while reading or writing
fileno( ) - get the file descriptor associated with a stream
getchar( ) - return the next character in the standard input stream
putchar( ) - append a character to the standard output stream
getc( ) - return the next character in an input stream
putc( ) - append a character to an output stream

```
STATUS stdioInit ()
```

```
BOOL isatty (fd)
STATUS fclose (fp)
FILE *fdopen (fd, type)
int fgetc (fp)
char *fgets (s, n, fp)
int fflush (fp)
FILE *fopen (filename, type)
int fprintf (fp, fmt, ...)
int fputc (c, fp)
int fputs (s, fp)
int fread (ptr, size, count, fp)
FILE *freopen (filename, type, fp)
STATUS fseek (fp, offset, ptrname)
long ftell (fp)
int fwrite (ptr, size, count, fp)
char *gets (s)
int getw (fp)
int puts (s)
int putw (w, fp)
VOID rewind (fp)
int scanf (fmt, ...)
int fscanf (fp, fmt, ...)
VOID setbuf (fp, buf)
VOID setbuffer (fp, buf, size)
VOID setlinebuf (fp)
int ungetc (c, fp)
VOID clearerr (fp)
int feof (fp)
BOOL ferror (fp)
int fileno (fp)
int getchar ()
int putchar (c)
int getc (fp)
int putc (c, fp)
```

DESCRIPTION

This library provides a complete UNIX compatible standard I/O buffering scheme. It is beyond the scope of this manual entry to describe all aspects of the buffering — see the *Vx960 Programmer's Guide: I/O System* and the Kernighan & Ritchie C manual. This manual entry primarily highlights the differences between the UNIX and Vx960 standard I/O.

VX_STDIO TASK OPTION

Traditionally *stdin, stdout,* and *stderr* are macros, but, in Vx960, they are in fact variables. They will only be defined for the task context of tasks spawned with the task option bit VX_STDIO. They are unique to each such task and correspond to the file descriptors 0, 1, and 2 in the basic I/O system. Their values are undefined when used at any other time, e.g., interrupt-level code or another task without the VX_STDIO option.

NOTE: If a task will use *stdin, stdout,* or *stderr,* the task must be spawned with the VX_STDIO task option bit.

The one exception is the use of the routine *printf*( ), which, as explained below, does not actually use standard I/O buffering.

FILE POINTERS

The routine *fopen*( ) creates a file pointer. Use of the file pointer follows conventional UNIX usage. In a shared address space, however, and perhaps more critically, with the Vx960 system symbol table, tasks may not use each others' file pointers, at least not without some interlocking mechanism. If it is necessary to use the same name for a file pointer but have incarnations for each task, then use task variables; see the manual entry for **taskVarLib**.

FIOLIB

Several routines normally considered part of standard I/O — ***printf***( ), ***sscanf***( ), and ***sprintf***( ) — are not implemented in **stdioLib**; 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 entire **stdioLib** package.

TASK TERMINATION

When a task exits, unlike in UNIX, 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.

INCLUDE FILES

stdioLib.h, taskLib.h

All the macros defined in **stdioLib.h** are also implemented as real functions so that they are available from the Vx960 shell.

SEE ALSO

fioLib, ioLib, taskVarLib, sigLib, Kernighan & Ritchie C manual,
Programmer's Guide: I/O System

stdioInit( )

NAME

stdioInit( ) - initialize **stdioLib** support

SYNOPSIS

```
STATUS stdioInit ()
```

DESCRIPTION

This routine must be called before using **stdioLib** buffering. If INCLUDE_STDIO is defined in **configAll.h**, it is called by the root task *usrRoot*( ) in **usrConfig.c**.

RETURNS

OK, or ERROR if there is an error installing standard I/O facilities.

SEE ALSO

stdioLib

isatty( )

NAME

isatty( ) - return whether the underlying driver is a tty device

SYNOPSIS

```
BOOL isatty (fd)
    int  fd;  /* file descriptor to check */
```

DESCRIPTION

This routine simply invokes the *ioctl*( ) function FIOISATTY on the specified fd.

RETURNS

TRUE if the driver indicates a tty device, otherwise FALSE.

SEE ALSO

stdioLib

fclose( )

NAME

fclose( ) - empty stream buffers and close a file

SYNOPSIS

```
STATUS fclose (fp)
    FILE  *fp;  /* stream */
```

DESCRIPTION

This routine empties the buffer associated with a specified stream and closes the file.

RETURNS

OK if the buffers flush, or EOF if the buffers cannot be written out or the file cannot be closed.

SEE ALSO

stdioLib, *fflush*( )

fdopen( )

NAME

fdopen( ) - associate a stream with a file descriptor

SYNOPSIS

```
FILE *fdopen (fd, type)
    int   fd;     /* already open file descriptor                 */
    char  *type;  /* mode to open file (must agree with open fd) */
```

DESCRIPTION

This routine associates a stream with the file descriptor obtained by the routines *open*( ) and *creat*( ).

The mode *type* must be included because its status cannot be queried, and it must agree with the mode of the already open fd.

RETURNS

A stream pointer, or NULL if the fd is invalid.

SEE ALSO

stdioLib, *fopen*( ), *freopen*( )

fgetc()

NAME

fgetc( ) - return the next character in an input stream

SYNOPSIS

```
int fgetc (fp)
    FILE  *fp;  /* stream */
```

DESCRIPTION

This routine returns the next character from a specified input stream as an integer. It also moves the file pointer ahead one character in the stream. This routine is the same as *getc*( ), but is a genuine routine, not a macro.

RETURNS

The next character, or EOF on either end-of-file or an error.

SEE ALSO

stdioLib

fgets( )

NAME

fgets( ) - read a string from an input stream

SYNOPSIS

```
char *fgets (s, n, fp)
    char  *s;   /* buffer to hold characters read */
    int   n;    /* number of bytes to read -1 for newline */
    FILE  *fp;  /* stream */
```

DESCRIPTION

This routine copies, at most, n-1 characters, or up to the newline character, from the stream *fp* into the buffer s. The buffer *s* is NULL-terminated.

RETURNS

The NULL-terminated string, or NULL on either end-of-file or an error.

SEE ALSO

stdioLib, *fopen*( ), *fread*( )

fflush( )

NAME

fflush( ) - write out any buffers on an output stream

SYNOPSIS

```
int fflush (fp)
    FILE  *fp;  /* stream */
```

DESCRIPTION

This routine writes the buffer associated with a specified stream. The stream remains open.

RETURNS

OK, or EOF if the the buffers cannot be written out.

SEE ALSO

stdioLib

fopen( )

NAME

fopen( ) - open a stream on a file

SYNOPSIS

```
FILE *fopen (filename, type)
    char  *filename;  /* name of file */
    char  *type;      /* mode to open file */
```

DESCRIPTION

This routine opens a specified file and associates it with a stream. The returned file pointer is used to identify the stream with subsequent operations.

The *type* is a character string consisting of:

r - read

w - write

a - update, open for writing at the end, or create the file if it does not exist.

In addition, a trailing + indicates that the file is to be opened for both reading and writing.

r+ - position at the beginning

w+ - create or truncate

a+ - position at the end

Both reads and writes may be intermixed as long as an intervening *fseek*( ) or *rewind*( ) occurs.

RETURNS

A file pointer, or NULL if the file cannot be opened.

SEE ALSO

stdioLib, *freopen*( ), *fdopen*( )

fprintf( )

NAME

fprintf( ) - print a formatted string to a stream

SYNOPSIS

```
int fprintf (fp, fmt, ...)
    FILE *  fp;   /* output stream                          */
    char *  fmt;  /* format specification                   */
```

DESCRIPTION

This routine is the same as *printf*( ), except that it copies output to a specified stream *fp*.

NOTE

The standard output file pointer *stdout* is buffered; thus *fflush*( ) must be used to force output.

RETURNS

OK, or EOF if there is an error.

SEE ALSO

stdioLib, *fopen*( )

fputc( )

NAME

fputc( ) - append a character to an output stream

SYNOPSIS

```
int fputc (c, fp)
    int   c;    /* character to append */
    FILE  *fp;  /* stream */
```

DESCRIPTION

This routine appends character *c* to a specified stream *fp*. It is the same as *putc*( ), but is a genuine routine, not a macro.

RETURNS

The character written, or EOF if there is an error.

SEE ALSO

stdioLib, *fopen*( ), *fputs*( )

fputs( )

NAME

fputs( ) - copy a NULL-terminated string to an output stream

SYNOPSIS

```
int fputs (s, fp)
    char  *s;   /* string to copy to output stream */
    FILE  *fp;  /* output stream                   */
```

DESCRIPTION

This routine copies the NULL-terminated string *s* to a specified stream *fp*.

RETURNS

The last character written, or 0 if the string is empty.

SEE ALSO

stdioLib

fread( )

NAME

fread( ) - perform a buffered read

SYNOPSIS

```
int fread (ptr, size, count, fp)
    char      *ptr;    /* where to copy into */
    unsigned  size;    /* size of item       */
    unsigned  count;   /* number of items    */
    FILE      *fp;     /* input stream       */
```

DESCRIPTION

This routine reads *count* items of data of size *size* from the stream *fp* into a block beginning at *ptr*.

RETURNS

The number of items read, or 0 on either end-of-file or an error.

SEE ALSO

stdioLib

freopen( )

NAME

freopen( ) - substitute a named file in place of an open stream

SYNOPSIS

```
FILE *freopen (filename, type, fp)
    char  *filename;   /* new file to open */
    char  *type;       /* mode to open file */
    FILE  *fp;         /* old stream        */
```

DESCRIPTION

This routine closes a stream and opens a new file in its place.

Typically *freopen*( ) is used to substitute a file in place of the standard pre-opened streams *stdin*, *stdout*, and *stderr*.

A file may be changed from unbuffered or line buffered to block buffered by using *freopen*( ). A file can be changed from block buffered or line buffered to unbuffered by using *freopen*( ) followed by *setbuf*( ) with a NULL buffer argument.

RETURNS

The original value of the stream, or NULL if the file cannot be opened.

SEE ALSO

stdioLib, *fopen*( ), *fdopen*( )

fseek( )

NAME

fseek( ) - reposition a stream

SYNOPSIS

```
STATUS fseek (fp, offset, ptrname)
    FILE  *fp;      /* stream */
    long  offset;   /* byte to seek to */
    int   ptrname;  /* type of seek (L_SET,L_INCR,L_XTND) */
```

DESCRIPTION

This routine sets the position of the next I/O operation on a specified stream. The new position is at the signed distance *offset* bytes from either the beginning, the current position, or the end-of-file, according to the value of *ptrname* (0, 1, or 2).

RETURNS

OK, or ERROR if there are any improper seeks.

SEE ALSO

stdioLib, *lseek*( ), *fopen*( )

ftell( )

NAME

ftell( ) - return the current offset in a stream

SYNOPSIS

```
long ftell (fp)
    FILE  *fp;  /* stream to report on */
```

DESCRIPTION

This routine returns the current value, in bytes, of the offset from the beginning of the file associated with the specified stream.

RETURNS

The offset, or ERROR if the stream is invalid.

SEE ALSO

stdioLib, *fseek*( ), *fopen*( )

fwrite( )

NAME

fwrite( ) - perform a buffered write

SYNOPSIS

```
int fwrite (ptr, size, count, fp)
    char      *ptr;   /* where to copy from */
    unsigned  size;   /* size of item       */
    unsigned  count;  /* number of items    */
    FILE      *fp;    /* output stream      */
```

DESCRIPTION

This routine appends, at most, *count* items of data of the size *size,* from the block beginning at *ptr* to the stream *fp*.

RETURNS

The number of items written, or 0 if there is an error.

SEE ALSO

stdioLib, *write*( ), *putc*( ), *puts*( )

gets( )

NAME

gets( ) - read a string from the standard input stream

SYNOPSIS

```
char *gets (s)
    char  *s;  /* buffer to hold characters read */
```

DESCRIPTION

This routine reads a string from standard input up to a newline character and copies it into the buffer *s*. The string in *s* is NULL-terminated.

RETURNS

The NULL-terminated string, or NULL on either end-of-file or an error.

SEE ALSO

stdioLib, *fopen*( ), *fread*( )

getw( )

NAME

getw( ) - read the next word (32-bit integer) from a stream

SYNOPSIS

```
int getw (fp)
    FILE  *fp;  /* input stream */
```

DESCRIPTION

This routine reads the next 32-bit quantity from a specified stream. It returns EOF on an 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.

RETURNS

A 32-bit number from the stream, or EOF on either end-of-file or an error.

SEE ALSO

stdioLib, *fopen*( ), *getc*( )

puts( )

NAME

puts( ) - copy a NULL-terminated string to the output stream

SYNOPSIS

```
int puts (s)
    char  *s;  /* string to copy to output */
```

DESCRIPTION

This routine is the same as *fputs*( ), but adds a newline at the end of the string.

RETURNS

The last character written (newline), or EOF.

SEE ALSO

stdioLib, *fputs*( ), *fopen*( ), *gets*( )

putw( )

NAME

putw( ) - append a word (32-bit integer) to an output stream

SYNOPSIS

```
int putw (w, fp)
    int   w;    /* word (32-bit integer) */
    FILE  *fp;  /* stream */
```

DESCRIPTION

This routine writes the 32-bit quantity *w* to a specified stream.

RETURNS

The value written.

SEE ALSO

stdioLib

rewind( )

NAME

rewind( ) - position a stream at the beginning

SYNOPSIS

```
VOID rewind (fp)
    FILE  *fp;
```

DESCRIPTION

This routine sets the position of the next I/O operation to the beginning of a specified stream. It is equivalent to:

```
    fseek (fp, 0L, 0);
```

RETURNS

N/A

SEE ALSO

stdioLib, *fseek*( ), *ftell*( ), *fopen*( )

scanf( )

NAME

scanf( ) - read and convert characters from the standard input stream

SYNOPSIS

```
int scanf (fmt, ...)
    char  *fmt;  /* the format string      */
```

DESCRIPTION

This routine reads characters from the standard input stream, interprets them according to the format in string *fmt*, and assigns values to variables pointed to by *va_alist*.

See the manual entry for *sscanf*( ) for a full description of format specification.

RETURNS

The number of items scanned, or EOF on end-of-file.

SEE ALSO

stdioLib, *fopen*( ), *fscanf*( ), *sscanf*( ), Kernighan & Ritchie C manual

fscanf( )

NAME

fscanf( ) - read and convert characters from an input stream

SYNOPSIS

```
int fscanf (fp, fmt, ...)
    FILE  *fp;   /* stream to read from     */
    char  *fmt;  /* the format string       */
```

DESCRIPTION

This routine reads characters from a stream, interprets them according to the format in string *fmt*, and assigns values to arguments starting with the one pointed to by *va_list*.

See the manual entry for *sscanf*( ) for a full description of format specification.

RETURNS

The number of items scanned, or EOF on end-of-file.

SEE ALSO

stdioLib, *fopen*( ), *sscanf*( ), Kernighan & Ritchie C manual

setbuf( )

NAME

setbuf( ) - specify a buffer to be used on a stream

SYNOPSIS

```
VOID setbuf (fp, buf)
    FILE  *fp;   /* stream */
    char  *buf;  /* buffer to be used for stream; NULL = unbuffered */
```

DESCRIPTION

This routine specifies a character array *buf* to be used on a stream in place of the automatically allocated buffer. If *buf* is NULL, the stream is unbuffered. This routine must be called before reading or writing on a stream.

The array must be of size BUFSIZE. The routine *setbuffer*( ) allows a user-specified buffer size.

RETURNS

N/A

SEE ALSO

stdioLib, *freopen*( ), *setbuffer*( )

setbuffer( )

NAME

setbuffer( ) - set a buffer to be used on a stream

SYNOPSIS

```
VOID setbuffer (fp, buf, size)
    FILE  *fp;   /* stream */
```

```
        char   *buf;   /* buffer to be used for stream; NULL = unbuffered */
        int    size;   /* size of buffer */
```

DESCRIPTION

This routine specifies a character array *buf* to be used on a stream in place of the automatically allocated buffer. If *buf* is NULL, the stream is unbuffered. This routine must be called before reading or writing on a stream. The routine *setbuf*( ) may be used if the buffer size is to be of size BUFSIZE.

RETURNS

N/A

SEE ALSO

stdioLib, *freopen*( ), *setbuf*( )

setlinebuf( )

NAME

setlinebuf( ) - set the line buffering for either *stdout* or *stderr*

SYNOPSIS

```
VOID setlinebuf (fp)
    FILE  *fp;  /* stream */
```

DESCRIPTION

This routine changes *stdout* or *stderr* from block buffered or unbuffered to line buffered. Unlike *setbuf*( ) and *setbuffer*( ), it can be used at any time the stream is active.

A stream can be changed from unbuffered or line buffered to block buffered using *freopen*( ). A stream can be changed from block or line buffered to unbuffered using *freopen*( ) followed by *setbuf*( ) with a buffer argument of NULL.

RETURNS

N/A

SEE ALSO

stdioLib, *freopen*( ), *setbuf*( )

ungetc( )

NAME

ungetc( ) - push a character back into an input stream

SYNOPSIS

```
int ungetc (c, fp)
    int   c;     /* character to push */
    FILE  *fp;   /* stream */
```

DESCRIPTION

This routine pushes *c* back into an input stream. The next call to *getc*( ) will return *c*. Only a one-character push is guaranteed.

RETURNS

The character *c*, or EOF if the character cannot be pushed back.

SEE ALSO

stdioLib, *fopen*( ), *getc*( )

clearerr( )

NAME

clearerr( ) - reset the error and end-of-file indicators

SYNOPSIS

```
VOID clearerr (fp)
    FILE  *fp;   /* stream */
```

DESCRIPTION

This routine clears the error and end-of-file indicators on a specified stream.

RETURNS

N/A

SEE ALSO

stdioLib

feof( )

NAME

feof( ) - determine if an end-of-file has been read

SYNOPSIS

```
int feof (fp)
    FILE  *fp;  /* stream */
```

DESCRIPTION

This routine returns TRUE if an end-of-file has been read from a specified input stream. The end-of-file indication remains until the stream is closed or is reset by *clearerr*( ).

RETURNS

TRUE if an end-of-file has been read, otherwise FALSE.

SEE ALSO

stdioLib, *clearerr*( ), *ferror*( )

ferror( )

NAME

ferror( ) - determine if an error has occurred while reading or writing

SYNOPSIS

```
BOOL ferror (fp)
    FILE  *fp;  /* stream */
```

DESCRIPTION

This routine returns TRUE if an error has occurred while reading from or writing to a specified stream. The error remains until the stream is closed or is reset by *clearerr*( ).

RETURNS

TRUE if an error has occurred, otherwise FALSE.

SEE ALSO

stdioLib, *clearerr*( ), *feof*( )

fileno( )

NAME

fileno( ) - get the file descriptor associated with a stream

SYNOPSIS

```
int fileno (fp)
    FILE  *fp;  /* stream */
```

DESCRIPTION

This routine returns the file descriptor associated with a specified stream.

RETURNS

The file descriptor for the specified stream.

SEE ALSO

stdioLib, *open*( )

getchar( )

NAME

getchar( ) - return the next character in the standard input stream

SYNOPSIS

```
int getchar ()
```

DESCRIPTION

This routine returns the next character in the standard input stream. It is equivalent to:

```
getc (stdin);
```

RETURNS

The next character in the standard input stream.

SEE ALSO

stdioLib

putchar( )

NAME

putchar( ) - append a character to the standard output stream

SYNOPSIS

```
int putchar (c)
    int  c;  /* character to output */
```

DESCRIPTION

This routine appends a character to the standard output stream. It is equivalent to:

```
putc (c, stdout);
```

This function replicates the macro by the same name and is useful for invocations from the shell.

RETURNS

The character written, or EOF if there is an error.

SEE ALSO

stdioLib

getc( )

NAME

getc( ) - return the next character in an input stream

SYNOPSIS

```
int getc (fp)
    FILE  *fp;  /* stream */
```

DESCRIPTION

This routine returns the next character in a specified input stream. It replicates the macro by the same name and is useful for invocations from the shell.

RETURNS

The next character, or EOF on either end-of-file or an error.

SEE ALSO

stdioLib

putc()

NAME

putc( ) - append a character to an output stream

SYNOPSIS

```
int putc (c, fp)
    int   c;    /* character */
    FILE  *fp;  /* stream */
```

DESCRIPTION

This routine appends a character to a specified output stream. It replicates the macro by the same name and is useful for invocations from the shell.

RETURNS

The character written, or EOF if there is an error.

SEE ALSO

stdioLib

strLib

NAME

strLib - string subroutine library

SYNOPSIS

***strcat*( )** - append second string to first

***strchr*( )** - return pointer to first occcurrence of char in string

***strcmp*( )** - compare two strings lexicographically

***strcpy*( )** - copy second string to first

***strcspn*( )** - return number of initial chars of second not found in first

***strerror*( )** - return pointer to error message string for argument

***strlen*( )** - return length of string

***strncat*( )** - append up to n chars of second string to first

***strncmp*( )** - compare upto n chars of two strings lexicographically

***strncpy*( )** - copy upto n chars of second string to first

***strpbrk*( )** - returnt pointer to first char of second string found in first

***strpos*( )** - return index of first occurrence of char in string

***strrchr*( )** - return pointer to last occurrence of char in string

***strrpbrk*( )** - return pointer to last char of second char found in first

***strrpos*( )** - return index of last occurrence of char in string

***strspn*( )** - return number of initial chars in first string found in second

***strstr*( )** - return pointer to first occurrence of second string in first

***strtok*( )** - return pointer substring of first delimited by chars in second

***strcoll*( )** - like strcmp but uses locale-specific collating rules

***strxfrm*( )** - transforms upto n chars so strcmp(t1, t2) == strcoll(s1, s2)

***index*( )** - same as ANSI strchr (non-ANSI)

***rindex*( )** - same as ANSI strrchr (non-ANSI)

```
char * strcat (char *s1, const char *s2)
char * strchr (const char *s, int n)
int    strcmp (const char *s1, const char *s2)
char * strcpy (char *s1, const char *s2)
size_t strcspn (const char *s1, const char *s2)
char * strerror (int e)
size_t strlen (const char *s)
char * strncat (char *s1, const char *s2, size_t n)
int    strncmp (const char *s1, const char *s2, size_t n)
char * strncpy (char *s1, const char *s2, size_t)
char * strpbrk (const char *s1, const char *s2)
int    strpos (const char *s, char c)
char * strrchr (const char *s, int n)
char * strrpbrk (const char *s1, const char *s2)
```

```
int     strrpos (const char *s, char c)
size_t strspn (const char *s1, const char *s2)
char * strstr (const char *s1, const char *s2)
char * strtok (char *s1, const char *s2)
int     strcoll (const char *s1, const char *s2)
size_t strxfrm (char *s1, const char *s2, size_t n)
char * index (const char *s, int c)
char * rindex (const char *s, int c)
```

DESCRIPTION

This library contains routines that duplicate the ANSI-compliant versions of the UNIX string processing package. The functions operate on null-terminated strings of characters. They do not check for the overflow of any resulting strings.

NOTE

These routines actually resolve to those supplied with the compiler tool set, for example the Intel GNU/960 tool set. The documentation for these routines is not included as part of Vx960, but they are documented in the GNU/960 documentation set in the book "C, A Reference Manual," by Samuel P. Harbison and Guy L. Steele Jr. (The routines *strcoll*( ) and *strxfrm*( ) only appear in the Third Edition and later.)

INCLUDE FILE

#include "strLib.h"

SEE ALSO

bLib, "C, A Reference Manual," Harbison and Steele, Prentice Hall

swapLib

NAME

swapLib - byte swapping functions for little endian machines

SYNOPSIS

htonl( ) - convert long (32bit) from host to network byte order.
ntohl( ) - convert long (32bit) from network to host byte order.
htons( ) - convert long (16bit) from host to network byte order.
ntohs( ) - convert long (16bit) from network to host byte order.

```
UINT32 htonl (x)
UINT32 ntohl (x)
UINT16 htons (x)
UINT16 ntohs (x)
```

DESCRIPTION

This library contains functions to convert long (32-bit) and short (16-bit) quantities from processor (host) byte order to network byte order, and visa-versa. These functions duplicate the macros of the same name for those who do not include the header file **in.h**.

INCLUDE FILE

in.h

SEE ALSO

Network.

htonl( )

NAME

htonl( ) - convert long (32 bit) from host to network byte order.

SYNOPSIS

```
UINT32 htonl (x)
    UINT32      x;
```

DESCRIPTION

Host to net byte swap function. On little endian processors such as the 960 this converts from little endian to big endian. Duplicates macro of the same name in the header file **in.h**.

RETURNS

Input value in network byte order

SEE ALSO

swapLib, *htons*( )

ntohl( )

NAME

ntohl( ) - convert long (32 bit) from network to host byte order.

SYNOPSIS

```
UINT32 ntohl (x)
    UINT32      x;
```

DESCRIPTION

Same code as *htonl*( ), but duplicated to save funtion call overhead. Duplicates macro of the same name in the header file **in.h**.

RETURNS

Input value in host byte order

SEE ALSO

swapLib, *htonl*( )

htons( )

NAME

htons( ) - convert short (16 bit) from host to network byte order.

SYNOPSIS

```
UINT16 htons (x)
    UINT16 x;
```

DESCRIPTION

Converts short (16bit) ints from processor (host) byte order to network byte order. On the i960 microprocessor, this converts from little endian to big endian byte orders. Duplicates macro of the same name in the header file **in.h**.

RETURNS

Input value in network byte order.

SEE ALSO

swapLib, *htonl*( )

ntohs( )

NAME

ntohs( ) - convert short (16 bit) from network to host byte order.

SYNOPSIS

```
UINT16 ntohs (x)
    UINT16 x;
```

DESCRIPTION

Same as htons, but duplicated to save function call. Duplicates macro of the same name in the header file **in.h**.

SEE ALSO

swapLib, *htons*( )

RETURNS

Input value in host byte order.

symLib

NAME

symLib - symbol table subroutine library

SYNOPSIS

symLibInit( ) - initialize the symbol table library
symTblCreate( ) - create a symbol table
symTblDelete( ) - delete a symbol table
symAdd( ) - create and add a symbol to a symbol table
symRemove( ) - remove a symbol from a symbol table
symFindByName( ) - look up a symbol by name
symFindByNameAndType( ) - look up a symbol by name and type
symFindByValue( ) - look up a symbol by value
symFindByValueAndType( ) - look up a symbol by value and type
symEach( ) - call a routine to examine each entry in a symbol table

```
STATUS symLibInit ()
SYMTAB_ID symTblCreate (hashSizeLog2, sameNameOk, symPartId)
STATUS symTblDelete (symTblId)
STATUS symAdd (symTblId, name, value, type)
STATUS symRemove (symTblId, name, type)
STATUS symFindByName (symTblId, name, pValue, pType)
STATUS symFindByNameAndType (symTblId, name, pValue, pType, sType, mask)
STATUS symFindByValue (symTblId, value, name, pValue, pType)
STATUS symFindByValueAndType (symTblId, value, name, pValue, pType, sType, mask)
SYMBOL *symEach (symTblId, routine, routineArg)
```

DESCRIPTION

This library provides facilities for managing symbol tables. A symbol table associates a name and type with a value. A name is simply an arbitrary, null-terminated string. A symbol type is a small integer (typdef 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.

Tables are created with *symTblCreate*( ), which returns a symbol table ID. This ID serves as a handle for symbol table operations, including the adding to, removing from, and searching of tables. All operations on a symbol table are interlocked by means of a mutual-exclusion semaphore in the symbol table structure. Tables are deleted with *symTblDelete*( ).

Symbols are added to a symbol table with *symAdd*( ). Each symbol in the symbol table has a name, a value, and a type. Symbois 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 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 FILE

symLib.h

SEE ALSO

loadLib

symLibInit( )

NAME

symLibInit( ) - initialize the symbol table library

SYNOPSIS

```
STATUS symLibInit ()
```

DESCRIPTION

This routine initializes the symbol table package. If INCLUDE_SYM_TBL is defined in **configAll.h**, it is called by *usrRoot( )* in **usrConfig.c**.

RETURNS

OK, or ERROR if the library could not be initialized.

SEE ALSO

symLib

symTblCreate( )

NAME

symTblCreate( ) - create a symbol table

SYNOPSIS

```
SYMTAB_ID symTblCreate (hashSizeLog2, sameNameOk, symPartId)
    int      hashSizeLog2;  /* size of hash table as a power of two */
    BOOL     sameNameOk;    /* allow two symbols of same name and type */
    PART_ID  symPartId;     /* memory partition 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 (symTblId)
    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

symAdd( )

NAME

symAdd( ) - create and add a symbol to a symbol table

SYNOPSIS

```
STATUS symAdd (symTblId, name, value, type)
    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 */
```

DESCRIPTION

This routine allocates a symbol *name* and adds it to a specified symbol table *symTblId* with the specified parameters *value* and *type*.

RETURNS

OK, or ERROR if the symbol table is invalid or there is insufficient memory for the symbol to be allocated.

SEE ALSO

symLib

symRemove( )

NAME

symRemove( ) - remove a symbol from a symbol table

SYNOPSIS

```
STATUS symRemove (symTblId, name, type)
    SYMTAB_ID  symTblId;  /* symbol table 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.

RETURNS

OK, or ERROR if the symbol is not found or could not be deallocated.

SEE ALSO

symLib

symFindByName( )

NAME

symFindByName( ) - look up a symbol by name

SYNOPSIS

```
STATUS symFindByName (symTblId, name, pValue, pType)
    SYMTAB_ID  symTblId;  /* ID of symbol table to find name in */
    char       *name;     /* symbol name to look for */
    char       **pValue;  /* pointer where to return symbol value */
    SYM_TYPE   *pType;    /* pointer where to return 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.

RETURNS

OK, or ERROR if the symbol table ID is invalid or the symbol is not found.

SEE ALSO

symLib

symFindByNameAndType( )

NAME

symFindByNameAndType( ) - look up a symbol by name and type

SYNOPSIS

```
STATUS symFindByNameAndType (symTblId, name, pValue, pType, sType, mask)
    SYMTAB_ID  symTblId;  /* ID of symbol table to find name in */
    char       *name;     /* symbol name to look for */
    char       **pValue;  /* pointer where to return symbol value */
    SYM_TYPE   *pType;    /* pointer where to return symbol type */
    SYM_TYPE   sType;     /* symbol type to look for */
    SYM_TYPE   mask;      /* which 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.

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 (symTblId, value, name, pValue, pType)
    SYMTAB_ID  symTblId;  /* ID of symbol table to find name in */
    int        value;     /* value of symbol to find */
    char        *name;      /* pointer where to return symbol name string */
    int         *pValue;    /* pointer where to return symbol value */
    SYM_TYPE    *pType;     /* pointer where to return 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*.

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 (symTblId, value, name, pValue, pType, sType, mask)
    SYMTAB_ID   symTblId;   /* ID of symbol table to find name in */
    int         value;      /* value of symbol to find */
    char *      name;       /* pointer where to return symbol name string */
    int *       pValue;     /* pointer where to return symbol value */
    SYM_TYPE *  pType;      /* pointer where to return symbol type */
    SYM_TYPE    sType;      /* symbol type to look for */
    SYM_TYPE    mask;       /* which 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.

RETURNS

OK, or ERROR if *value* is less than the lowest value in the table.

SEE ALSO

symLib

symEach( )

NAME

symEach( ) - call a routine to examine each entry in a symbol table

SYNOPSIS

```
SYMBOL *symEach (symTblId, routine, routineArg)
    SYMTAB_ID  symTblId;     /* pointer to symbol table              */
    FUNCPTR    routine;      /* the routine to call for each table entry */
    int        routineArg;   /* arbitrary user-supplied argument     */
```

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 (name, val, type, arg)
    char     *name; /* entry name                     *
    int      val; /* value associated with the entry  *
    SYM_TYPE  type; /* entry type                     *
    int      arg; /* arbitrary user-supplied argument *
```

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

taskArchLib

NAME

taskArchLib - architecture specific task management routines for kernel

SYNOPSIS

taskRegsInit( ) - initialize a task's registers
taskArgsSet( ) - set a task's arguments
taskArgsGet( ) - get a task's arguments
taskRegsShow( ) - display contents of a task's registers
taskPCWSet( ) - set task proccessor control word
taskACWSet( ) - set task arithmetic control word
taskTCWSet( ) - set task trace control word
taskRtnValueSet( )taskRtnValueSet -
taskStackAllot( ) - allot memory from caller's stack
taskRegsStackToTcb( ) - move the i960 local registers from stack to tcb.
taskRegsTcbToStack( ) - move the i960 local registers from tcb to stack.

```
VOID taskRegsInit (pTcb, pStackBase)
VOID taskArgsSet (pTcb, pStackBase, pArgs)
VOID taskArgsGet (pTcb, pStackBase, pArgs)
VOID taskRegsShow (tid)
STATUS taskPCWSet (tid, pcw)
STATUS taskACWSet (tid, acw)
STATUS taskTCWSet (tid, tcw)
VOID taskRtnValueSet (pTcb, value)
void *taskStackAllot (tid, nBytes)
VOID taskRegsStackToTcb (pTcb)
VOID taskRegsTcbToStack (pTcb)
```

DESCRIPTION

This library provides an interface to i960 specific task management routines.

taskRegsInit( )

NAME

taskRegsInit( ) - initialize a task's registers

SYNOPSIS

```
VOID taskRegsInit (pTcb, pStackBase)
    WIND_TCB *pTcb;       /* pointer TCB to initialize */
    char *pStackBase;           /* bottom of task's stack */
```

DESCRIPTION

This routine initializes the task's local and global registers to zero, the rip, fp, and sp to stack addresses (and initializes their locations within the stack frame as well), and initializes the tcw, acw, and pcw for normal operation.

SEE ALSO

taskArchLib

taskArgsSet( )

NAME

taskArgsSet( ) - set a task's arguments

SYNOPSIS

```
VOID taskArgsSet (pTcb, pStackBase, pArgs)
    WIND_TCB *  pTcb;       /* pointer TCB to initialize */
    char *      pStackBase; /* bottom of task's stack */
    int         pArgs[];    /* array of startup arguments */
```

DESCRIPTION

This routine puts the task's arguments into global registers as well as saving them in stack variables in preparation for calling the task entry point.

SEE ALSO

taskArchLib

taskArgsGet( )

NAME

taskArgsGet( ) - get a task's arguments

SYNOPSIS

```
VOID taskArgsGet (pTcb, pStackBase, pArgs)
    WIND_TCB *  pTcb;       /* pointer TCB to initialize */
    char *      pStackBase; /* bottom of task's stack */
    int         pArgs[];    /* array of arguments to fill */
```

DESCRIPTION

Fills the pArgs array from the stack available as saved by *taskArgsSet*( ).

SEE ALSO

taskArchLib, *taskArgsSet*( )

taskRegsShow( )

NAME

taskRegsShow( ) - display contents of a task's registers

SYNOPSIS

```
VOID taskRegsShow (tid)
    int tid;            /* task ID */
```

DESCRIPTION

This routine prints to standard out the contents of a task's registers.

EXAMPLE

```
    -> taskRegsShow (taskNameToId ("fNetTask"))
```

CAVEAT

Since taskRegsShow gets its information from the TCB, this will only be accurate up to the point that the TCB was last saved at task-switch time. Therefore, it may not be meaningful for a task to attempt to examine its own registers.

SEE ALSO

taskArchLib

taskPCWSet( )

NAME

taskPCWSet( ) - set task proccessor control word

SYNOPSIS

```
STATUS taskPCWSet (tid, pcw)
    int         tid;            /* task id */
    UINT32      pcw;            /* new PCW */
```

DESCRIPTION

This routine sets the processor control word (PCW) of a task not running (i.e. the task ID must NOT be that of the calling task). It is used by the debugging facilities to set the trace bit in the PCW of a task being single-stepped.

RETURNS

OK or ERROR if invalid task id

SEE ALSO

taskArchLib

taskACWSet( )

NAME

taskACWSet( ) - set task arithmetic control word

SYNOPSIS

```
STATUS taskACWSet (tid, acw)
    int         tid;            /* task id */
    UINT32      acw;            /* new ACW */
```

DESCRIPTION

This routine sets the arithmetic control word (ACW) of a task not running (i.e. the task ID must NOT be that of the calling task). It is used by the debugging facilities to set the "no imprecise faults" bit in the ACW of a task being single-stepped.

RETURNS

OK or ERROR if invalid task id

SEE ALSO

taskArchLib

taskTCWSet( )

NAME

taskTCWSet( ) - set task trace control word

SYNOPSIS

```
STATUS taskTCWSet (tid, tcw)
    int         tid;            /* task id */
    UINT32      tcw;            /* new TCW */
```

DESCRIPTION

This routine sets the trace control word (TCW) of a task not running (i.e. the task ID must NOT be that of the calling task). It is used by the debugging facilities to set mask trace bits in the TCW of a task being debugged.

RETURNS

OK or ERROR if invalid task id

SEE ALSO

taskArchLib

taskRtnValueSet( )

NAME

taskRtnValueSet( )taskRtnValueSet -

SYNOPSIS

```
VOID taskRtnValueSet (pTcb, value)
    WIND_TCB *  pTcb;           /* pointer to TCB */
    INT32       value;          /* value to set */
```

DESCRIPTION

Set the return value of function in the given TCB.

SEE ALSO

taskArchLib

taskStackAllot( )

NAME

taskStackAllot( ) - allot memory from caller's stack

SYNOPSIS

```
void *taskStackAllot (tid, nBytes)
    int      tid;    /* task whose stack will be allotted from */
    unsigned nBytes; /* number of bytes to allot */
```

DESCRIPTION

This routine allots the specified amount of memory from the end of the caller's stack. This is a non-blocking operation. This routine is utilized by task create hooks to allocate any additional memory they need. The memory cannot be added back to the stack. It will be reclaimed as part of the reclamation of the task stack when the task is deleted.

Note that a stack crash will overwrite the allotments made from this routine because all portions are carved from the end of the stack. Use checkStack to diagnose possible task stack overflows.

This routine will return NULL if requested size exceeds available stack memory.

RETURNS

pointer to block, or
NULL if unsuccessful.

SEE ALSO

taskArchLib, checkStack

taskRegsStackToTcb( )

NAME

taskRegsStackToTcb( ) - move the i960 local registers from stack to TCB.

SYNOPSIS

```
VOID taskRegsStackToTcb (pTcb)
    WIND_TCB *pTcb;              /* pointer to task TCB */
```

DESCRIPTION

Used by taskRegsShow task switching and the debug facilities to put local registers, which may have been cached on chip, into the task's TCB.

SEE ALSO

taskArchLib

taskRegsTcbToStack( )

NAME

taskRegsTcbToStack( ) - move the i960 local registers from TCB to stack.

SYNOPSIS

```
VOID taskRegsTcbToStack (pTcb)
    WIND_TCB *pTcb;                /* pointer to task TCB */
```

DESCRIPTION

Used by the task switching and debug facilities to move the local registers from the task's TCB to the stack from which they will be loaded when the task resumes.

SEE ALSO

taskArchLib

taskHookLib

NAME

taskHookLib - task hook library

SYNOPSIS

taskCreateHookAdd( ) - add a routine to be called at every task create
taskCreateHookDelete( ) - delete a previously added task create routine
taskCreateHookShow( ) - show the list of task create routines
taskSwitchHookAdd( ) - add a routine to be called at every task switch
taskSwitchHookDelete( ) - delete a previously added task switch routine
taskSwitchHookShow( ) - show switch routines
taskDeleteHookAdd( ) - add a routine to be called at every task delete
taskDeleteHookDelete( ) - delete a previously added task delete routine
taskDeleteHookShow( ) - show the delete routines

```
STATUS taskCreateHookAdd (createHook)
STATUS taskCreateHookDelete (createHook)
VOID taskCreateHookShow ()
STATUS taskSwitchHookAdd (switchHook)
STATUS taskSwitchHookDelete (switchHook)
VOID taskSwitchHookShow ()
STATUS taskDeleteHookAdd (deleteHook)
STATUS taskDeleteHookDelete (deleteHook)
VOID taskDeleteHookShow ()
```

DESCRIPTION

This library provides routines for adding extensions to the Vx960 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, which 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. Vx960 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.

Vx960 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.

SEE ALSO

taskLib, *Programmer's Guide: Basic OS*

taskCreateHookAdd( )

NAME

taskCreateHookAdd( ) - add a routine to be called at every task create

SYNOPSIS

```
STATUS taskCreateHookAdd (createHook)
    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 (pNewTcb)
    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 (createHook)
    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 ()
```

DESCRIPTION

This routine shows all the task create routines installed in the task create hook table in the order they were installed.

RETURNS

N/A

SEE ALSO

taskHookLib, *taskCreateHookAdd*( )

taskSwitchHookAdd( )

NAME

taskSwitchHookAdd( ) - add a routine to be called at every task switch

SYNOPSIS

```
STATUS taskSwitchHookAdd (switchHook)
    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 (pOldTcb, pNewTcb)
        WIND_TCB *pOldTcb; /* pointer to old task's WIND_TCB *
        WIND_TCB *pNewTcb; /* pointer to new task's WIND_TCB *
```

RETURNS

OK, or ERROR if the table of task switch routines is full.

SEE ALSO

taskHookLib, *taskSwitchHookDelete*( )

taskSwitchHookDelete( )

NAME

taskSwitchHookDelete( ) - delete a previously added task switch routine

SYNOPSIS

```
STATUS taskSwitchHookDelete (switchHook)
    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 switch routines

SYNOPSIS

```
VOID taskSwitchHookShow ()
```

DESCRIPTION

This routine shows all the switch routines installed in the task switch hook table in the order they were installed.

RETURNS

N/A

SEE ALSO

taskHookLib, *taskSwitchHookAdd*( )

taskDeleteHookAdd( )

NAME

taskDeleteHookAdd( ) - add a routine to be called at every task delete

SYNOPSIS

```
STATUS taskDeleteHookAdd (deleteHook)
    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 (pTcb)
        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 (deleteHook)
    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 delete routines

SYNOPSIS

```
VOID taskDeleteHookShow ()
```

DESCRIPTION

This routine shows all the delete routines installed in the task delete hook table in the order they were installed. Note that the delete routines will be run in reverse of the order they were installed.

RETURNS

N/A

SEE ALSO

taskHookLib, *taskDeleteHookAdd*( )

taskLib

NAME

taskLib - task management library for the Vx960 kernel

SYNOPSIS

taskSpawn( ) - spawn a task
taskInit( ) - initialize a task with a stack at a specified address
taskActivate( ) - activate a task that has been initialized
exit( ) - exit a task
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
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
taskIdVerify( ) - verify the existence of a task
taskIdSelf( ) - get the task ID of a running task
taskIdDefault( ) - set the default task ID
taskIsReady( ) - check if a task is ready to run
taskIsSuspended( ) - check if a task is suspended
taskTcb( ) - get the task control block for a task ID
taskIdListGet( ) - get a list of active task IDs
taskInfoGet( ) - get information about a task
taskStatusString( ) - get a task's status as a string

```
int taskSpawn (name, priority, options, stackSize, entryPt, ...)
STATUS taskInit (pTcb, name, priority, options, pStackBase, stackSize, entryPt, .
STATUS taskActivate (tid)
VOID exit (code)
STATUS taskDelete (tid)
STATUS taskDeleteForce (tid)
STATUS taskSuspend (tid)
STATUS taskResume (tid)
STATUS taskRestart (tid)
STATUS taskPrioritySet (tid, newPriority)
STATUS taskPriorityGet (tid, pPriority)
STATUS taskLock ()
STATUS taskUnlock ()
STATUS taskSafe ()
STATUS taskUnsafe ()
STATUS taskDelay (ticks)
STATUS taskOptionsSet (tid, mask, newOptions)
STATUS taskOptionsGet (tid, pOptions)
STATUS taskRegsGet (tid, pRegs)
STATUS taskRegsSet (tid, pRegs)
char *taskName (tid)
int taskNameToId (name)
STATUS taskIdVerify (tid)
int taskIdSelf ()
int taskIdDefault (tid)
BOOL taskIsReady (tid)
BOOL taskIsSuspended (tid)
WIND_TCB *taskTcb (tid)
int taskIdListGet (idList, maxTasks)
STATUS taskInfoGet (tid, pTaskDesc)
STATUS taskStatusString (tid, pString)
```

DESCRIPTION

This library provides the interface to the Vx960 task management facilities. Task control services are provided by the Vx960 kernel, which is comprised of kernelLib, taskLib, semLib, tickLib, and wdLib.

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 Vx960 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 Vx960, as long as sufficient memory is available to satisfy allocation requirements.

The routine *sp*( ) is provided in **usrLib** as a convenient abbreviation for spawning tasks. It calls *taskSpawn*( ) with default parameters.

TASK DELETION

If a task exits its "main" routine, specified during task creation, the kernel implicitly calls *exit*( ) to delete the task. Tasks can be explicitly deleted with the *taskDelete*( ) 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 Vx960 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. Vx960 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

Vx960 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 Vx960 is dynamic, and an existing task's priority can be changed using *taskPrioritySet*( ).

TASK INFORMATION

The following routines provide task information: *taskInfoGet*( ), *taskRegsGet*( ), *taskIdListGet*( ), *taskPriorityGet*( ), *taskStatusString*( ), *taskIsSuspended*( ), *taskIsReady*( ) and *taskTcb*( ). Task information is crucial as a debugging aid and user-interface convenience during the development cycle of an application. Its chief drawback 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.

INCLUDE FILE

taskLib.h

SEE ALSO

taskHookLib, taskVarLib, semLib, kernelLib,
Programmer's Guide: Basic OS

taskSpawn( )

NAME

taskSpawn( ) - spawn a task

SYNOPSIS

```
int taskSpawn (name, priority, options, stackSize, entryPt,
                         arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8, arg9, ar
    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;        /* task argument one */
    int       arg2;        /* task argument two */
    int       arg3;        /* task argument three */
    int       arg4;        /* task argument four */
    int       arg5;        /* task argument five */
    int       arg6;        /* task argument six */
    int       arg7;        /* task argument seven */
    int       arg8;        /* task argument eight */
    int       arg9;        /* task argument nine */
    int       arg10;       /* task argument ten */
```

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 Vx960 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.

Bits in the options argument may be set to run with the following modes:

VX_UNBREAKABLE	- do not allow breakpoint debugging
VX_FP_TASK	- execute with coprocessor support
VX_STDIO	- execute with standard I/O support

See the definitions in **taskLib.h**.

RETURNS

The task ID, or ERROR if memory is insufficient or the task cannot be created.

SEE ALSO

taskLib, *taskInit*( ), *taskActivate*( ), *vxTaskEntry*( ), *sp*( ),
Programmer's Guide: Basic OS

taskInit( )

NAME

taskInit( ) - initialize a task with a stack at a specified address

SYNOPSIS

```
STATUS taskInit (pTcb, name, priority, options, pStackBase, stackSize, entryPt,
                              arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8, arg9,
    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;  /* bottom of new task's stack */
int       stackSize;    /* size (bytes) of stack needed */
FUNCPTR   entryPt;      /* entry point of new task */
int       arg1;         /* task argument one */
int       arg2;         /* task argument two */
int       arg3;         /* task argument three */
int       arg4;         /* task argument four */
int       arg5;         /* task argument five */
int       arg6;         /* task argument six */
int       arg7;         /* task argument seven */
int       arg8;         /* task argument eight */
int       arg9;         /* task argument nine */
int       arg10;        /* task argument ten */
```

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.

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.

SEE ALSO

taskLib, *taskActivate*( ), *taskSpawn*( )

taskActivate( )

NAME

taskActivate( ) - activate a task that has been initialized

SYNOPSIS

```
STATUS taskActivate (tid)
    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 *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( )*

exit( )

NAME

exit( ) - exit a task

SYNOPSIS

```
VOID exit (code)
    int  code;  /* code stored in TCB for delete hooks */
```

DESCRIPTION

A task may call this routine 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.

SEE ALSO

taskLib, *taskDelete( )*, *Programmer's Guide: Basic OS*

taskDelete( )

NAME

taskDelete( ) - delete a task

SYNOPSIS

```
STATUS taskDelete (tid)
    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.

SEE ALSO

taskLib, excLib, *taskDeleteHookAdd*( ), *taskDelete*( ),
Programmer's Guide: Basic OS

taskDeleteForce( )

NAME

taskDeleteForce( ) - delete a task without restriction

SYNOPSIS

```
STATUS taskDeleteForce (tid)
    int  tid;  /* task ID of task to delete */
```

DESCRIPTION

This routine is equivalent to *taskDelete*( ), except that it deletes a task even if the task is protected from deletion. Upon deletion, all routines specified by *taskDeleteHookAdd*( ) will be called in the context of the deleting task.

CAVEAT

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.

RETURNS

OK, or ERROR if the task cannot be deleted.

SEE ALSO

taskLib, *taskDeleteHookAdd*( ), *taskDelete*( )

taskSuspend( )

NAME

taskSuspend( ) - suspend a task

SYNOPSIS

```
STATUS taskSuspend (tid)
    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 may be delayed and suspended, or pended and suspended. Suspended, delayed tasks whose delays expire will remain suspended. Likewise, suspended, pended tasks that unblock will only remain suspended.

RETURNS

OK, or ERROR if the task cannot be suspended.

SEE ALSO

taskLib

taskResume( )

NAME

taskResume( ) - resume a task

SYNOPSIS

```
STATUS taskResume (tid)
    int  tid;  /* task ID of task to resume */
```

DESCRIPTION

This routine resumes a specified task. Suspension will be cleared, and the task will operate in the remaining state.

RETURNS

OK, or ERROR if the task cannot be resumed.

SEE ALSO

taskLib

taskRestart( )

NAME

taskRestart( ) - restart a task

SYNOPSIS

```
STATUS taskRestart (tid)
    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.

SEE ALSO

taskLib

taskPrioritySet( )

NAME

taskPrioritySet( ) - change the priority of a task

SYNOPSIS

```
STATUS taskPrioritySet (tid, newPriority)
    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.

SEE ALSO

taskLib, *taskPriorityGet*( )

taskPriorityGet( )

NAME

taskPriorityGet( ) - examine the priority of a task

SYNOPSIS

```
STATUS taskPriorityGet (tid, pPriority)
    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.

SEE ALSO

taskLib, *taskPrioritySet*( )

taskLock( )

NAME

taskLock( ) - disable task rescheduling

SYNOPSIS

```
STATUS taskLock ()
```

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.

RETURNS

OK or ERROR.

SEE ALSO

taskLib, *taskUnlock*( ), *intLock*( ), *taskSafe*( ), *semTake*( )

taskUnlock( )

NAME

taskUnlock( ) - enable task rescheduling

SYNOPSIS

```
STATUS taskUnlock ()
```

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.

RETURNS

OK or ERROR.

SEE ALSO

taskLib, *taskLock*( )

taskSafe( )

NAME

taskSafe( ) - make the calling task safe from deletion

SYNOPSIS

```
STATUS taskSafe ()
```

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 nesting occurs, the task becomes unsafe only after the outermost ***taskUnsafe***( ) is executed.

RETURNS

OK

SEE ALSO

taskLib, *Programmer's Guide: Basic OS*

taskUnsafe( )

NAME

taskUnsafe( ) - make the calling task unsafe from deletion

SYNOPSIS

```
STATUS taskUnsafe ()
```

DESCRIPTION

This routine removes the calling task's protection from deletion. Tasks that attempt to delete a protected task will block until the task is unsafe. When a task becomes unsafe, the deleter will be unblocked and allowed to delete the task.

The ***taskUnsafe***( ) primitive utilizes a count to keep track of nested calls for task protection. When nesting occurs, the task becomes unsafe only after the outermost ***taskUnsafe***( ) is executed.

RETURNS

OK

SEE ALSO

taskLib, *Programmer's Guide: Basic OS*

taskDelay( )

NAME

taskDelay( ) - delay a task from executing

SYNOPSIS

```
STATUS taskDelay (ticks)
    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.

RETURNS

OK, or ERROR if called from interrupt level.

SEE ALSO

taskLib

taskOptionsSet( )

NAME

taskOptionsSet( ) - change task options

SYNOPSIS

```
STATUS taskOptionsSet (tid, mask, newOptions)
    int  tid;         /* task ID */
    int  mask;        /* mask, 1 = ok to change this bit */
    int  newOptions;  /* new options */
```

DESCRIPTION

This routine changes the execution options of a task. The only options that are alterable after task creation are the following:

VX_UNBREAKABLE - do not allow debugging

See the definitions in **taskLib.h**.

RETURNS

OK, or ERROR if the task ID is invalid.

SEE ALSO

taskLib, *taskOptionsGet*( )

taskOptionsGet( )

NAME

taskOptionsGet( ) - examine task options

SYNOPSIS

```
STATUS taskOptionsGet (tid, pOptions)
    int  tid;        /* task ID */
    int  *pOptions;  /* task's options */
```

DESCRIPTION

This routine gets the current execution options of the specified task. Bits in the options argument may be set to indicate the following modes:

VX_UNBREAKABLE - do not allow debugging

VX_FP_TASK - execute with coprocessor support

VX_STDIO - execute with standard I/O support

See the definitions in **taskLib.h**.

RETURNS

OK, or ERROR if the task ID is invalid.

SEE ALSO

taskLib, *taskOptionsSet*( )

taskRegsGet( )

NAME

taskRegsGet( ) - get a task's registers from the TCB

SYNOPSIS

```
STATUS taskRegsGet (tid, pRegs)
    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, not-running state. Self-examination, for instance, is ill advised, as results are unpredictable.

RETURNS

OK, or ERROR if the task ID is invalid.

SEE ALSO

taskLib, *taskSuspend*( ), *taskRegsSet*( )

taskRegsSet( )

NAME

taskRegsSet( ) - set a task's registers

SYNOPSIS

```
STATUS taskRegsSet (tid, pRegs)
    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

taskLib, *taskSuspend*( ), *taskRegsGet*( )

taskName( )

NAME

taskName( ) - get the name associated with a task ID

SYNOPSIS

```
char *taskName (tid)
    int  tid;  /* ID of task whose name is to be found */
```

DESCRIPTION

This routine returns a pointer to the name of a task of specified ID, if it has a name; otherwise it returns NULL.

RETURNS

A pointer to the task name, or NULL if the task ID is invalid.

SEE ALSO

taskLib

taskNameToId( )

NAME

taskNameToId( ) - look up the task ID associated with a task name

SYNOPSIS

```
int taskNameToId (name)
    char  *name;  /* task name to look up */
```

DESCRIPTION

This routine returns the ID of the task matching a specified name. Referencing a task in this way is inefficient, since it involves a search of the task list.

RETURNS

The task ID, or ERROR if the task is not found.

SEE ALSO

taskLib

taskIdVerify( )

NAME

taskIdVerify( ) - verify the existence of a task

SYNOPSIS

```
STATUS taskIdVerify (tid)
    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.

SEE ALSO

taskLib, *objVerify*( )

taskIdSelf( )

NAME

taskIdSelf( ) - get the task ID of a running task

SYNOPSIS

```
int taskIdSelf ()
```

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

taskIdDefault( )

NAME

taskIdDefault( ) - set the default task ID

SYNOPSIS

```
int taskIdDefault (tid)
    int  tid;  /* user supplied task ID; if 0, return default */
```

DESCRIPTION

This routine maintains a global default task ID. This ID is used by libraries that want to allow a task ID argument to take on a default value if the user did not explicitly supply one.

If *tid* is not zero (i.e., the user did specify a task ID), the default ID is set to that value, and that value is returned. If *tid* is zero (i.e., the user did not specify a task ID), the default ID is not changed and its value is returned. Thus the value returned is always the last task ID the user specified.

RETURNS

Most recent non-zero task ID.

SEE ALSO

taskLib, **dbgLib**, *Programmer's Guide: Debugging*

taskIsReady( )

NAME

taskIsReady( ) - check if a task is ready to run

SYNOPSIS

```
BOOL taskIsReady (tid)
    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

taskLib

taskIsSuspended( )

NAME

taskIsSuspended( ) - check if a task is suspended

SYNOPSIS

```
BOOL taskIsSuspended (tid)
    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

taskLib

taskTcb( )

NAME

taskTcb( ) - get the task control block for a task ID

SYNOPSIS

```
WIND_TCB *taskTcb (tid)
    int  tid;  /* task ID */
```

DESCRIPTION

This routine returns a pointer to the task control block (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.

SEE ALSO

taskLib

taskIdListGet( )

NAME

taskIdListGet( ) - get a list of active task IDs

SYNOPSIS

```
int taskIdListGet (idList, maxTasks)
    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 no new tasks have been created by the time this routine returns.

RETURNS

The number of tasks put into the ID list.

SEE ALSO

taskLib

taskInfoGet( )

NAME

taskInfoGet( ) - get information about a task

SYNOPSIS

```
STATUS taskInfoGet (tid, pTaskDesc)
    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 WIND_TCB. The TASK_DESC is useful for common information and avoids dealing directly with the unwieldy WIND_TCB.

NOTE

Examination of WIND_TCBs should be restricted to debugging aids.

RETURNS

OK, or ERROR if the task ID is invalid.

SEE ALSO

taskLib

taskStatusString( )

NAME

taskStatusString( ) - get a task's status as a string

SYNOPSIS

```
STATUS taskStatusString (tid, pString)
    int   tid;       /* task to get string for */
    char  *pString;  /* where to return string */
```

DESCRIPTION

For a specified task, this routine deciphers the WIND task status word in the TCB and copies the appropriate string to *pString*.

EXAMPLE

```
    -> taskStatusString (taskNameToId ("shell"), xx=malloc (10));
    new symbol "xx" added to symbol table.
    value = 0 = 0x0
    -> printf ("shell status = <%s>\n", xx)
    shell status = <READY>
    value = 2 = 0x2
```

RETURNS

OK, or ERROR if the task ID is invalid.

SEE ALSO

taskLib

taskVarLib

NAME

taskVarLib - task variables support library

SYNOPSIS

taskVarInit( ) - initialize the task variables facility
taskVarAdd( ) - add a task variable to a task
taskVarDelete( ) - remove a task variable from a task
taskVarGet( ) - get the value of a task variable
taskVarSet( ) - set the value of a task variable
taskVarInfo( ) - get a list of task variables of a task

```
STATUS taskVarInit ()
STATUS taskVarAdd (tid, pVar)
STATUS taskVarDelete (tid, pVar)
int taskVarGet (tid, pVar)
STATUS taskVarSet (tid, pVar, value)
int taskVarInfo (tid, varList, maxVars)
```

DESCRIPTION

Vx960 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.

SEE ALSO

taskHookLib, *Programmer's Guide: Basic OS*

taskVarInit( )

NAME

taskVarInit( ) - initialize the task variables facility

SYNOPSIS

```
STATUS taskVarInit ()
```

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.

Vx960 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.

RETURNS

OK, or ERROR if the task switch/delete hooks could not be installed.

SEE ALSO

taskVarLib

taskVarAdd( )

NAME

taskVarAdd( ) - add a task variable to a task

SYNOPSIS

```
STATUS taskVarAdd (tid, pVar)
    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. An efficient use of 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 ( )*. They 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; /* pointer to operator task's global variables *

operator (opNum)
    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 (tid, pVar)
    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 (tid, pVar)
    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*( )

taskVarSet( )

NAME

taskVarSet( ) - set the value of a task variable

SYNOPSIS

```
STATUS taskVarSet (tid, pVar, value)
    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*( )

taskVarInfo( )

NAME

taskVarInfo( ) - get a list of task variables of a task

SYNOPSIS

```
int taskVarInfo (tid, varList, maxVars)
    int       tid;        /* ID of task whose task variable is to be set */
    TASK_VAR  varList[];  /* array of task variable addresses to be filled in */
    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 no new task variables have been created by the time this routine returns.

RETURNS

The number of tasks variables in the list.

SEE ALSO

taskVarLib

telnetLib

NAME

telnetLib - telnet library

SYNOPSIS

telnetInit( ) - initialize the telnet daemon
telnetd( ) - Vx960 telnet daemon

```
VOID telnetInit ()
VOID telnetd ()
```

DESCRIPTION

This library provides a remote login facility for Vx960. It uses the ARPA telnet protocol to enable users on remote systems to log into Vx960.

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*( ); if INCLUDE_TELNET is defined in **configAll.h**, it is called from the root task, *usrRoot*( ), in **usrConfig.c**.

Internally, the telnet daemon provides a tty-like interface to the remote user through the use of the Vx960 pseudo-terminal driver, **ptyDrv**.

SEE ALSO

ptyDrv, rlogLib, UNIX documentation for telnet, telnetd, and pty

telnetInit( )

NAME

telnetInit( ) - initialize the telnet daemon

SYNOPSIS

```
VOID telnetInit ()
```

DESCRIPTION

This routine initializes the telnet facility, which supports remote login to the Vx960 shell via the ARPA telnet protocol. It creates a pty device and spawns the telnet daemon. If INCLUDE_TELNET is defined in **configAll.h**, it is called from the root task, *usrRoot*( ), in **usrConfig.c**, before any system tries to log into this Vx960 system via telnet.

RETURNS

N/A

SEE ALSO

telnetLib

telnetd( )

NAME

telnetd( ) - Vx960 telnet daemon

SYNOPSIS

```
VOID telnetd ()
```

DESCRIPTION

This routine enables remote users to log into Vx960 over the network via the ARPA telnet protocol. It is spawned by ***telnetInit***( ), which should be called by ***usrRoot***( ) 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, *telnetInTask* and *telnetOutTask,* whenever a remote user is logged in. These processes exit when the remote connection is terminated.

RETURNS

N/A

SEE ALSO

telnetLib

tickLib

NAME

tickLib - clock tick support library

SYNOPSIS

tickAnnounce( ) - announce a clock tick to the kernel
tickSet( ) - set the value of the kernel's tick counter
tickGet( ) - get the value of the kernel's tick counter

```
VOID tickAnnounce ()
VOID tickSet (ticks)
ULONG tickGet ()
```

DESCRIPTION

This library is the interface to the Vx960 kernel routines that announce a clock tick to the kernel, that get the current time in ticks, and that 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 from time to time.

SEE ALSO

kernelLib, taskLib, semLib, wdLib, *Programmer's Guide: Basic OS*

tickAnnounce( )

NAME

tickAnnounce( ) - announce a clock tick to the kernel

SYNOPSIS

```
VOID tickAnnounce ()
```

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, *usrClock*( ) in **usrConfig** calls this routine.

RETURNS

N/A

SEE ALSO

tickLib, **kernelLib**, **taskLib**, **semLib**, **wdLib**, *Programmer's Guide: Basic OS*

tickSet( )

NAME

tickSet( ) - set the value of the kernel's tick counter

SYNOPSIS

```
VOID tickSet (ticks)
    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*( )

tickGet( )

NAME

tickGet( ) - get the value of the kernel's tick counter

SYNOPSIS

```
ULONG tickGet ()
```

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 the number of *tickAnnounce*( ) calls since.

SEE ALSO

tickLib, *tickSet*( ), *tickAnnounce*( )

timexLib

NAME

timexLib - execution timer facilities

SYNOPSIS

timexInit( ) - include the execution timer library
timexClear( ) - clear the list of function calls to be timed
timexFunc( ) - specify functions to be timed
timexHelp( ) - display synopsis of execution timer facilities
timex( ) - time a single execution of a function or functions
timexN( ) - time repeated executions of a function or group of functions
timexPost( ) - specify functions to be called after timing
timexPre( ) - specify functions to be called prior to timing
timexShow( ) - display the list of function calls to be timed

```
VOID timexInit ()
VOID timexClear ()
VOID timexFunc (i, func, arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8)
VOID timexHelp ()
VOID timex (func, arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8)
VOID timexN (func, arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8)
VOID timexPost (i, func, arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8)
VOID timexPre (i, func, arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8)
VOID timexShow ()
```

DESCRIPTION

This library contains routines for timing the execution of programs, individual functions, and groups of functions. The Vx960 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 functions 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.

SEE ALSO

spyLib

timexInit( )

NAME

timexInit( ) - include the execution timer library

SYNOPSIS

```
VOID timexInit ()
```

DESCRIPTION

This null routine is provided so that **timexLib** can be linked in. If INCLUDE_TIMEX is defined in **configAll.h**, it is called by the root task, *usrRoot*( ), in **usrConfig.c**.

RETURNS

N/A

SEE ALSO

timexLib

timexClear( )

NAME

timexClear( ) - clear the list of function calls to be timed

SYNOPSIS

```
VOID timexClear ()
```

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 (i, func, arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8)
    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 params called with function */
    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

timexHelp( )

NAME

timexHelp( ) - display synopsis of execution timer facilities

SYNOPSIS

```
VOID timexHelp ()
```

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

timex( )

NAME

timex( ) - time a single execution of a function or functions

SYNOPSIS

```
VOID timex (func, arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8)
   FUNCPTR  func;  /* function to time (optional)                 */
   int      arg1;  /* args with which to call function (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*( )

timexN( )

NAME

timexN( ) - time repeated executions of a function or group of functions

SYNOPSIS

```
VOID timexN (func, arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8)
    FUNCPTR  func;  /* function to time (optional)                 */
    int      arg1;  /* first of up to 8 params to call function with */
    int      arg2;
    int      arg3;
    int      arg4;
    int      arg5;
    int      arg6;
    int      arg7;
    int      arg8;
```

DESCRIPTION

This routine times the execution of the current list of functions to be timed in the same manner as *timex*( ); however, the list of functions is called a variable number of times until sufficient resolution is achieved to establish the time with an error less than 2%. (Since each iteration of the list may be measured to a resolution of +/- 1 clock tick, repetitive timings decrease this error to 1/N ticks, where N is the number of repetitions.)

RETURNS

N/A

SEE ALSO

timexLib, *timexFunc*( ), *timex*( )

timexPost( )

NAME

timexPost( ) - specify functions to be called after timing

SYNOPSIS

```
VOID timexPost (i, func, arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8)
    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 params 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 (i, func, arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8)
    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 params 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 ()
```

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

trcLib

NAME

trcLib - stack trace library

SYNOPSIS

trcStack( ) - print trace of function calls from stack
trcArgCntSet( ) - change the number of arguments printed by *trcStack*( )

```
VOID trcStack (pRegSet, printRtn)
UINT32 trcArgCntSet (cnt)
```

DESCRIPTION

This module provides a routine, *trcStack*( ), which traces a stack given the current frame pointer, stack pointer, and instruction pointer. The resulting stack trace lists the nested routine calls and their arguments.

This module provides the low-level stack trace facility. A higher-level symbolic stack trace, implemented on top of this facility, is provided by the routine *tt*( ) in dbgLib.

SEE ALSO

"Debugging", dbgLib, *tt*( )

trcStack( )

NAME

trcStack( ) - print trace of function calls from stack

SYNOPSIS

```
VOID trcStack (pRegSet, printRtn)
    REG_SET     *pRegSet;   /* pointer to task's register set     */
    FUNCPTR     printRtn;   /* routine to print single function call */
```

DESCRIPTION

This routine provides the low-level stack trace function. (A higher-level symbolic stack trace built on top of this is provided by the routine *tt*( ) in dbgLib.) A list of the nested routine calls that are on the stack is printed. Only the last routine's parameters are shown since arguments are passed in registers, not on the stack with gcc960.

The stack being traced should be quiescent. Avoid tracing your own stack.

PRINT ROUTINE

In order to allow symbolic or other alternative printout formats, the call to this routine includes the "printRtn" parameter, which is a routine to be called at each nesting level to print out the routine name and its arguments. This routine should be declared as follows:

```
VOID printRtn (callAdrs, rtnAdrs, nargs, args)
  INSTR *callAdrs;  /* address from which routine was called *
  int rtnAdrs;      /* address of routine called *
  int nargs;        /* number of arguments in call *
  int *args;        /* pointer to arguments *
```

If a NULL printRtn is specified, a default routine will be used which just prints out the call address, function address, and arguments as hex values.

CAVEAT

In order to do the trace, some assumptions are made. In general, the trace will work for all C language routines, and asembly language routines. However, routines written in other languages, strange entries into routines, or tasks with corrupted stacks, can make the trace very confused. Also, all parameters are assumed to be 32-bit quantities, so structures passed as parameters will be displayed as some number of integers.

SEE ALSO

trcLib, tt(2)

trcArgCntSet( )

NAME

trcArgCntSet( ) - change the number of arguments printed by *trcStack*( )

SYNOPSIS

```
UINT32 trcArgCntSet (cnt)
    UINT32      cnt;      /* value to set count to        */
```

DESCRIPTION

This routine changes the default number of arguments printed out per function call. Note that this only affects the last routine found on the stack since arguments are not passed on the stack with gcc960.

RETURNS

Previous argument count.

SEE ALSO

trcLib

tyLib

NAME

tyLib - tty driver support library

SYNOPSIS

tyDevInit( ) - initialize the tty device descriptor
tyAbortFuncSet( ) - set the abort function
tyAbortSet( ) - change the abort character
tyBackspaceSet( ) - change the backspace character
tyDeleteLineSet( ) - change the line-delete character
tyEOFSet( ) - change the end-of-file character
tyMonitorTrapSet( ) - change the trap-to-monitor character
tyIoctl( ) - handle device control requests
tyWrite( ) - do a task-level write for a tty device
tyRead( ) - do a task-level read for a tty device
tyITx( ) - interrupt-level output
tyIRd( ) - interrupt level input

```
STATUS tyDevInit (pTyDev, rdBufSize, wrtBufSize, txStartup)
VOID tyAbortFuncSet (func)
VOID tyAbortSet (ch)
VOID tyBackspaceSet (ch)
VOID tyDeleteLineSet (ch)
VOID tyEOFSet (ch)
VOID tyMonitorTrapSet (ch)
STATUS tyIoctl (pTyDev, request, arg)
int tyWrite (pTyDev, buffer, nbytes)
int tyRead (pTyDev, buffer, maxbytes)
STATUS tyITx (pTyDev, pChar)
STATUS tyIRd (pTyDev, inchar)
```

DESCRIPTION

This library provides routines used to implement drivers for serial devices. It provides all 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 deliver an input character

tyITx( ) - interrupt-level routine to get 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 any of the **tyCoDrv** drivers in the target directories.

TTY OPTIONS

Tty devices have a full range of options that affect the behavior of the device. These options are selected by setting bits in the device option word using the FIOSETOPTIONS function in the *ioctl*( ) routine (see the section I/O CONTROL FUNCTIONS). 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 ^Q and ^S in what is commonly known as X-on/X-off protocol. Receipt of a ^S input character will suspend output to that channel. Subsequent receipt of a ^Q will resume the output. Also, when the Vx960 input buffer is almost full, a ^S will be output to signal the other side to suspend transmission. When the input buffer is almost empty, a ^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 ^X. When this character is received and this option is enabled, Vx960 will trap to the ROM resident monitor program. Note that this is quite drastic. All normal Vx960 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 Vx960 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 ^C. When this character is received and this option is enabled, the Vx960 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 "Shell" chapter of the *Programmer's Guide*.

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 fd and copies it to the buffer referenced to by *nameBuf*:

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

This function is common to all fds 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);
```

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);
```

FIOWFLUSH - Discards all the bytes currently in the output buffer:

```
status = ioctl (fd, FIOWFLUSH);
```

FIORFLUSH - Discards all the bytes currently in the input buffers:

```
status = ioctl (fd, FIORFLUSH);
```

FIOCANCEL - Cancels a read or write. A task blocked on a read may have previously started a watchdog routine to time out the read. The watchdog would use this call on the appropriate fd:

```
status = ioctl (fd, FIOCANCEL);
```

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);
```

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 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);
```

FIOBUFSET - Changes the size of the receive-side buffer to *size*:

```
status = ioctl (fd, FIOBUFSET, size);
```

Any other *ioctl*( ) request will return an error, and set the status to S_ioLib_UNKNOWN REQUEST.

INCLUDE FILE

tyLib.h

SEE ALSO

ioLib, iosLib, tyCoDrv, *Programmer's Guide: I/O System*

tyDevInit( )

NAME

tyDevInit( ) - initialize the tty device descriptor

SYNOPSIS

```
STATUS tyDevInit (pTyDev, rdBufSize, wrtBufSize, txStartup)
    TY_DEV_ID  pTyDev;     /* pointer to ty device descriptor     */
                           /* to be initialized                   */
    int        rdBufSize;  /* required read buffer size in bytes  */
    int        wrtBufSize; /* required write buffer size in bytes */
    FUNCPTR    txStartup;  /* device transmit startup 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

tyAbortFuncSet( )

NAME

tyAbortFuncSet( ) - set the abort function

SYNOPSIS

```
VOID tyAbortFuncSet (func)
    FUNCPTR  func;  /* function to call when abort char is 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 (ch)
    char  ch;  /* character to be made the abort character */
```

DESCRIPTION

This routine sets the abort character to *ch*. The default abort character is ^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 (ch)
    char  ch;  /* character to be made the backspace character */
```

DESCRIPTION

This routine sets the backspace character to *ch*. The default backspace character is ^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 (ch)
    char  ch;  /* character to be made the line-delete character */
```

DESCRIPTION

This routine sets the line-delete character to *ch*. The default line-delete character is ^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

tyEOFSet( )

NAME

tyEOFSet( ) - change the end-of-file character

SYNOPSIS

```
VOID tyEOFSet (ch)
    char  ch;  /* character to be made the eof character */
```

DESCRIPTION

This routine sets the EOF character to *ch*. The default EOF character is ^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

tyMonitorTrapSet( )

NAME

tyMonitorTrapSet( ) - change the trap-to-monitor character

SYNOPSIS

```
VOID tyMonitorTrapSet (ch)
    char  ch;  /* character to be made the monitor trap character */
```

DESCRIPTION

This routine sets the trap-to-monitor character to *ch*. The default trap-to-monitor character is ^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 Vx960 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

tyIoctl( )

NAME

tyIoctl( ) - handle device control requests

SYNOPSIS

```
STATUS tyIoctl (pTyDev, request, arg)
    TY_DEV_ID  pTyDev;   /* pointer 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

tyWrite( )

NAME

tyWrite( ) - do a task-level write for a tty device

SYNOPSIS

```
int tyWrite (pTyDev, buffer, nbytes)
    TY_DEV_ID  pTyDev;   /* pointer 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

tyRead( )

NAME

tyRead( ) - do a task-level read for a tty device

SYNOPSIS

```
int tyRead (pTyDev, buffer, maxbytes)
    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 up to *maxbytes* available bytes into the buffer.

This routine should only be called from serial device drivers.

RETURNS

The number of bytes actually read into the buffer.

SEE ALSO

tyLib

tyITx( )

NAME

tyITx( ) - interrupt-level output

SYNOPSIS

```
STATUS tyITx (pTyDev, pChar)
    TY_DEV_ID  pTyDev;  /* pointer to tty device descriptor       */
    char       *pChar;  /* ptr 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

tyIRd( )

NAME

tyIRd( ) - interrupt level input

SYNOPSIS

```
STATUS tyIRd (pTyDev, inchar)
    TY_DEV_ID  pTyDev;  /* pointer 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

usrConfig

NAME

usrConfig - user-defined system configuration module

SYNOPSIS

usrInit( ) - user-defined system initialization routine
usrRoot( ) - the root task
usrClock( ) - user-defined system clock interrupt routine
usrScsiConfig( ) - configure a SCSI peripheral (example)

```
VOID usrInit (startType)
VOID usrRoot ()
VOID usrClock ()
STATUS usrScsiConfig ()
```

DESCRIPTION

This is the Intel-supplied configuration module for Vx960. 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.

INCLUDE FILE

config.h

SEE ALSO

Programmer's Guide: Getting Started, Cross-Development

usrInit( )

NAME

usrInit( ) - user-defined system initialization routine

SYNOPSIS

```
VOID usrInit (startType)
    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, so 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 ()
```

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 the 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 Vx960 shell.

RETURNS

N/A

SEE ALSO

usrConfig

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

usrScsiConfig( )

NAME

usrScsiConfig( ) - configure a SCSI peripheral (example)

SYNOPSIS

```
STATUS usrScsiConfig ()
```

DESCRIPTION

This routine 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.

If you are just getting started, you can test your hardware configuration by defining SCSI_AUTO_CONFIG in **config.h**, which will probe the bus and display all devices found. No device should have the same SCSI bus ID as your Vx960 SCSI port (default = 7), or the same as any other device. Check for proper bus termination.

As an aid to debugging, either of the variables *scsiDebug* or *scsiIntsDebug* can be set to TRUE (1). When the hardware is working, be sure to undefine SCSI_AUTO_CONFIG. Of course, you must rework the rest of this routine to conform to your own hardware, as well as associated partitioning and file system mappings.

In this example, there are two disk drives on the bus, an 80-Mbyte Winchester disk (ID=2, LUN=0) and a 1.2-Mbyte 5.25" floppy drive (ID=3, LUN=1). The floppy is actually interfaced via an OMTI 3500 universal SCSI-to-floppy adapter.

The Winchester 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 Vx960 from the device. This will simplify the task considerably.

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.

While the Winchester configuration is fairly straightforward, the floppy setup in this example is a bit intricate. Note that the ***scsiPhysDevCreate*( )** call is issued twice. The first time is merely to get a "handle" to pass to the ***scsiModeSelect*( )** function, 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 the ***scsiPhysDevDelete*( )** call, after which a second ***scsiPhysDevCreate*( )** call correctly configures the peripheral. (Before the ***scsiModeSelect*( )** call, the configuration information was incorrect.) Also note that following the ***scsiBlkDevCreate*( )** call, the correct values for *sectorsPerTrack* and *nHeads* must be set via the ***scsiBlkDevInit*( )** call. This is necessary for IBM PC compatibility.

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 Vx960 shell, i.e.:

```
-> dosFsMkfs "/fd0/", pSbdFloppy
```

If a disk is new, use ***diskFormat*( )** to format it.

RETURNS

OK or ERROR.

SEE ALSO

usrConfig, *Programmer's Guide: I/O System, Local File Systems*

usrLib

NAME

usrLib - user interface subroutine library

SYNOPSIS

help( ) - print a synopsis of selected routines
netHelp( ) - print a synopsis of network routines
bootChange( ) - change the boot line
periodRun( ) - call a function periodically
period( ) - spawn a task to call a function periodically
repeatRun( ) - call a function repeatedly
repeat( ) - spawn a task to call a function repeatedly
sp( ) - spawn a task with default parameters
checkStack( ) - print a summary of each task's stack usage
i( ) - print a summary of each task's TCB
ts( ) - suspend a task
tr( ) - resume a task
td( ) - delete a task
ti( ) - print complete information from a task's TCB
version( ) - print Vx960 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 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 global symbols
lkAddr( ) - list symbols whose values are near a given value
pfp( ) - return the contents of register pfp (also sp, rip, r3-r15, g0-g14, fp)
fp0( ) - return the contents of register fp0 (also fp1-3, 80960KB/SB only)
pcw( ) - return the contents of the PCW register

tcw( ) - return the contents of the TCW register
acw( ) - return the contents of the ACW register
mRegs( ) - modify registers
printErrno( ) - print the definition of a specified error status value
printLogo( ) - print the Vx960 logo
logout( ) - log out of the Vx960 system
h( ) - display (or set) shell history

```
VOID help ()
VOID netHelp ()
VOID bootChange ()
VOID periodRun (secs, func, arg1, arg2, arg3, arg4, arg5, ...
int period (secs, func, arg1, arg2, arg3, arg4, arg5, ...
VOID repeatRun (n, func, arg1, arg2, arg3, arg4, arg5, ...
int repeat (n, func, arg1, arg2, arg3, arg4, arg5, ...
int sp (func, arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8, arg9)
VOID checkStack (taskNameOrId)
VOID i (taskNameOrId)
VOID ts (taskNameOrId)
VOID tr (taskNameOrId)
VOID td (taskNameOrId)
VOID ti (taskNameOrId)
VOID version ()
VOID m (adrs)
VOID d (adrs, nwords)
STATUS cd (name)
VOID pwd ()
STATUS copy (in, out)
STATUS copyStreams (inFd, outFd)
STATUS diskFormat (devName)
STATUS diskInit (devName)
STATUS squeeze (devName)
STATUS ld (syms, noAbort, name)
STATUS ls (dirName, doLong)
STATUS ll (dirName)
STATUS lsOld (dirName)
STATUS mkdir (dirName)
STATUS rmdir (dirName)
STATUS rm (fileName)
VOID devs ()
VOID lkup (substr)
VOID lkAddr (addr)
int pfp (taskId)
double fp0 (taskId)
```

```
int pcw (taskId)
int tcw (taskId)
int acw (taskId)
STATUS mRegs (taskNameOrId)
VOID printErrno (errno)
VOID printLogo ()
VOID logout ()
VOID h (size)
```

DESCRIPTION

This library consists of routines meant to be executed from the Vx960 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 Vx960. Users should feel free to modify or extend this library. It may be preferable to customize capabilities by creating a new private library, using this one as a model, and appropriately linking it 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. **usrLib** uses *taskIdDefault*( ) to set and get the last-referenced task ID, as do many other Vx960 routines.

SEE ALSO

Programmer's Guide: Shell, Debugging

help( )

NAME

help( ) - print a synopsis of selected routines

SYNOPSIS

```
VOID help ()
```

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[,nwords]]    Display memory
m          adr               Modify memory
mRegs      [task]            Modify a task's registers interactively
r3-r15, g0-g14      [task]   Display a register of a task (80960ca version)
fp0-fp1    [task]            Display a floating point register of a task (80960KB/
version                      Print Vx960 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

netHelp( )

NAME

netHelp( ) - print a synopsis of network routines

SYNOPSIS

```
VOID netHelp ()
```

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
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 "mars", "90.0.0.2"
          -> netDevCreate "mars:", "mars", 0
          -> iam "fred"
          -> copy <mars:/etc/passwd   /* copy file from host "mars" *
          -> rlogin "mars"            /* rlogin to host "mars" *
```

RETURNS

N/A

SEE ALSO

usrLib

bootChange( )

NAME

bootChange( ) - change the boot line

SYNOPSIS

```
VOID bootChange ()
```

DESCRIPTION

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

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

RETURNS

N/A

SEE ALSO

usrLib

periodRun( )

NAME

periodRun( ) - call a function periodically

SYNOPSIS

```
VOID periodRun (secs, func, arg1, arg2, arg3, arg4, arg5,
       arg6, arg7, arg8)
    int secs;             /* number of seconds to delay between calls */
    FUNCPTR func;         /* function to call repeatedly              */
    int arg1, arg2, arg3; /* args to pass to func                     */
    int arg4, arg5, arg6; /* args to pass to func                     */
    int arg7, arg8;       /* args to pass to func                     */
```

DESCRIPTION

This routine 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*( )

period( )

NAME

period( ) - spawn a task to call a function periodically

SYNOPSIS

```
int period (secs, func, arg1, arg2, arg3, arg4, arg5,
            arg6, arg7, arg8)
    int secs;             /* period (in seconds)  */
    FUNCPTR func;         /* function to call     */
    int arg1, arg2, arg3; /* args to pass to func */
    int arg4, arg5, arg6; /* args to pass to func */
    int arg7, arg8;       /* args to pass to func */
```

DESCRIPTION

This command spawns a task that will repeatedly call 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*( )

repeatRun( )

NAME

repeatRun( ) - call a function repeatedly

SYNOPSIS

```
VOID repeatRun (n, func, arg1, arg2, arg3, arg4, arg5,
          arg6, arg7, arg8)
    int n;                /* # of times to call func (0=forever) */
    FUNCPTR func;         /* function to call repeatedly         */
    int arg1, arg2, arg3; /* args to pass to func                */
    int arg4, arg5, arg6; /* args to pass to func                */
    int arg7, arg8;       /* args to pass to func                */
```

DESCRIPTION

This routine 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*( )

repeat( )

NAME

repeat( ) - spawn a task to call a function repeatedly

SYNOPSIS

```
int repeat (n, func, arg1, arg2, arg3, arg4, arg5,
         arg6, arg7, arg8)
    int n;                /* # of times to call function (0=forever) */
    FUNCPTR func;         /* function to call repeatedly             */
    int arg1, arg2, arg3; /* args to pass to func                    */
    int arg4, arg5, arg6; /* args to pass to func                    */
    int arg7, arg8;       /* args to pass to func                    */
```

DESCRIPTION

This command spawns a task that will call a specified function *n* times, with up to eight of its arguments. If *n* is 0, the routine will be 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*( )

sp( )

NAME

sp( ) - spawn a task with default parameters

SYNOPSIS

```
int sp (func, arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8, arg9)
    FUNCPTR func;               /* function to call */
    int arg1, arg2, arg3;       /* passed to spawned task */
    int arg4, arg5, arg6;       /* passed to spawned task */
    int arg7, arg8, arg9;       /* passed to spawned task */
```

DESCRIPTION

This command spawns a specified function as a task with priority 100, the default options listed below, a 20000-byte stack, the highest task ID currently not used, and a default task name. The default options are:

VX_FP_TASK - execute with floating point support

VX_STDIO - execute with standard I/O support

The task ID is displayed after the task is spawned.

The default name assigned to the task is of the form "t*n*" where *n* is an integer which increments as new tasks are spawned, e.g., *t1*, *t2*, *t3*, etc.

This routine 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*( )**

checkStack( )

NAME

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

SYNOPSIS

```
VOID checkStack (taskNameOrId)
    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 from the top of the stack for the first byte whose value is not 0xee. In Vx960, 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, which will be undetected by *checkStack*( ).

RETURNS

N/A

SEE ALSO

usrLib, *taskSpawn*( )

i( )

NAME

i( ) - print a summary of each task's TCB

SYNOPSIS

```
VOID i (taskNameOrId)
    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*( ) command provides more complete information on a specific task.

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 routine 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*( )

ts( )

NAME

ts( ) - suspend a task

SYNOPSIS

```
VOID ts (taskNameOrId)
    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*( )

tr( )

NAME

tr( ) - resume a task

SYNOPSIS

```
VOID tr (taskNameOrId)
    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*( )

td( )

NAME

td( ) - delete a task

SYNOPSIS

```
VOID td (taskNameOrId)
    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*( )

ti( )

NAME

ti( ) - print complete information from a task's TCB

SYNOPSIS

```
VOID ti (taskNameOrId)
    int taskNameOrId; /* task name or task ID; 0 = use default */
```

DESCRIPTION

This command prints the TCB contents, including registers, for a specified task. If *taskNameOrId* is omitted or zero, the last task referenced is assumed.

EXAMPLE

The following shows the TCB contents for the shell task:

```
-> ti

   NAME        ENTRY        TID    PRI   STATUS      PC       SP     ERRNO  DELAY
---------- ------------ -------- --- ---------- -------- -------- ------- -----
tNetTask   _netTask      7ef2a0   50 PEND        1b064   7ef530        0      0
```

```
stack: base 0x7ef450  end 0x7f1b60  size 9488   high 767    margin 8721
options: 0x7
VX_SUPERVISOR_MODE  VX_UNBREAKABLE      VX_DEALLOC_STACK
pfp:    7ef450  sp :    7ef530  rip:     1b064  r3 : eeeeeeee
r4 :         0  r5 : eeeeeeee  r6 : eeeeeeee  r7 : eeeeeeee
r8 : eeeeeeee  r9 : eeeeeeee  r10: eeeeeeee  r11:   7ef2a0
r12:     1b050  r13: eeeeeeee  r14: eeeeeeee  r15: eeeeeeee
g0 :         0  g1 :         0  g2 :   7ef2a0  g3 :    318b8
g4 :         0  g5 :         0  g6 : ffffffff  g7 :       ff
g8 :         0  g9 :         0  g10:        0  g11:        0
g12:         0  g13:    7ef510  g14:        0  fp :   7ef4d0
pcw:  d86088ff  acw:      1102  tcw: f001ff81
ip  : 0x0 pcw: 0x0 acw: 0x0
Task: 0x7ef2a0 "tNetTask"
value = 26 = 0x1a
->
```

RETURNS

N/A

SEE ALSO

usrLib, *Programmer's Guide: Debugging*

version( )

NAME

version( ) - print Vx960 version information

SYNOPSIS

```
VOID version ()
```

DESCRIPTION

This command prints the Vx960 version number, the date this copy of Vx960 was made, and other pertinent information.

EXAMPLE

```
-> version
Vx960 (for Heurikon HK80/V960E) version 5.0.2
Kernel: WIND version 2.0.
Made on Tue Jul 2 10:59:01 PDT 1991.
Boot line:
ei(0,0)mars:/usr/vw/config/hkv960/vxWorks e=143.185.6.89 h=143.185.6.
```

```
9 u=target
value = 103 = 0x67 = 'g'
->
```

RETURNS

N/A

SEE ALSO

usrLib

m( )

NAME

m( ) - modify memory

SYNOPSIS

```
VOID m (adrs)
    char *adrs;          /* address to change */
```

DESCRIPTION

This command prompts the user for modifications to memory, starting at the specified address. It prints each address and the current contents of that address, in turn. 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. Memory is treated as 16-bit words.

RETURNS

N/A

SEE ALSO

usrLib, *mRegs*( )

d( )

NAME

d( ) - display memory

SYNOPSIS

```
VOID d (adrs, nwords)
    char *adrs; /* address to display                     */
    int nwords; /* number of words to print; if 0, use default */
```

DESCRIPTION

This command displays the contents of memory, starting at *adrs*. If *adrs* is omitted, *d*( ) displays the next memory block, starting from where the last *d*( ) command completed.

Memory is displayed in words. If *nwords* is zero or absent, the number of words displayed defaults to 64. If *nwords* is non-zero, that number of words is displayed and that number then becomes the default. The number of words *d*( ) displays is rounded up to the nearest number of full lines.

RETURNS

N/A

SEE ALSO

usrLib, *m*( )

cd( )

NAME

cd( ) - change the default directory

SYNOPSIS

```
STATUS cd (name)
    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 Vx960 network device must already have been created using ***netDevCreate*( )**.

EXAMPLES

The following example changes the directory to device "/fd0/":

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

This example changes the directory to device "mars:" with the local directory "~leslie/vw":

```
-> cd "mars:~leslie/vw"
```

After the previous command, the following changes the directory to "mars:~leslie/vw/config":

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

After the previous command, the following changes the directory to "mars:~leslie/vw/demo":

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

After the previous command, the following changes the directory to "mars:/etc".

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

Note that ~ can be used only on network devices (*rsh* or *ftp*).

RETURNS

OK or ERROR.

SEE ALSO

usrLib, *pwd*( )

pwd( )

NAME

pwd( ) - print the current default directory

SYNOPSIS

```
VOID pwd ()
```

DESCRIPTION

This command displays the current working device/directory.

RETURNS

N/A

SEE ALSO

usrLib, *cd*( )

copy( )

NAME

copy( ) - copy *in* (or *stdin*) to *out* (or *stdout*)

SYNOPSIS

```
STATUS copy (in, out)
    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 ^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 file named "file1" to 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*( )

copyStreams( )

NAME

copyStreams( ) - copy from/to specified streams

SYNOPSIS

```
STATUS copyStreams (inFd, outFd)
    int inFd;   /* file descriptor of stream to copy from */
    int outFd;  /* file descriptor of stream to copy to */
```

DESCRIPTION

This routine copies from the stream identified by *inFd* to the stream identified by *outFd* until an end of file is reached in *inFd*. This routine is used by *copy*( ).

RETURNS

OK, or ERROR if there is an error reading from *inFd* or writing to *outFd*.

SEE ALSO

usrLib, *copy*( )

diskFormat( )

NAME

diskFormat( ) - format a disk

SYNOPSIS

```
STATUS diskFormat (devName)
    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 routine 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

diskInit( )

NAME

diskInit( ) - initialize a file system on a block device

SYNOPSIS

```
STATUS diskInit (devName)
    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 routine 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

squeeze( )

NAME

squeeze( ) - reclaim fragmented free space on an RT-11 volume

SYNOPSIS

```
STATUS squeeze (devName)
    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

ld( )

NAME

ld( ) - load object module into memory

SYNOPSIS

```
STATUS ld (syms, noAbort, name)
    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 GNU/960 *b.out* or *coff* format. External references in the module are resolved during loading. The *syms* parameter determines how symbols are loaded; possible values are:

0 - Add global symbols to the system symbol table.

1 - Add global and local symbols to the system symbol table.

-1 - Add no symbols to the system symbol table.

If there is an error during loading (e.g., externals undefined, too many symbols, etc.), then *shellScriptAbort*( ) will be 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 *b.out* or *coff* 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

OK, or ERROR if there are too many symbols, the object file format is invalid, or there is an error reading the file.

SEE ALSO

usrLib, loadLib

ls( )

NAME

ls( ) - list the contents of a directory

SYNOPSIS

```
STATUS ls (dirName, doLong)
    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. If doing a long listing, any entries that describe subdirectories will also be flagged with a "DIR" comment.

The *dirName* parameter specifies the directory to be listed. If *dirName* is omitted or NULL, the current working directory will be listed.

Empty directory entries and MS-DOS volume label entries are not reported.

ls does not work with ftp or rsh network drives. Use lsOld instead. The error message for rsh or ftp network drives is "Can't open".

EXAMPLE

```
    -> ls "/usr/vw"
```

RETURNS

OK or ERROR.

SEE ALSO

usrLib, *ll*( ), *lsOld*( ), *stat*( )

ll( )

NAME

ll( ) - do a long listing of directory contents

SYNOPSIS

```
STATUS ll (dirName)
    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:

```
    -> dirName="/usr/vw"
    -> ll dirName, TRUE
```

RETURNS

OK or ERROR.

SEE ALSO

usrLib, *ls*( ), *stat*( )

lsOld( )

NAME

lsOld( ) - list the contents of an RT-11, ftp, or rsh directory

SYNOPSIS

```
STATUS lsOld (dirName)
    char *dirName;        /* device to list */
```

DESCRIPTION

This is the old version of *ls*( ), which used the old-style *ioctl*( ) function FIODIRENTRY to get information about entries in a directory. With Vx960 release 5.0, this has been replaced with a new version of *ls*( ) which uses POSIX directory and file functions.

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 *rsh* and *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*( )

mkdir( )

NAME

mkdir( ) - make a directory

SYNOPSIS

```
STATUS mkdir (dirName)
    char *dirName;              /* directory name */
```

DESCRIPTION

This command is provided for UNIX similarity. It works only on an NFS device.

RETURNS

OK, or ERROR if the directory cannot be created.

SEE ALSO

usrLib, *rmdir*( )

rmdir( )

NAME

rmdir( ) - remove a directory

SYNOPSIS

```
STATUS rmdir (dirName)
    char *dirName;
```

DESCRIPTION

This command is provided for UNIX similarity. It works only on an NFS device.

RETURNS

OK, or ERROR if the directory cannot be removed.

SEE ALSO

usrLib, *mkdir*( )

rm( )

NAME

rm( ) - remove a file

SYNOPSIS

```
STATUS rm (fileName)
    char *fileName;
```

DESCRIPTION

This command is provided for UNIX similarity. It simply calls ***delete***( ).

RETURNS

OK, or ERROR if the file cannot be removed.

SEE ALSO

usrLib, *delete*( )

devs( )

NAME

devs( ) - list all system-known devices

SYNOPSIS

```
VOID devs ()
```

DESCRIPTION

This command displays a list of all devices known to the I/O system.

EXAMPLE

```
-> devs
drv name
  0 /null
  1 /tyCo/0
  1 /tyCo/1
  7 mars:
  8 /pty/rlogin.S
  9 /pty/rlogin.M
  8 /pty/telnet.S
  9 /pty/telnet.M
 10 /
 10 /usr
->
```

RETURNS

N/A

SEE ALSO

usrLib, *iosDevShow*( )

lkup( )

NAME

lkup( ) - list global symbols

SYNOPSIS

```
VOID lkup (substr)
    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 an empty string (""), all symbols in the table will be 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*( ).

RETURNS

N/A

SEE ALSO

usrLib, symLib, *symEach*( )

lkAddr( )

NAME

lkAddr( ) - list symbols whose values are near a given value

SYNOPSIS

```
VOID lkAddr (addr)
    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*( )

pfp()

NAME

pfp( ) - return the contents of register pfp (also sp, rip, r3-r15, g0-g14, fp)

SYNOPSIS

```
int pfp (taskId)
    int taskId;         /* task's ID, 0 means default task */
```

DESCRIPTION

This command extracts the contents of register pfp for a specified task from the task's TCB. If *taskId* is omitted or 0, the current default task is used.

The values are not guaranteed for the current executing task.

The following similar routines are also available for accessing the contents of their associated registers:

r3( ) - *r15*( )	- all local registers
g0( ) - *g14*( )	- global registers
tsp( )	- the stack pointer
rip( )	- the return instruction pointer
fp( )	- the frame pointer
fp0( ) - *fp3*( )	- floating point registers (80960KB and 80960SB only)
pcw( )	- the processor control word
acw( )	- the arithmetic control word
tcw( )	- the trace control word

NOTE

To get the stack pointer of a task, use *tsp*( ), because *sp*( ) (the logical name choice) conflicts with the routine *sp*( ) for spawning a task with default parameters.

The return type for *fp0*( )-*fp3*( ) is double, not int. To get their return values printed out correctly on the shell command line, invoke them with the appropriate cast (as described in the Programmer's Guide, section 9.3.1 Data Types).

```
-> (double ( ) ) fp0 ( taskId )
```

RETURNS

The contents of the requested register.

SEE ALSO

usrLib, *Programmer's Guide: Debugging*
80960CA User's Manual

pcw( )

NAME

pcw( ) - return the contents of the PCW register

SYNOPSIS

```
int pcw (taskId)
    int taskId;         /* task's ID, 0 means default task */
```

DESCRIPTION

This command extracts the contents of the PCW register for a specified task from the task's TCB. If *taskId* is omitted or 0, the default task is used.

RETURNS

The contents of the register.

SEE ALSO

usrLib, *Programmer's Guide: Debugging* 80960CA User's Manual

tcw( )

NAME

tcw( ) - return the contents of the TCW register

SYNOPSIS

```
int tcw (taskId)
    int taskId;         /* task's ID, 0 means default task */
```

DESCRIPTION

This command extracts the contents of the TCW register for a specified task from the task's TCB. If *taskId* is omitted or 0, the default task is used.

RETURNS

The contents of the TCW register

SEE ALSO

usrLib, *Programmer's Guide: Debugging* 80960CA User's Manual.

acw( )

NAME

acw( ) - return the contents of the ACW register

SYNOPSIS

```
int acw (taskId)
    int taskId;         /* task's ID, 0 means default task */
```

DESCRIPTION

This command extracts the contents of the ACW register for a specified task from the task's TCB. If *taskId* is omitted or 0, the default task is used.

RETURNS

The contents of the ACW register.

SEE ALSO

usrLib, *Programmer's Guide: Debugging* 80960CA User's Manual.

mRegs( )

NAME

mRegs( ) - modify registers

SYNOPSIS

```
STATUS mRegs (taskNameOrId)
    int taskNameOrId; /* task name or task ID, 0 = default task */
```

DESCRIPTION

This command sequentially prompts the user for new values for a task's registers. If *taskNameOrId* is omitted or zero, the last task referenced is assumed.

mRegs( ) prompts the user for modifications starting at register d0. 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 does not exist.

SEE ALSO

usrLib, *m*( )

printErrno( )

NAME

printErrno( ) - print the definition of a specified error status value

SYNOPSIS

```
VOID printErrno (errno)
    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*( )

printLogo( )

NAME

printLogo( ) - print the Vx960 logo

SYNOPSIS

```
VOID printLogo ()
```

DESCRIPTION

This routine displays the Vx960 banner seen at boot time. It also displays the Vx960 version number and kernel version number.

RETURNS

N/A

SEE ALSO

usrLib

logout( )

NAME

logout( ) - log out of the Vx960 system

SYNOPSIS

```
VOID logout ()
```

DESCRIPTION

This routine logs out of the Vx960 shell. If a remote login in active (via *rlogin* or *telnet*), it is stopped, and standard I/O is restored to the console.

SEE ALSO

usrLib, *rlogin*( ), *telnet*( ), *shellLogout*( )

h( )

NAME

h( ) - display (or set) shell history

SYNOPSIS

```
VOID h (size)
    int size; /* 0 = display, >0 then set history to new size */
```

DESCRIPTION

With no argument, this command displays Vx960 shell history. If *size* is specified, that number of the most recent commands will be saved for display. The value of *size* is initially 20.

RETURNS

N/A

SEE ALSO

usrLib, *shellHistory*( ), ledLib

vxALib

NAME

vxALib - miscellaneous assembly language routines

SYNOPSIS

vxTas( ) - C-callable atomic test-and-set primitive

```
BOOL vxTas (address)
```

DESCRIPTION

This module contains miscellaneous Vx960 support routines.

SEE ALSO

vxLib

vxTas( )

NAME

vxTas( ) - C-callable atomic test-and-set primitive

SYNOPSIS

```
BOOL vxTas (address)
    char  *address;  /* address to be tested *
```

DESCRIPTION

This routine provides a C-callable interface to the i960 atomic-modify instruction. The "atmod" instruction is executed on the specified address.

RETURNS

TRUE if the value had not been set, but now is; FALSE if the value was already set.

SEE ALSO

vxALib

vxLib

NAME

vxLib - miscellaneous support routines

SYNOPSIS

vxMemProbe( ) - probe an address for bus error

```
STATUS vxMemProbe (adrs, mode, length, pVal)
```

DESCRIPTION

This module contains miscellaneous Vx960 support routines.

SEE ALSO

vxALib

vxMemProbe( )

NAME

vxMemProbe( ) - probe an address for bus error

SYNOPSIS

```
STATUS vxMemProbe (adrs, mode, length, pVal)
    char  *adrs;   /* address to be probed          */
    int   mode;    /* READ or WRITE                 */
    int   length;  /* 1, 2, or 4                    */
    char  *pVal;   /* where to return value,        */
                   /* or ptr to value to be written */
```

DESCRIPTION

This routine probes a specified address to see if it is readable or writable, as specified by *mode*. The address will be 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 READ, the value read will be copied to the location pointed to by *pVal*. If the probe is a WRITE, the value written will be 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., not generate an address error).

EXAMPLE

```
testMem (adrs)
    char *adrs;
    {
    char testW = 1;
    char testR;

    if (vxMemProbe (adrs, WRITE, 1, &testW) == OK)
        printf ("value %d written to adrs %x\n", testW, adrs);

    if (vxMemProbe (adrs, READ, 1, &testR) == OK)
        printf ("value %d read from adrs %x\n", testR, adrs);
    }
```

RETURNS

OK if the probe is successful, or ERROR if the probe caused a bus error or an address misalignment.

SEE ALSO

vxLib

wdLib

NAME

wdLib - watchdog timer library

SYNOPSIS

wdCreate( ) - create a watchdog timer
wdDelete( ) - delete a watchdog timer
wdStart( ) - start a watchdog timer
wdCancel( ) - cancel a currently counting watchdog

```
WDOG_ID wdCreate ()
STATUS wdDelete (wdId)
STATUS wdStart (wdId, delay, pRoutine, parameter)
STATUS wdCancel (wdId)
```

DESCRIPTION

This library provides a general watchdog timer facility. Any task may create a watchdog timer, then use it to provide events that can happen after a specified delay, outside the context of the task itself.

Once a timer has been created with *wdCreate*( ), it can be started with *wdStart*( ). The *wdStart*( ) routine takes as parameters a timeout routine, an arbitrary timeout routine parameter, and a timeout in ticks. (This will be the number of ticks as determined by the system clock; see *sysClkRateSet*( ) for more information.) After the specified delay ticks have elapsed, and if the timer has not been canceled with *wdCancel*( ), the timeout routine will be invoked with the parameter that was specified with *wdStart*( ). The timeout routine will be invoked even if the task which started the watchdog is running, suspended or deleted.

Note that the timeout routine is invoked at interrupt level, rather than in the context of the task. Therefore, it must be careful about what it can and cannot do. Watchdog routines are constrained to the same rules as interrupt service routines. For example, they may not take semaphores.

EXAMPLE

```
WDOG_ID wid = wdCreate ();
wdStart (wid, 60, logMsg, "Help, I've timed out!");
maybeSlowRoutine ();
wdCancel (wid);
```

In the above fragment, 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.

INCLUDE FILE

wdLib.h

SEE ALSO

logLib, *Programmer's Guide: Basic OS, Cross-Development*

wdCreate( )

NAME

wdCreate( ) - create a watchdog timer

SYNOPSIS

```
WDOG_ID wdCreate ()
```

DESCRIPTION

This routine creates a watchdog timer by allocating a WDOG structure from memory.

RETURNS

The ID for the watchdog created, or NULL if memory is insufficient.

SEE ALSO

wdLib

wdDelete( )

NAME

wdDelete( ) - delete a watchdog timer

SYNOPSIS

```
STATUS wdDelete (wdId)
    WDOG_ID  wdId;  /* watchdog id 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

wdStart( )

NAME

wdStart( ) - start a watchdog timer

SYNOPSIS

```
STATUS wdStart (wdId, delay, pRoutine, parameter)
    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 */
```

VOID FUNCPTR

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. Use *wdCancel( )* to remove the watchdog from the system time queue.

RETURNS

OK, or ERROR if the watchdog timer cannot be started.

SEE ALSO

wdLib, *wdCancel( )*

wdCancel( )

NAME

wdCancel( ) - cancel a currently counting watchdog

SYNOPSIS

```
STATUS wdCancel (wdId)
    WDOG_ID  wdId;  /* id of watchdog to cancel */
```

DESCRIPTION

This routine cancels a watchdog timer that is currently running by zeroing its delay count. Watchdog timers may be canceled from interrupt level.

RETURNS

OK, or ERROR if the watchdog timer cannot be cancelled.

SEE ALSO

wdLib, *wdStart*( )

Drivers

CONTENTS

mb87030Lib	- Fujitsu mb87030 SCSI Protocol Controller (SPC) Library	2-1
memDrv	- pseudo memory device driver	2-5
netDrv	- network remote file I/O driver	2-8
nfsDrv	- Network File System I/O driver	2-12
pipeDrv	- pipe I/O driver	2-18
ptyDrv	- pseudo-terminal driver	2-22
ramDrv	- RAM disk driver	2-25
wd33c93Lib	- library for the wd33c93 SCSI-Bus Interface Controller (SBIC)..	2-29

mb87030Lib

NAME

mb87030Lib - Fujitsu mb87030 SCSI Protocol Controller (SPC) Library

SYNOPSIS

mb87030CtrlCreate( ) - create a control structure for an mb87030 SPC
mb87030CtrlInit( ) - initialize a control structure for an mb87030 SPC
spcShow( ) - display the values of all readable mb87030 (SPC) registers

```
MB_87030_SCSI_CTRL *mb87030CtrlCreate (spcBaseAdrs, regOffset, clkPeriod, ...
STATUS mb87030CtrlInit (pSpc, scsiCtrlBusId, defaultSelTimeOut, scsiPriority)
VOID spcShow (pSpc)
```

DESCRIPTION

This is the I/O driver for the Fujitsu mb87030 SCSI Protocol Controller (SPC) chip. It is designed to work in conjunction with the **scsiLib** library.

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, *Programmer's Guide: I/O System*

mb87030CtrlCreate( )

NAME

mb87030CtrlCreate( ) - create a control structure for an mb87030 SPC

SYNOPSIS

```
MB_87030_SCSI_CTRL *mb87030CtrlCreate (spcBaseAdrs, regOffset, clkPeriod,
                                       spcDataParity, spcDmaBytesIn,
                                       spcDmaBytesOut)
    UINT8      *spcBaseAdrs;     /* base address of the SPC */
    int        regOffset;        /* address offset between consecutive regs. */
    UINT       clkPeriod;        /* period of the controller clock (nsec) */
    int        spcDataParity;    /* type of input to SPC DP (data parity) */
    FUNCPTR  spcDmaBytesIn;    /* function for SCSI DMA input */
    FUNCPTR  spcDmaBytesOut;   /* function for SCSI DMA output */
```

DESCRIPTION

Before using the SPC chip, a data structure must be created by calling this routine. This routine should be called once and only once for a given SPC, and 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 (BDID) 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
- period in nanoseconds of the signal to SPC CLK input (only used for select timeouts).

spcDataParity
- must be one of:
  SPC_DATA_PARITY_LOW
  SPC_DATA_PARITY_HIGH
  SPC_DATA_PARITY_VALID
  according to whether the input to SPC DP is GND, +5V, or a valid parity signal.

spcDmaBytesIn and *spcDmaBytesOut*
- 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} (pScsiPhysDev, pBuffer, bufLength)

    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 unavailable or there are bad parameters.

SEE ALSO

mb87030Lib

mb87030CtrlInit( )

NAME

mb87030CtrlInit( ) - initialize a control structure for an mb87030 SPC

SYNOPSIS

```
STATUS mb87030CtrlInit (pSpc, scsiCtrlBusId, defaultSelTimeOut, scsiPriority)
    MB_87030_SCSI_CTRL  *pSpc;              /* ptr to SPC struct */
    int                 scsiCtrlBusId;      /* SCSI bus ID of this SPC */
    UINT                defaultSelTimeOut;  /* default device select time-out (usec) *
    int                 scsiPriority;       /* priority of a task when doing SCSI I/O
```

DESCRIPTION

After an SPC control structure is created with *mb87030CtrlCreate*( ), it must be initialized by calling this routine before using the SPC. It may be called more than once if desired. However, it should only be called while there is no activity on the SCSI interface, since the specified configuration is written to the SPC.

Before returning, this routine pulses RST (reset) on the SCSI bus, thus resetting all attached devices.

A detailed description of the input parameters follows:

pSpc - pointer to the MB_87030_SCSI_CTRL structure created with *mb87030CtrlCreate*( ).

scsiCtrlBusId

- the SCSI bus ID of the SPC. The ID is somewhat arbitrary (ranges between 0-7); the value 7, or highest priority, is usually assigned.

defaultSelTimeOut

- the timeout (in microseconds) for selecting a SCSI device attached to this controller; called default since the timeout may be specified for each device (see the manual entry for *scsiPhysDevCreate*( )). The recommended value 0 specifies SCSI_DEF_SELECT_TIMEOUT (250 ms). The maximum timeout possible is approximately 3 seconds. Values that exceed this revert to the maximum.

scsiPriority

- the priority to which a task is set when performing a SCSI transaction (ranging from 0 to 255). Otherwise, the value -1 indicates that the priority should not be altered during SCSI transactions.

RETURNS

OK, or ERROR if any parameters are out of range.

SEE ALSO

mb87030Lib

spcShow( )

NAME

spcShow( ) - display the values of all readable mb87030 (SPC) registers

SYNOPSIS

```
VOID spcShow (pSpc)
    MB_87030_SCSI_CTRL  *pSpc;  /* ptr to SPC struct */
```

DESCRIPTION

This routine displays the state of the SPC registers in a user-friendly way. It is useful primarily for debugging.

RETURNS

N/A

SEE ALSO

mb87030Lib

memDrv

NAME

memDrv - pseudo memory device driver

SYNOPSIS

memDrv( ) - install a memory driver
memDevCreate( ) - create a memory device

```
STATUS memDrv ()
STATUS memDevCreate (name, base, length)
```

DESCRIPTION

This driver allows the I/O system to access memory directly as a pseudo-I/O device. Memory location and size are specified when the device is created. This feature is useful when data must be preserved between boots of Vx960 or when sharing data between CPUs. This driver does not implement a file system as does **ramDrv**. The **ramDrv** driver must be given memory over which it has absolute control; **memDrv** simply provides a high-level method of reading and writing bytes in absolute memory locations through I/O calls.

USER-CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. Two routines, however, must be called directly: *memDrv*( ) to initialize the driver, and *memDevCreate*( ) to create devices.

Before using the driver, it must be initialized by calling *memDrv*( ). This routine should be called only once, before any reads, writes, or *memDevCreate*( ) calls. It may be called from *usrRoot*( ) in **usrConfig.c** or at some later point.

IOCTL

The memory driver responds to the *ioctl*( ) codes FIOSEEK and FIOWHERE.

SEE ALSO

Programmer's Guide: I/O System

memDrv( )

NAME

memDrv( ) - install a memory driver

SYNOPSIS

```
STATUS memDrv ()
```

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

memDevCreate( )

NAME

memDevCreate( ) - create a memory device

SYNOPSIS

```
STATUS memDevCreate (name, base, length)
    char  *name;   /* device name    */
    char  *base;   /* where to start in memory */
    int   length;  /* number of bytes  */
```

DESCRIPTION

This routine creates a memory device. Memory for the device is simply an absolute memory location beginning at *base*. The *length* parameter indicates the size of memory.

For example, to create the device "/mem/cpu0/", a device for accessing the entire memory of the local processor, the proper call would be:

```
    memDevCreate ("/mem/cpu0/", LOCAL_MEM_LOCAL_ADRS, sysMemTop())
```

The device is created with the specified name, start location, and size. LOCAL_MEM_LOCAL_ADRS is target specific and is defined in **config.h**.

To open a file descriptor to the memory, use *open*( ). Specify a pseudo-file name of the byte offset desired, or open the "raw" file at the beginning and specify a position to seek to. For example, the following call to *open*( ) allows memory to be read starting at decimal offset 1000.

```
-> fd = open ("/mem/cpu0/1000", O_RDONLY, 0)
```

Pseudo-file name offsets are scanned with "%d".

EXAMPLE

Consider a system configured with two CPUs in the backplane and a separate dual-ported memory board, each with 1 megabyte of memory. The first CPU is mapped at VMEbus address 0x00400000 (4 Meg.), the second at bus address 0x00800000 (8 Meg.), the dual-ported memory board at 0x00c00000 (12 Meg.). Three devices can be created on each CPU as follows. On processor 0:

```
-> memDevCreate ("/mem/local/", 0, sysMemTop())
...
-> memDevCreate ("/mem/cpu1/", 0x00800000, 0x00100000)
...
-> memDevCreate ("/mem/share/", 0x00c00000, 0x00100000)
```

On processor 1:

```
-> memDevCreate ("/mem/local/", 0, sysMemTop())
...
-> memDevCreate ("/mem/cpu0/", 0x00400000, 0x00100000)
...
-> memDevCreate ("/mem/share/", 0x00c00000, 0x00100000)
```

Processor 0 has a local disk. Data or an object module needs to be passed from processor 0 to processor 1. To accomplish this, processor 0 first calls:

```
-> copy </disk1/module.o >/mem/share/0
```

Processor 1 can then be given the load command:

```
-> ld </mem/share/0
```

RETURNS

OK, or ERROR if memory is insufficient or the I/O system cannot add the device.

SEE ALSO

memDrv

netDrv

NAME

netDrv - network remote file I/O driver

SYNOPSIS

netDrv( ) - install the network remote file driver
netDevCreate( ) - create a remote file device

```
STATUS netDrv ()
STATUS netDevCreate (devName, host, protocol)
```

DESCRIPTION

This driver provides facilities for accessing files transparently over the network via *ftp* or *rsh* (see **nfsLib** for file access via NFS). 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 exactly once, before any reads, writes, or calls to *netDevCreate*( ). If INCLUDE_NETWORK is defined in **configAll.h**, it is called from *usrRoot*( ) in **usrConfig.c**.

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 "vxhost", files can be accessed by creating a device called "vxhost:". See the manual entry for ***netDevCreate*( )** for more information.

IOCTL FUNCTIONS

The network driver responds to the following *ioctl*( ) functions:

FIOGETNAME - Gets the file name of the 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);
```

SEE ALSO

remLib, netLib, sockLib, ***hostAdd*( ),** *Programmer's Guide: Network*

netDrv( )

NAME

netDrv( ) - install the network remote file driver

SYNOPSIS

```
STATUS netDrv ()
```

DESCRIPTION

This routine initializes and installs the network driver. It must be called before other network remote file functions are performed. If INCLUDE_NETWORK is defined in **configAll.h**, it is called from *usrRoot*( ) in **usrConfig.c**.

RETURNS

OK or ERROR.

SEE ALSO

netDrv

netDevCreate( )

NAME

netDevCreate( ) - create a remote file device

SYNOPSIS

```
STATUS netDevCreate (devName, host, protocol)
    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 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 "vxhost", files can be accessed by creating a device called "vxhost:". Files can be accessed via rsh as follows:

```
    netDevCreate ("vxhost:", "vxhost", rsh);
```

The file /usr/dog on the UNIX system "vxhost" can now be accessed as "vxhost:/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*( )**

nfsDrv

NAME

nfsDrv - Network File System I/O driver

SYNOPSIS

nfsDrv( ) - install the NFS driver
nfsMount( ) - mount an NFS file system
nfsMountAll( ) - mount all file systems exported by a specified host
nfsDevShow( ) - display the mounted NFS devices
nfsUnmount( ) - unmount an NFS device

```
STATUS nfsDrv ()
STATUS nfsMount (host, fileSystem, localName)
STATUS nfsMountAll (host, clientName, quiet)
VOID nfsDevShow ()
STATUS nfsUnmount (localName)
```

DESCRIPTION

This driver provides facilities for transparently accessing files over the network via the Network File System (NFS). 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

nfsDrv( ) is called to initialize the driver; *nfsMount*( ) is called to mount file systems; and *nfsUnmount*( ) is called to unmount them.

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. If INCLUDE_NFS is defined in configAll.h, it is called from *usrRoot*( ) in usrConfig.c.

CREATING NFS DEVICES

In order to access a remote file system, an NFS device must be created by calling *nfsMount*( ). For example, to create the device "/myd0/" for the file system "/d0/" on the host "vxhost", call:

```
    nfsMount ("vxhost", "/d0/", "/myd0/");
```

The file "/d0/dog" on the host "vxhost" can now be accessed as "/myd0/dog".

If the third parameter to *nfsMount*( ) is NULL, Vx960 creates a device with the same name as the file system. For example, the call:

```
nfsMount ("vxhost", "/d0/", NULL);
```

or from the shell:

```
nfsMount "vxhost", "/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 have been already created with *hostAdd*( ). The routine *nfsDevShow*( ) displays the mounted NFS devices.

IOCTL

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);
```

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);
```

FIOREADDIR - Reads the next directory entry. The argument *dirStruct* is a pointer to a DIR directory descriptor. 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", READ);
```

```
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", READ);
status = ioctl (fd, FIOFSTATGET, &statStruct);
```

DEFICIENCIES

There is only one client handle/cache per task. Performance is poor if a task is accessing two or more NFS files.

Changing the *nfsCacheSize* variable after a file is open could cause adverse effects. However, changing it before opening any NFS file descriptors should not pose a problem. *nfsCacheSize* is of type unsigned integer.

SEE ALSO

dirLib, nfsLib, *hostAdd*( ), *ioctl*( ), *Programmer's Guide: Network*

nfsDrv( )

NAME

nfsDrv( ) - install the NFS driver

SYNOPSIS

```
STATUS nfsDrv ()
```

DESCRIPTION

This routine initializes and installs the NFS driver. It must be called before any other NFS calls. If INCLUDE_NFS is defined in **configAll.h**, it is called from ***usrRoot*( )** in **usrConfig.c**.

RETURNS

OK, or ERROR if there is no room for the driver.

SEE ALSO

nfsDrv

nfsMount( )

NAME

nfsMount( ) - mount an NFS file system

SYNOPSIS

```
STATUS nfsMount (host, fileSystem, localName)
    char *host;         /* name of remote host                    */
    char *fileSystem;   /* name of remote directory to mount      */
    char *localName;    /* local device name for remote directory */
                        /* (NULL = use fileSystem name)           */
```

DESCRIPTION

This routine mounts a remote file system. It creates a local device *localName* for a remote file system on a specified host. The host must have already been added to the local host table with *hostAdd*( ). If *localName* is NULL, the local name will be the same as the remote name.

RETURNS

OK, or ERROR if the driver is not installed, *host* is invalid, or the system is out of memory.

SEE ALSO

nfsDrv, *nfsUnmount*( ), *hostAdd*( )

nfsMountAll( )

NAME

nfsMountAll( ) - mount all file systems exported by a specified host

SYNOPSIS

```
STATUS nfsMountAll (host, clientName, quiet)
    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 by either all clients or only *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 mounts failed.

SEE ALSO

nfsDrv, *nfsMount*( )

nfsDevShow( )

NAME

nfsDevShow( ) - display the mounted NFS devices

SYNOPSIS

```
VOID nfsDevShow ()
```

DESCRIPTION

This routine displays the device names and their associated NFS file systems. For example:

```
-> nfsDevShow
device name          file system
-----------          -----------
/yuba1/              yuba:/yuba1
/vxhost1/               vxhost:/vxhost1
```

RETURNS

N/A

SEE ALSO

nfsDrv

nfsUnmount( )

NAME

nfsUnmount( ) - unmount an NFS device

SYNOPSIS

```
STATUS nfsUnmount (localName)
    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

pipeDrv

NAME

pipeDrv - pipe I/O driver

SYNOPSIS

pipeDrv( ) - initialize the pipe driver
pipeDevCreate( ) - create a pipe device

```
STATUS pipeDrv ()
STATUS pipeDevCreate (name, nMessages, nBytes)
```

DESCRIPTION

Pipes are virtual devices that let 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 Vx960 message queue facility to do the actual buffering and delivering of the 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. If INCLUDE_PIPES is defined in **configAll.h**, it is called by the root task *usrRoot*( ) in **usrConfig.c**.

CREATING PIPES

Before a pipe can be used, it must be created with *pipeDevCreate*( ). For example, to create the device pipe "/pipe/demo" with up to 10 messages of size 100 bytes, the proper call would be:

```
    pipeDevCreate ("/pipe/demo", 10, 100);
```

USING PIPES

Once a pipe has been created it can be opened, closed, read, and written just as 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, they differ significantly from UNIX pipes which are stream-oriented and do not preserve message boundaries.

WRITING TO PIPES FROM INTERRUPT SERVICE

Interrupt service routines can write to pipes providing one of several ways in which interrupt service routines 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 **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 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);
```

SEE ALSO

msgQLib, *Programmer's Guide: I/O System*

pipeDrv( )

NAME

pipeDrv( ) - initialize the pipe driver

SYNOPSIS

```
STATUS pipeDrv ()
```

DESCRIPTION

This routine initializes and installs the driver. It must be called before any pipes are created. If INCLUDE_PIPES is defined in **configAll.h**, it is called by the root task *usrRoot*( ) in **usrConfig.c**.

RETURNS

OK, or ERROR if the driver installation fails.

SEE ALSO

pipeDrv

pipeDevCreate( )

NAME

pipeDevCreate( ) - create a pipe device

SYNOPSIS

```
STATUS pipeDevCreate (name, nMessages, nBytes)
    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

ptyDrv

NAME

ptyDrv - pseudo-terminal driver

SYNOPSIS

ptyDrv( ) - initialize the pseudo-terminal driver
ptyDevCreate( ) - create a pseudo terminal

```
STATUS ptyDrv ()
STATUS ptyDevCreate (name, rdBufSize, wrtBufSize)
```

DESCRIPTION

The pseudo-terminal driver provides a tty-like interface between a master and slave process, typically in network applications. The master process simulates the "hardware" side of the driver (e.g., a USART serial chip), while the slave process is the application program that normally talks to the driver.

USER-CALLABLE ROUTINES

Most 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*( ). Normally, it is called from *usrRoot*( ) in **usrConfig.c**.

CREATING PSEUDO-TERMINAL DEVICES

Before a pseudo-terminal can be used, it must be created by calling *ptyDevCreate*( ):

```
   STATUS ptyDevCreate (name, rdBufSize, wrtBufSize)
 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 (the master and slave device), one called *name*M and the other *name*S, which 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**.

SEE ALSO

tyLib, *Programmer's Guide: I/O System*

ptyDrv( )

NAME

ptyDrv( ) - initialize the pseudo-terminal driver

SYNOPSIS

```
STATUS ptyDrv ()
```

DESCRIPTION

This routine initializes the pseudo-terminal driver. It must be called before any other routine in this module.

SEE ALSO

ptyDrv

ptyDevCreate( )

NAME

ptyDevCreate( ) - create a pseudo terminal

SYNOPSIS

```
STATUS ptyDevCreate (name, rdBufSize, wrtBufSize)
    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 the routine runs out of memory.

SEE ALSO

ptyDrv

ramDrv

NAME

ramDrv - RAM disk driver

SYNOPSIS

ramDrv( ) - prepare a RAM disk driver for use (optional)
ramDevCreate( ) - create a RAM disk device

```
STATUS ramDrv ()
BLK_DEV *ramDevCreate (ramAddr, bytesPerBlk, blksPerTrack, nBlocks, blkOffset)
```

DESCRIPTION

This driver emulates a disk driver, but actually keeps all data in memory. The memory location and size are specified when the "disk" is created. The RAM disk feature is useful when data must be preserved between boots of Vx960 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. The second routine, *ramDevCreate*( ), must be called directly, however, to create RAM disk devices.

Once the device has been created, it must be associated with a name and file system (DOS, RT-11, or raw disk) 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 for *ramDevCreate*( ) for a more detailed discussion.

IOCTL

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.

SEE ALSO

dosFsDevInit( ), *dosFsMkfs*( ), *rt11FsDevInit*( ), *rt11FsMkfs*( ), *rawFsDevInit*( ), *Programmer's Guide: I/O System, Local File Systems*

ramDrv( )

NAME

ramDrv( ) - prepare a RAM disk driver for use (optional)

SYNOPSIS

```
STATUS ramDrv ()
```

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 Vx960. Otherwise, there is no need to call this routine before using the RAM disk driver.

RETURNS

OK, always.

SEE ALSO

ramDrv

ramDevCreate( )

NAME

ramDevCreate( ) - create a RAM disk device

SYNOPSIS

```
BLK_DEV *ramDevCreate (ramAddr, bytesPerBlk, blksPerTrack, nBlocks, blkOffset)
    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;     /* Number of blocks to skip at
                          * beginning of physical 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. (Vx960 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 (DOS, RT-11, or raw disk). 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 DOS 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*( ), since the file system will already exist 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 Vx960. The contents of the RAM disk will then be preserved.

These same procedures apply for creating a RAM disk with RT-11 using *rt11FsDevInit*( ) and *rt11FsMkfs*( ), or creating a RAM disk with rawFs using *rawFsDevInit*( ).

RETURNS

A pointer to a block device (BLK_DEV) structure, or NULL if memory cannot be allocated for the device structure or RAM disk.

SEE ALSO

ramDrv

wd33c93Lib

NAME

wd33c93Lib - library for the wd33c93 SCSI-Bus Interface Controller (SBIC)

SYNOPSIS

wd33c93CtrlCreate( ) - create and partially initialize an SBIC structure
wd33c93CtrlInit( ) - initialize the user-specified fields in an SBIC structure
sbicShow( ) - display values of all readable wd33c93 chip registers

```
WD_33C93_SCSI_CTRL *wd33c93CtrlCreate (sbicBaseAdrs, regOffset, clkPeriod, ...
STATUS wd33c93CtrlInit (pSbic, scsiCtrlBusId, defaultSelTimeOut, scsiPriority)
STATUS sbicShow (pScsiCtrl)
```

DESCRIPTION

This is the I/O driver for the Western Digital wd33c93 SCSI-Bus Interface Controller (SBIC). It is designed to work in conjunction with the **scsiLib** library.

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: *wd33c93CtrlCreate*( ) to create a controller structure, and *wd33c93CtrlInit*( ) to initialize it.

INCLUDE FILES

wd33c93.h

SEE ALSO

Programmer's Guide: I/O System

wd33c93CtrlCreate( )

NAME

wd33c93CtrlCreate( ) - create and partially initialize an SBIC structure

SYNOPSIS

```
WD_33C93_SCSI_CTRL *wd33c93CtrlCreate (sbicBaseAdrs, regOffset, clkPeriod,
                                       sbicScsiReset, sbicDmaBytesIn,
                                       sbicDmaBytesOut)
    UINT8     *sbicBaseAdrs;           /* base address of the SBIC */
    int       regOffset;               /* address offset between consecutive regs. */
    UINT      clkPeriod;               /* period of the controller clock (nsec) */
    FUNCPTR   sbicScsiReset;           /* function to reset SCSI bus */
    FUNCPTR  sbicDmaBytesIn;    /* function for SCSI DMA input */
    FUNCPTR  sbicDmaBytesOut;   /* function for SCSI DMA output */
```

DESCRIPTION

This routine creates an SBIC data structure and must be called before using an SBIC chip. It should be called only once for a given 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: Only the non-multiplexed processor interface is supported.

A detailed description of the input parameters follows:

sbicBaseAdrs
- the address where the CPU would access the lowest (AUX STATUS) 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 only used for *select* timeouts.

spcDmaBytesIn and *spcDmaBytesOut*
- board-specific routines to handle DMA input and output; if these are NULL (0), SBIC program transfer mode is used. DMA is implemented only during SCSI data in/out phases. The interface to these DMA routines must be of the form:

```
STATUS xxDmaBytes{In, Out} (pScsiPhysDev, pBuffer, bufLength)

    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 unavailable or there are bad parameters.

SEE ALSO

wd33c93Lib

wd33c93CtrlInit( )

NAME

wd33c93CtrlInit( ) - initialize the user-specified fields in an SBIC structure

SYNOPSIS

```
STATUS wd33c93CtrlInit (pSbic, scsiCtrlBusId, defaultSelTimeOut, scsiPriority)
    SBIC  *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 *wd33c93CtrlCreate*( ). This structure must be initialized before the SBIC can be used. It may be called more than once if needed; however, it should only be called 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*( ).

scsiCtrlBusId

- the SCSI bus ID of the SBIC. Its value is somewhat arbitrary: seven (7), or highest priority, is conventional. The value must be in the range 0 - 7.

defaultSelTimeOut

- the timeout (in microseconds) for selecting a SCSI device attached to this controller; called "default" since the timeout may be specified per device (see *scsiPhysDevCreate*( )). The recommended value zero (0) specifies SCSI_DEF_SELECT_TIMEOUT (250 millisec). The maximum timeout possible is approximately 2 seconds. Values over the maximum specify the maximum. See the Western Digital documentation for more information.

scsiPriority

- the priority to which a task is set when performing a SCSI transaction. Legal priorities are 0 to 255. Alternately, a -1 indicates that the priority should not be altered during SCSI transactions.

RETURNS

OK, or ERROR if parameters are out of range.

SEE ALSO

wd33c93Lib

sbicShow( )

NAME

sbicShow( ) - display values of all readable wd33c93 chip registers

SYNOPSIS

```
STATUS sbicShow (pScsiCtrl)
    SCSI_CTRL  *pScsiCtrl;  /* ptr to SCSI controller info */
```

DESCRIPTION

This routine displays the state of the SBIC registers in a user-friendly way. It is only used during debugging.

RETURNS

OK, or ERROR if the call fails.

SEE ALSO

wd33c93Lib

Tools

CONTENTS

compress	- general purpose file compression utility	3-1
extract	- conditionally extract files from an archive	3-3
jump	- show jumpering of target and associated boards	3-4
makeStatTbl	- make a table of status values	3-5
makeSymTbl	- make a table of symbols	3-6
makeVersion	- create the Vx960 version module	3-7
picLib	- jump program support routines	3-8
vwbug	- submit a Technical Support Request (TSR) to Intel	3-9
vwman	- print Vx960 manual entries	3-10
vxencrypt	- encryption program for **loginLib**	3-11
xsym	- extract the symbol table from an object module	3-12

compress

NAME

compress - general purpose file compression utility

SYNOPSIS

compress [-dfFqc] [-b *bits*] [*file* ...]

DESCRIPTION

This tool reduces the size of a file using the modified Lempel-Ziv method.

INPUT

file ...	- the files to be compressed. If none are specified, input is taken from standard input.

OUTPUT

file.Z	- the compressed form of the file(s) with the same mode, owner, and times; or standard output if the input was standard input.

OPTIONS

-d	- decompress instead.
-c	- write output on standard output; do not remove original.
-b *bits*	- limit the maximum number of bits/code to *bits*.
-f	- force the output file to be generated, even if one already exists. If **-f** is not used, the user will be prompted if standard input is a tty; otherwise, the output file will not be overwritten.
-F	- forces the output file to be generated, even if no space is saved by compressing.
-q	- generate no output, unless there is an error.

ASSUMPTIONS

When a filename is given, it is replaced with the compressed version (.Z suffix) only if the file decreased in size.

ALGORITHM

The algorithm used is a modified Lempel-Ziv method (LZW) which finds common substrings and replaces them with a variable size code. This is deterministic, and can be done on the fly. Thus, the decompression procedure needs no input table, but tracks the way the table was built. The algorithm is from "A Technique for High Performance Data Compression," Terry A. Welch, IEEE Computer, Vol. 17, No. 6 (June 1984), pp. 8-19.

AUTHOR

The original **compress** program was written by: Spencer W. Thomas, Jim McKie, Steve Davies, Ken Turkowski, James A. Woods, Joe Orost.

extract

NAME

extract - conditionally extract files from an archive

SYNOPSIS

extract [-v] *archive*

DESCRIPTION

This tool extracts any files from a specified archive that are not already present in the current working directory. File modification times are preserved as stored in the archive.

OPTIONS

-v - verbose; give a file-by-file description of the extraction of archive files.

jump

NAME

jump - show jumpering of target and associated boards

SYNOPSIS

NO CALLABLE ROUTINES

SYNOPSIS

jump [-a] *targetDir*

DESCRIPTION

This tool is a utility to aid users in jumpering VME boards. The pictures generated are derived from the parameters in the **config.h** file of the specified target directory.

OPTIONS

-a - output information about all boards for the specified target.

EXAMPLES

The following will prompt the user for boards:

```
jump ../config/hkv960
```

The following will output all boards to the line printer:

```
jump -a ../config/hkv960 | lpr
```

WARNING

Not all target CPUs have 100% reliable pictures. Verify with board's hardware manual.

FILES

vw/bin/picLib.o	- pictures for network boards
vw/config/*cpu*/pic_cpu.c	- target CPU's picture program
vw/config/*cpu*/pic_board.c	- other board's picture program

makeStatTbl

NAME

makeStatTbl - make a table of status values

SYNOPSIS

makeStatTbl *hdir* [...]

DESCRIPTION

This tool creates an array of type SYMBOL which contains the names and values of all the status codes defined in the .h files in the specified directory(s). All status codes must be prefixed with "S_" to be included in this table, with the exception of status codes in the UNIX-compatible header file **errno.h**. In each *hdir* there must be a ***ModNum.h** file which defines the module numbers, e.g., "M_". The generated code is written to standard output.

The symbol array is accessible through the global variable *statTbl*. The array contains *statTblSize* elements. These are the only external declarations in the generated code.

This tool's primary use is for creating the Vx960 status table used by *printErrno*( ), but may be used by applications as well. For an example, see **vw/config/all/statTbl.c**, which is generated by this tool from **vw/h/***.

FILES

hdir/***ModNum.h**	- module number file for each h directory
symLib.h	- symbol header file

SEE ALSO

errnoLib, symLib

makeSymTbl

NAME

makeSymTbl - make a table of symbols

SYNOPSIS

makeSymTbl *objMod*

DESCRIPTION

This tool creates the C code for a symbol table structure containing the names, addresses, and types of all global symbols in the specified object module; the generated code is written to standard output. ***usrRoot*( )** in **usrConfig.c** inserts these symbols in the *standAloneSymTbl* using ***symAddSymbol*( )**.

This tool is used only when creating a stand-alone system. Normally, it is not needed, since the symbol table is constructed by reading and interpreting the symbol table contained in the system load module (*b.out* or format), either from the local boot disk or from a host system over the network.

The generated symbol table is an array of type SYMBOL accessible through the global variable *standTbl*. The array contains *standTblSize* elements. These are the only external declarations in the generated code.

For an example, see the file **vw/config/*cpu*/symTbl.c**, which is generated by this tool for **vxWorks.st** in the same directory.

FILES

symLib.h	- symbol table header file
b_out.h	- UNIX GNU/960 object module header file

SEE ALSO

xsym

makeVersion

NAME

makeVersion - create the Vx960 version module

SYNOPSIS

makeVersion

DESCRIPTION

This tool creates the source code for a version.c module, which gets linked with every Vx960 system for identification. This module contains the system version number and creation date in strings. The generated code is written to standard output.

picLib

NAME

picLib - jump program support routines

DESCRIPTION

This module contains the primitives required to show board jumpering with the jump tool. The following routines can be used in ***drawBoard*( )**, defined in the target-specific **pic_cpu.c**.

```
VOID printAt (pic, x, y, string)
VOID jumperAt (pic, x, y, len, orient, type, contents)
```

Parameters:

pic - array holding the picture.

x - x coordinate, character position 2 - 78.

y - y coordinate, line position 1 - 20.

string - message to display in picture.

len - number of jumpers in the block.

orient - orientation of the jumper block:

HORIZONTAL = LSB of contents is rightmost jumper.
VERTICAL = LSB of contents is bottom jumper.

type - type of individual jumpers in the jumper block:

HORIZONTAL: 0 = removed, 1 = installed
VERTICAL: 0 = removed, 1 = installed
HORIZONTAL_3: 0 = right, 1 = left
VERTICAL_3: 0 = down, 1 = up

contents - configuration of the jumpers in the jumper block.

INCLUDE FILE

picLib.h

SEE ALSO

jump, vw/config/*target*/pic_cpu.c

vwbug

NAME

vwbug - submit a Technical Support Request (TSR) to Intel

SYNOPSIS

vwbug [-o *outfile*] [*recipients*]

DESCRIPTION

This tool creates a software error report and mails it to *uunet!ichips!vx960bug*. The user is placed in the editor with a Technical Support Request (TSR) template. The report is mailed automatically when the user exits the editor.

OPTIONS

-o *outfile*	- the report is saved in *outfile* for printing.
recipients	- additional copies of the report are mailed to the specified recipients.

CAVEAT

A mail path to *uunet* must exist if the report is to be successfully routed to Intel. The script can be edited to set the appropriate mail path.

ENVIRONMENT

vwbug uses the editor specified by the environment variables VISUAL or EDITOR if they are set, otherwise it uses **vi.**

vwman

NAME

vwman - print Vx960 manual entries

SYNOPSIS

vwman [*section*] **[-l] [-t]** *entry*

DESCRIPTION

This tool looks up and displays an entry in the on-line Vx960 reference manual. With no options, it finds and displays *entry*. If it cannot find a formatted version of the entry in the **vw/man** directory, it formats and displays the nroff source copy from **vw/mansrc**, and, if write restrictions permit, adds a formatted version to **vw/man**.

OPTIONS

-l	- List all entries which include the string *entry*.
section	- Confine the search to a particular section: 1, 2, 3, 4, or t.
-t	- Send troff-formatted entry to printer. The default troff formatter is nroff. The **vwman** script can be edited to accommodate other versions.

EXAMPLES

List all entries whose names include the string "log":

```
% vwman -l log
```

Display hkv960 target information:

```
% vwman hkv960/target
```

Display and print a troff version of the **netDrv** entry from Section 2:

```
% vwman 2 -t netDrv
```

NOTE

The files tmac.ref and tmac.angen may need to be installed in usr/lib/tmac for this to work.

FILES

vw/mansrc/man{1,2,3,4,t}	- source manual entries
vw/man/man{1,2,3,4,t}	- nroffed manual entries
vw/mansrc/include/tmac/tmac.angen	- nroff/troff macros

vxencrypt

NAME

vxencrypt - encryption program for **loginLib**

SYNOPSIS

vxencrypt

DESCRIPTION

This tool generates the encrypted equivalent of a supplied password. It prompts the user to enter a password, and then displays the encrypted version.

The encrypted password should then be supplied to Vx960 using the ***loginUserAdd*( )** routine. This is only necessary if you have enabled login security by defining INCLUDE_SECURITY in **configAll.h**. For more information, see the manual entry for **loginLib**.

This tool contains the default encryption routine used in ***loginDefaultEncrypt*( )**. If the default encryption routine is replaced in Vx960, the routine in this module should also be replaced to reflect the changes in the encryption algorithm.

SEE ALSO

loginLib, *Programmer's Guide: Shell*

xsym

NAME

xsym - extract the symbol table from an object module

SYNOPSIS

xsym < *objMod* > *symTbl*

DESCRIPTION

This tool reads an object module on standard input, and writes an object module on standard output. The output object module contains only the symbol table, with no code, but is otherwise a normal, executable object module.

This tool is used to generate the Vx960 symbol table, **vxWorks.sym**.

FILES

b_out.h - GNU/960 **bout.h** object module header file

SEE ALSO

makeSymTbl, GNU/960 *b.out* and *coff* documentation

Keyword Index

Keyword Index

change	abort character.	*tyAbortSet*( )	1-468
set	abort function.	*tyAbortFuncSet*( )	1-467
	absolute value of integer.	*abs*( )	1-1
	absolute value routine.	abs	1-1
	accept connection from socket.	*accept*( )	1-354
initialized.	activate task that has been	*taskActivate*( )	1-422
specified partition.	allocate block of memory from	*memPartAlloc*( )	1-221
system memory partition.	allocate block of memory from	*malloc*( )	1-225
	allocate space for array.	*calloc*( )	1-225
initialize backplane	anchor.	*bpInit*( )	1-127
extract files from	archive. conditionally	extract	3-3
library.	ARPA File Transfer Protocol	ftpLib	1-98
server.	ARPA File Transfer Protocol	ftpdLib	1-106
allocate space for	array.	*calloc*( )	1-225
dot notation address to	ASCII. convert network	*inet_ntoa_b*( )	1-145
dot notation address to	ASCII. convert network	*inet_ntoa*( )	1-146
values for arguments from	ASCII string. obtain	*sscanf*( )	1-87
interrupt library	assembly language routines.	intALib	1-147
miscellaneous	assembly language routines.	vxALib	1-512
floating-point support	assembly language routines.	fppALib	1-93
modify NFS UNIX	authentication parameters.	*nfsAuthUnixPrompt*( )	1-251
display NFS UNIX	authentication parameters.	*nfsAuthUnixShow*( )	1-251
get NFS UNIX	authentication parameters.	*nfsAuthUnixGet*( )	1-252
set NFS UNIX	authentication parameters.	*nfsAuthUnixSet*( )	1-252
set ID number of NFS UNIX	authentication parameters.	*nfsIdSet*( )	1-253
field. extract	backplane address from device .	*bootBpAnchorExtract*( )	1-18
initialize	backplane anchor.	*bpInit*( )	1-127
Vx960 & SunOS	backplane driver.	if_bp	1-125
show information about	backplane network.	*bpShow*( )	1-128
change	backspace character.	*tyBackspaceSet*( )	1-468

set	baud rate for SLIP interface.	*slipBaudSet*( )	1-136
create and initialize	binary semaphore.	*semBCreate*( )	1-322
and initialize version 4.x	binary semaphore. create	*semCreate*( )	1-336
initialize static	binary semaphore.	*semInit*( )	1-337
	binary semaphore library.	semBLib	1-320
version 4.x	binary semaphore library.	semOLib	1-336
	bind name to socket.	*bind*( )	1-352
port.	bind socket to privileged IP	*bindresvport*( )	1-269
interpret boot parameters from	boot line.	*bootStringToStruct*( )	1-15
construct	boot line.	*bootStructToString*( )	1-15
change	boot line.	*bootChange*( )	1-484
prompt for	boot line parameters.	*bootParamsPrompt*( )	1-16
display	boot line parameters.	*bootParamsShow*( )	1-16
line. interpret	boot parameters from boot	*bootStringToStruct*( )	1-15
	boot ROM subroutine library.	bootLib	1-13
configuration module for	boot ROMs. system	bootConfig	1-10
and transfer control to	boot ROMs. /network devices	*reboot*( )	1-262
set data	breakpoint.	*db*( )	1-27
delete	breakpoint.	*bd*( )	1-28
continue from	breakpoint.	*c*( )	1-29
set or display	breakpoints.	*b*( )	1-26
delete all	breakpoints.	*bdall*( )	1-28
interface. set	broadcast address for network	*ifBroadcastSet*( )	1-117
interface. get	broadcast address for network	*ifBroadcastGet*( )	1-118
create empty ring	buffer.	*rngCreate*( )	1-274
delete ring	buffer.	*rngDelete*( )	1-275
get characters from ring	buffer.	*rngBufGet*( )	1-276
put bytes into ring	buffer.	*rngBufPut*( )	1-276
number of free bytes in ring	buffer. determine	*rngFreeBytes*( )	1-278
number of bytes in ring	buffer. determine	*rngNBytes*( )	1-278
invert order of bytes in	buffer.	*binvert*( )	1-4
zero out	buffer.	*bzero*( )	1-5
formatted string in specified	buffer. put	*sprintf*( )	1-84
argument list in specified	buffer. /string with variable	*vsprintf*( )	1-85
read	buffer.	*fioRead*( )	1-89
make ring	buffer empty.	*rngFlush*( )	1-275
test if ring	buffer is empty.	*rngIsEmpty*( )	1-277
test if ring	buffer is full (no more room).	*rngIsFull*( )	1-277
	buffer manipulation library.	bLib	1-2
ring	buffer subroutine library.	rngLib	1-273
compare one	buffer to another.	*bcmp*( )	1-3
copy one	buffer to another.	*bcopy*( )	1-6
a time. copy one	buffer to another one byte at	*bcopyBytes*( )	1-6
word at a time. copy one	buffer to another one long	*bcopyLongs*( )	1-7

a time. copy one	buffer to another one word at	*bcopyWords*( )	
specify	buffer to be used on stream.	*setbuf*( )	1-383
set	buffer to be used on stream.	*setbuffer*( )	1-383
character. fill	buffer with specified	*bfill*( )	1-8
character one byte at/ fill	buffer with specified	*bfillBytes*( )	1-9
put byte ahead in ring	buffer without moving ring/	*rngPutAhead*( )	1-279
perform	buffered read.	*fread*( )	1-377
perform	buffered write.	*fwrite*( )	1-379
stderr. set line	buffering for either stdout or	*setlinebuf*( )	1-384
swap	buffers.	*bswap*( )	1-4
empty stream	buffers and close file.	*fclose*( )	1-372
write out any	buffers on output stream.	*fflush*( )	1-374
pulse reset signal on SCSI	bus.	*scsiBusReset*( )	1-307
probe address for	bus error.	*vxMemProbe*( )	1-513
change abort	character.	*tyAbortSet*( )	1-468
change backspace	character.	*tyBackspaceSet*( )	1-468
change line-delete	character.	*tyDeleteLineSet*( )	1-469
change end-of-file	character.	*tyEOFSet*( )	1-469
change trap-to-monitor	character.	*tyMonitorTrapSet*( )	1-470
fill buffer with specified	character.	*bfill*( )	1-8
stream. push	character back into input	*ungetc*( )	1-385
conversion macros.	character classification and	ctypeLib	1-21
return next	character in input stream.	*fgetc*( )	1-373
return next	character in input stream.	*getc*( )	1-388
return next	character in input stream.	*getc*( )	1-388
stream. return next	character in standard in	*getchar*( )	1-387
fill buffer with specified	character one byte at a/	*bfillBytes*( )	1-9
append	character to output stream.	*fputc*( )	1-376
append	character to output stream.	*putc*( )	1-389
stream. append	character to standard out	*putchar*( )	1-388
read and convert	characters from input stream.	*fscanf*( )	1-382
get	characters from ring buffer.	*rngBufGet*( )	1-276
stream. read and convert	characters from standard in	*scanf*( )	1-382
user-defined system	clock interrupt routine.	*usrClock*( )	1-476
	clock tick support library.	tickLib	1-446
announce	clock tick to kernel.	*tickAnnounce*( )	1-446
	close directory.	*closedir*( )	1-38
	close file.	*close*( )	1-163
empty stream buffers and	close file.	*fclose*( )	1-372
interface driver.	CMC ENP 10/L Ethernet	if_enp	1-130
	compare one buffer to another.	*bcmp*( )	1-3
general purpose file	compression utility.	compress	3-1
	concatenate two lists.	*lstConcat*( )	1-209
user-defined system	configuration module.	usrConfig	1-474

ROMs. system	configuration module for boot	bootConfig	1-10
initialize dosFs volume	configuration structure.	*dosFsConfigInit*( )	1-53
connected to SCSI controller.	configure all devices	*scsiAutoConfig*( )	1-304
(example).	configure SCSI peripheral	*usrScsiConfig*( )	1-476
	continue from breakpoint.	*c*( )	1-29
current subroutine.	continue until return from	*cret*( )	1-29
character classification and	conversion macros.	ctypeLib	1-21
stream. read and	convert characters from input	*fscanf*( )	1-382
standard in stream. read and	convert characters from	*scanf*( )	1-382
address to long integer.	convert dot notation Internet	*inet_addr*( )	1-140
	convert format string.	*fioFormatV*( )	1-86
number from string to/	convert Internet network	*inet_network*( )	1-145
host to network byte order..	convert long (32 bit) from	*htonl*( )	1-392
network to host byte order..	convert long (32 bit) from	*ntohl*( )	1-393
address to ASCII.	convert network dot notation	*inet_ntoa_b*( )	1-145
address to ASCII.	convert network dot notation	*inet_ntoa*( )	1-146
host to network byte order..	convert short (16 bit) from	*htons*( )	1-393
network to host byte order..	convert short (16 bit) from	*ntohs*( )	1-394
streams.	copy from / to specified	*copyStreams*( )	1-497
stdout).	copy in (or stdin) to out (or	*copy*( )	1-496
output stream.	copy NULL-terminated string to	*fputs*( )	1-376
output stream.	copy NULL-terminated string to	*puts*( )	1-380
	copy one buffer to another.	*bcopy*( )	1-6
byte at a time.	copy one buffer to another one	*bcopyBytes*( )	1-6
long word at a time.	copy one buffer to another one	*bcopyLongs*( )	1-7
word at a time.	copy one buffer to another one	*bcopyWords*( )	1-7
set value of kernel's tick	counter.	*tickSet*( )	1-447
get value of kernel's tick	counter.	*tickGet*( )	1-448
create and initialize	counting semaphore.	*semCCreate*( )	1-325
	counting semaphore library.	semCLib	1-324
cancel currently	counting watchdog.	*wdCancel*( )	1-517
to be called at every task	create. add routine	*taskCreateHookAdd*( )	1-411
symbol table.	create and add symbol to	*symAdd*( )	1-398
semaphore.	create and initialize binary	*semBCreate*( )	1-322
semaphore.	create and initialize counting	*semCCreate*( )	1-325
queue.	create and initialize message	*msgQCreate*( )	1-231
mutual-exclusion semaphore.	create and initialize	*semMCreate*( )	1-334
4.x binary semaphore.	create and initialize version	*semCreate*( )	1-336
initialize SBIC structure.	create and partially	*wd33c93CtrlCreate*( )	2-29
mb87030 SPC.	create control structure for	*mb87030CtrlCreate*( )	2-1
initialize device and	create dosFs file system.	*dosFsMkfs*( )	1-57
	create empty ring buffer.	*rngCreate*( )	1-274
	create file.	*creat*( )	1-160
	create memory device.	*memDevCreate*( )	2-6

	create memory partition.	*memPartCreate*( )	1-219
	create new line-editor ID.	*ledOpen*( )	1-184
	create pipe device.	*pipeDevCreate*( )	2-21
	create pseudo terminal.	*ptyDevCreate*( )	2-23
	create RAM disk device.	*ramDevCreate*( )	2-26
	create remote file device.	*netDevCreate*( )	2-10
delete previously added task	create routine.	*taskCreateHookDelete*( )	1-412
show list of task	create routines.	*taskCreateHookShow*( )	1-412
initialize device and	create RT-11 file system.	*rt11FsMkfs*( )	1-294
structure.	create SCSI physical device	*scsiPhysDevCreate*( )	1-303
	create symbol table.	*symTblCreate*( )	1-397
	create Vx960 version module.	makeVersion	3-7
	create watchdog timer.	*wdCreate*( )	1-516
Vx960 remote login	daemon.	*rlogind*( )	1-271
initialize telnet	daemon.	*telnetInit*( )	1-444
Vx960 telnet	daemon.	*telnetd*( )	1-445
FTP server	daemon task.	*ftpdTask*( )	1-107
set current	date.	*rt11FsDateSet*( )	1-295
set current	date.	*dosFsDateSet*( )	1-54
partition. set	debug options for memory	*memPartOptionsSet*( )	1-221
memory partition. set	debug options for system	*memOptionsSet*( )	1-224
initialize	debug package.	*dbgInit*( )	1-25
	debugging facilities.	dbgLib	1-22
display	debugging help menu.	*dbgHelp*( )	1-24
	delay task from executing.	*taskDelay*( )	1-429
to be called at every task	delete. add routine	*taskDeleteHookAdd*( )	1-414
	delete all breakpoints.	*bdall*( )	1-28
from list.	delete and return first node	*lstGet*( )	1-212
	delete breakpoint.	*bd*( )	1-28
	delete device from I/O system.	*iosDevDelete*( )	1-175
	delete file.	*remove*( )	1-160
	delete file.	*delete*( )	1-161
	delete file.	*unlink*( )	1-161
	delete host from host table..	*hostDelete*( )	1-111
	delete logging fd.	*logFdDelete*( )	1-197
	delete message queue.	*msgQDelete*( )	1-232
find and	delete node on wake-up list.	*selNodeDelete*( )	1-317
create routine.	delete previously added task	*taskCreateHookDelete*( )	1-412
delete routine.	delete previously added task	*taskDeleteHookDelete*( )	1-415
switch routine.	delete previously added task	*taskSwitchHookDelete*( )	1-413
	delete ring buffer.	*rngDelete*( )	1-275
	delete route.	*routeDelete*( )	1-282
network interface.	delete routes associated with	*ifRouteDelete*( )	1-123
delete previously added task	delete routine.	*taskDeleteHookDelete*( )	1-415

show	delete routines.	*taskDeleteHookShow*( )	1-415
structure.	delete SCSI physical device	*scsiPhysDevDelete*( )	1-303
	delete semaphore.	*semDelete*( )	1-329
	delete SLIP interface.	*slipDelete*( )	1-137
list.	delete specified node from	*lstDelete*( )	1-210
	delete symbol table.	*symTblDelete*( )	1-398
	delete task.	*taskDelete*( )	1-423
	delete task.	*td*( )	1-491
restriction.	delete task without	*taskDeleteForce*( )	1-423
table.	delete user entry from login	*loginUserDelete*( )	1-201
	delete watchdog timer.	*wdDelete*( )	1-516
initialize list	descriptor.	*lstInit*( )	1-208
initialize RT-11 device	descriptor.	*rt11FsDevInit*( )	1-292
initialize tty device	descriptor.	*tyDevInit*( )	1-466
read bytes from file or	device.	*read*( )	1-164
partition on SCSI block	device. define logical	*scsiBlkDevCreate*( )	1-306
command to SCSI	device. issue TEST_UNIT_READY	*scsiTestUnitRdy*( )	1-307
FORMAT_UNIT command to SCSI	device. issue	*scsiFormatUnit*( )	1-308
issue INQUIRY command to SCSI	device.	*scsiInquiry*( )	1-308
MODE SELECT command to SCSI	device. issue	*scsiModeSelect*( )	1-309
MODE SENSE command to SCSI	device. issue	*scsiModeSense*( )	1-309
read sectors from SCSI block	device.	*scsiRdSecs*( )	1-310
READ_CAPACITY command to SCSI	device. issue	*scsiReadCapacity*( )	1-310
write sectors to SCSI block	device.	*scsiWrtSecs*( )	1-311
underlying driver is tty	device. return whether	*isatty*( )	1-371
do task-level write for tty	device.	*tyWrite*( )	1-471
do task-level read for tty	device.	*tyRead*( )	1-472
file system on block	device. initialize	*diskInit*( )	1-498
create remote file	device.	*netDevCreate*( )	2-10
unmount NFS	device.	*nfsUnmount*( )	2-16
create pipe	device.	*pipeDevCreate*( )	2-21
create RAM disk	device.	*ramDevCreate*( )	2-26
create memory	device.	*memDevCreate*( )	2-6
system. initialize	device and create dosFs file	*dosFsMkfs*( )	1-57
system. initialize	device and create RT-11 file	*rt11FsMkfs*( )	1-294
issue REQUEST SENSE command to	device and read results.	*scsiReqSense*( )	1-311
handle	device control requests.	*tyIoctl*( )	1-471
initialize RT-11	device descriptor.	*rt11FsDevInit*( )	1-292
initialize tty	device descriptor.	*tyDevInit*( )	1-466
pseudo memory	device driver.	memDrv	2-5
extract backplane address from	device field.	*bootBpAnchorExtract*( )	1-18
raw block	device file system library.	rawFsLib	1-254
delete	device from I/O system.	*iosDevDelete*( )	1-175
find I/O	device in device list.	*iosDevFind*( )	1-176

find I/O device in	device list.	*iosDevFind*( )	1-176
create SCSI physical	device structure.	*scsiPhysDevCreate*( )	1-303
delete SCSI physical	device structure.	*scsiPhysDevDelete*( )	1-303
pointer to SCSI physical	device structure. return	*scsiPhysDevIdGet*( )	1-304
add	device to I/O system.	*iosDevAdd*( )	1-175
modify mode of raw	device volume.	*rawFsModeChange*( )	1-260
disable raw	device volume.	*rawFsVolUnmount*( )	1-261
functions. associate block	device with dosFs file system	*dosFsDevInit*( )	1-55
functions. associate block	device with raw volume	*rawFsDevInit*( )	1-258
list all system-known	devices.	*devs*( )	1-504
display mounted NFS	devices.	*nfsDevShow*( )	2-16
to boot ROMs. reset network	devices and transfer control	*reboot*( )	1-262
controller. list physical	devices attached to SCSI	*scsiShow*( )	1-305
controller. configure all	devices connected to SCSI	*scsiAutoConfig*( )	1-304
display list of	devices in system.	*iosDevShow*( )	1-176
read one entry from	directory.	*readdir*( )	1-36
reset position to start of	directory.	*rewinddir*( )	1-37
close	directory.	*closedir*( )	1-38
change default	directory.	*cd*( )	1-494
print current default	directory.	*pwd*( )	1-496
list contents of	directory.	*ls*( )	1-500
contents of RT-11, ftp, or rsh	directory. list	*lsOld*( )	1-502
make	directory.	*mkdir*( )	1-502
remove	directory.	*rmdir*( )	1-503
do long listing of	directory contents.	*ll*( )	1-501
open	directory for searching.	*opendir*( )	1-36
POSIX	directory handling library.	dirLib	1-34
specified number of/	disassemble and display	*l*( )	1-31
	disassemble i960 object code..	*dsm960Inst*( )	1-62
Vx960/80960	disassembler.	dsm960Lib	1-61
format	disk.	*diskFormat*( )	1-498
create RAM	disk device.	*ramDevCreate*( )	2-26
RAM	disk driver.	ramDrv	2-25
(optional). prepare RAM	disk driver for use	*ramDrv*( )	2-26
initialize device and create	dosFs file system.	*dosFsMkfs*( )	1-57
associate block device with	dosFs file system functions.	*dosFsDevInit*( )	1-55
prepare to use	dosFs library.	*dosFsInit*( )	1-56
modify mode of	dosFs volume.	*dosFsModeChange*( )	1-58
unmount	dosFs volume.	*dosFsVolUnmount*( )	1-60
structure. initialize	dosFs volume configuration	*dosFsConfigInit*( )	1-53
Vx960 & SunOS backplane	driver.	if_bp	1-125
ENP 10/L Ethernet interface	driver. CMC	if_enp	1-130
201/202/302 Ethernet interface	driver. Excelan EXOS	if_ex	1-131
LANCE Ethernet interface	driver.	if_ln	1-132

IP (SLIP) network interface	driver. serial line	if_sl	1-134
install I/O	driver.	*iosDrvInstall*( )	1-173
remove I/O	driver.	*iosDrvRemove*( )	1-174
install network remote file	driver.	*netDrv*( )	2-10
Network File System I/O	driver.	nfsDrv	2-12
install NFS	driver.	*nfsDrv*( )	2-14
pipe I/O	driver.	pipeDrv	2-18
initialize pipe	driver.	*pipeDrv*( )	2-20
pseudo-terminal	driver.	ptyDrv	2-22
initialize pseudo-terminal	driver.	*ptyDrv*( )	2-23
RAM disk	driver.	ramDrv	2-25
pseudo memory device	driver.	memDrv	2-5
install memory	driver.	*memDrv*( )	2-6
network remote file I/O	driver.	netDrv	2-8
prepare RAM disk	driver for use (optional).	*ramDrv*( )	2-26
return whether underlying	driver is tty device.	*isatty*( )	1-371
tty	driver support library.	tyLib	1-461
display list of system	drivers.	*iosDrvShow*( )	1-174
loginLib.	encryption program for	vxencrypt	3-11
install	encryption routine.	*loginEncryptInstall*( )	1-204
default password	encryption routine.	*loginDefaultEncrypt*( )	1-205
change	end-of-file character.	*tyEOFSet*( )	1-469
determine if	end-of-file has been read.	*feof*( )	1-386
reset error and	end-of-file indicators.	*clearerr*( )	1-385
driver. CMC	ENP 10/L Ethernet interface	if_enp	1-130
attach	enp interface to network.	*enpattach*( )	1-130
probe address for bus	error.	*vxMemProbe*( )	1-513
formatted string to standard	error. print	*printErr*( )	1-83
indicators. reset	error and end-of-file	*clearerr*( )	1-385
reading or/ determine if	error has occurred while	*ferror*( )	1-386
log formatted	error message.	*logMsg*( )	1-194
	error status library.	errnoLib	1-63
print definition of specified	error status value.	*printErrno*( )	1-509
task. get	error status value of calling	*errnoGet*( )	1-65
task. set	error status value of calling	*errnoSet*( )	1-66
specified task. get	error status value of	*errnoOfTaskGet*( )	1-65
specified task. set	error status value of	*errnoOfTaskSet*( )	1-66
Internet address. resolve	Ethernet address for given	*etherAddrResolve*( )	1-72
add routine to receive all	Ethernet input packets.	*etherInputHookAdd*( )	1-70
send packet on	Ethernet interface.	*etherOutput*( )	1-69
CMC ENP 10/L	Ethernet interface driver.	if_enp	1-130
Excelan EXOS 201/202/302	Ethernet interface driver.	if_ex	1-131
LANCE	Ethernet interface driver.	if_ln	1-132
add routine to receive all	Ethernet output packets.	*etherOutputHookAdd*( )	1-71

hooks.	Ethernet raw I/O routines and	etherLib	1-68
Ethernet interface driver.	Excelan EXOS 201/202/302	if_ex	1-131
	~~exception handling facilities.~~	~~excLib~~	1-76
architecture-dependent	exception handling: i960.	exc960Lib	1-74
initialize	exception handling package.	*excInit*( )	1-77
display	exception information.	*excInfoShow*( )	1-75
initialize	exception/interrupt vectors.	*excVecInit*( )	1-74
routine to be called with	exceptions. specify	*excHookAdd*( )	1-78
handle task-level	exceptions.	*excTask*( )	1-79
	exit task.	*exit*( )	1-422
interface driver. Excelan	EXOS 201/202/302 Ethernet	if_ex	1-131
mount all file systems	exported by specified host.	*nfsMountAll*( )	2-15
remote host. display	exported file systems of	*nfsExportShow*( )	1-250
add logging	fd.	*logFdAdd*( )	1-196
set primary logging	fd.	*logFdSet*( )	1-196
delete logging	fd.	*logFdDelete*( )	1-197
associate stream with	fd.	*fdopen*( )	1-372
formatted string to specified	fd. print	*fdprintf*( )	1-83
argument list to specified	fd. /string with variable	*vfdprintf*( )	1-85
value. validate open	fd and return driver-specific	*iosFdValue*( )	1-177
get	fd associated with stream.	*fileno*( )	1-387
in/out/error. get	fd for global standard	*ioGlobalStdGet*( )	1-169
in/out/error. set	fd for global standard	*ioGlobalStdSet*( )	1-169
in/out/error. get	fd for task standard	*ioTaskStdGet*( )	1-170
in/out/error. set	fd for task standard	*ioTaskStdSet*( )	1-170
display list of	fd names in system.	*iosFdShow*( )	1-177
pend on set of	fds.	*select*( )	1-314
default input/output/error	fds. set shell's	*shellOrigStdSet*( )	1-341
create	file.	*creat*( )	1-160
delete	file.	*remove*( )	1-160
delete	file.	*delete*( )	1-161
delete	file.	*unlink*( )	1-161
open	file.	*open*( )	1-162
close	file.	*close*( )	1-163
change name of	file.	*rename*( )	1-163
write bytes to	file.	*write*( )	1-164
empty stream buffers and close	file.	*fclose*( )	1-372
open stream on	file.	*fopen*( )	1-374
remove	file.	*rm*( )	1-503
read string from	file.	*fioRdString*( )	1-89
general purpose	file compression utility.	compress	3-1
create remote	file device.	*netDevCreate*( )	2-10
install network remote	file driver.	*netDrv*( )	2-10
substitute named	file in place of open stream.	*freopen*( )	1-377

network remote	file I/O driver.	netDrv	2-8
read bytes from	file or device.	*read*( )	1-164
set	file read/write pointer.	*lseek*( )	1-165
get	file status information.	*fstat*( )	1-38
pathname. get	file status information using	*stat*( )	1-39
device and create RT-11	file system. initialize	*rt11FsMkfs*( )	1-294
device and create dosFs	file system. initialize	*dosFsMkfs*( )	1-57
mount NFS	file system.	*nfsMount*( )	2-15
/block device with dosFs	file system functions.	*dosFsDevInit*( )	1-55
Network	File System I/O driver.	nfsDrv	2-12
Network	File System library.	nfsLib	1-248
raw block device	file system library.	rawFsLib	1-254
RT-11 media-compatible	file system library.	rt11FsLib	1-287
MS-DOS media-compatible	file system library.	dosFsLib	1-40
initialize	file system on block device.	*diskInit*( )	1-498
specified host. mount all	file systems exported by	*nfsMountAll*( )	2-15
display exported	file systems of remote host.	*nfsExportShow*( )	1-250
library. ARPA	File Transfer Protocol	ftpLib	1-98
ARPA	File Transfer Protocol server.	ftpdLib	1-106
scanning library.	floating-point formatting and	floatLib	1-91
initialize	floating-point I/O support.	*floatInit*( )	1-91
task TCB. get	floating-point registers from	*fppTaskRegsGet*( )	1-95
task. display	floating-point registers of	*fppTaskRegsShow*( )	1-95
task. set	floating-point registers of	*fppTaskRegsSet*( )	1-96
probe for presence of	floating-point support.	*fppProbe*( )	1-96
assembly language routines.	floating-point support	fppALib	1-93
library.	floating-point support	fppLib	1-94
	format disk.	*diskFormat*( )	1-498
convert	format string.	*fioFormatV*( )	1-86
log	formatted error message.	*logMsg*( )	1-194
	formatted I/O library.	fioLib	1-80
buffer. put	formatted string in specified	*sprintf*( )	1-84
fd. print	formatted string to specified	*fdprintf*( )	1-83
error. print	formatted string to standard	*printErr*( )	1-83
out. print	formatted string to standard	*printf*( )	1-81
print	formatted string to stream.	*fprintf*( )	1-375
argument list in/ put	formatted string with variable	*vsprintf*( )	1-85
argument list to/ print	formatted string with variable	*vfdprintf*( )	1-85
argument list to/ print	formatted string with variable	*vprintf*( )	1-84
library. floating-point	formatting and scanning	floatLib	1-91
initiate transfer via	FTP.	*ftpXfer*( )	1-100
send	FTP command and get reply.	*ftpCommand*( )	1-99
get	FTP command reply.	*ftpReplyGet*( )	1-102
initialize	FTP data connection.	*ftpDataConnInit*( )	1-104

get completed	FTP data connection.	*ftpDataConnGet*( )	1-105
list contents of RT-11,	ftp, or rsh directory.	*lsOld*( )	1-502
log in to remote	FTP server.	*ftpLogin*( )	1-104
	FTP server daemon task.	*ftpdTask*( )	1-107
get control connection to	FTP server on specified host.	*ftpHookup*( )	1-103
clean up and finalize	FTP server task.	*ftpdDelete*( )	1-108
initialize	FTP server task.	*ftpdInit*( )	1-108
Controller (SPC) Library.	Fujitsu mb87030 SCSI Protocol	mb87030Lib	2-1
get fd for	global standard in/out/error.	*ioGlobalStdGet*( )	1-169
set fd for	global standard in/out/error.	*ioGlobalStdSet*( )	1-169
list	global symbols.	*lkup*( )	1-505
set non-local	goto.	*setjmp*( )	1-19
perform non-local	goto.	*longjmp*( )	1-20
connect C routine to	hardware interrupt.	*intConnect*( )	1-149
display NFS	help menu.	*nfsHelp*( )	1-249
display debugging	help menu.	*dbgHelp*( )	1-24
display task monitoring	help menu.	*spyHelp*( )	1-367
display (or set) shell	history.	*shellHistory*( )	1-340
display (or set) shell	history.	*h*( )	1-510
task	hook library.	taskHookLib	1-410
event of NML. add	hook routine to be called in	*excNMIHookAdd*( )	1-75
Ethernet raw I/O routines and	hooks.	etherLib	1-68
to FTP server on specified	host. get control connection	*ftpHookup*( )	1-103
file systems of remote	host. display exported	*nfsExportShow*( )	1-250
log in to remote	host.	*rlogin*( )	1-272
systems exported by specified	host. mount all file	*nfsMountAll*( )	2-15
tables. display	host and network routing	*routeShow*( )	1-283
long (32 bit) from network to	host byte order.. convert	*ntohl*( )	1-393
short (16 bit) from network to	host byte order.. convert	*ntohs*( )	1-394
delete	host from host table..	*hostDelete*( )	1-111
address. look up	host in host table by Internet	*hostGetByAddr*( )	1-112
look up	host in host table by name..	*hostGetByName*( )	1-111
address from network and	host numbers. form Internet	*inet_makeaddr*( )	1-142
address from network and	host numbers. form Internet	*inet_makeaddr_b*( )	1-142
add host to	host table.	*hostAdd*( )	1-110
initialize network	host table.	*hostTblInit*( )	1-110
delete host from	host table..	*hostDelete*( )	1-111
display	host table.	*hostShow*( )	1-112
address. look up host in	host table by Internet	*hostGetByAddr*( )	1-112
look up host in	host table by name..	*hostGetByName*( )	1-111
	host table subroutine library.	hostLib	1-109
add	host to host table.	*hostAdd*( )	1-110
convert long (32 bit) from	host to network byte order..	*htonl*( )	1-392
convert short (16 bit) from	host to network byte order..	*htons*( )	1-393

display statistics for	ICMP.	*icmpstatShow*( )	1-243
get file status	information.	*fstat*( )	1-38
print Vx960 version	information.	*version*( )	1-492
display exception	information.	*excInfoShow*( )	1-75
network. show	information about backplane	*bpShow*( )	1-128
queue. get	information about message	*msgQInfoGet*( )	1-236
queue. show	information about message	*msgQShow*( )	1-238
get	information about task.	*taskInfoGet*( )	1-436
network related	information display routines.	netShow	1-242
print complete	information from task's TCB.	*ti*( )	1-491
get file status	information using pathname.	*stat*( )	1-39
generic ROM	initialization.	*romStart*( )	1-11
ROM	initialization module.	bootInit	1-11
user-defined system	initialization routine.	*usrInit*( )	1-474
	initialize backplane anchor.	*bpInit*( )	1-127
create and	initialize binary semaphore.	*semBCreate*( )	1-322
for mb87030 SPC.	initialize control structure	*mb87030CtrlInit*( )	2-3
create and	initialize counting semaphore.	*semCCreate*( )	1-325
	initialize debug package.	*dbgInit*( )	1-25
dosFs file system.	initialize device and create	*dosFsMkfs*( )	1-57
RT-11 file system.	initialize device and create	*rt11FsMkfs*( )	1-294
configuration structure.	initialize dosFs volume	*dosFsConfigInit*( )	1-53
package.	initialize exception handling	*excInit*( )	1-77
vectors.	initialize exception/interrupt	*excVecInit*( )	1-74
logical partition.	initialize fields in SCSI	*scsiBlkDevInit*( )	1-306
block device.	initialize file system on	*diskInit*( )	1-498
support.	initialize floating-point I/O	*floatInit*( )	1-91
connection.	initialize FTP data	*ftpDataConnInit*( )	1-104
	initialize FTP server task.	*ftpdInit*( )	1-108
	initialize I/O system.	*iosInit*( )	1-173
	initialize kernel.	*kernelInit*( )	1-179
	initialize list descriptor.	*lstInit*( )	1-208
	initialize login table.	*loginInit*( )	1-200
library.	initialize message logging	*logInit*( )	1-194
create and	initialize message queue.	*msgQCreate*( )	1-231
semaphore. create and	initialize mutual-exclusion	*semMCreate*( )	1-334
	initialize network host table.	*hostTblInit*( )	1-110
	initialize network package.	*netLibInit*( )	1-240
routines.	initialize network show	*netShowInit*( )	1-245
	initialize pipe driver.	*pipeDrv*( )	2-20
driver.	initialize pseudo-terminal	*ptyDrv*( )	2-23
facility.	initialize remote login	*rlogInit*( )	1-270
	initialize RPC package.	*rpcInit*( )	1-286
descriptor.	initialize RT-11 device	*rt11FsDevInit*( )	1-292

create and partially	initialize SBIC structure.	*wd33c93CtrlCreate*( )	2-29
	initialize *select*( ) library.	*selectInit*( )	1-314
list.	initialize *select*( ) wake-up	*selWakeupListInit*( )	1-317
	initialize signal facilities.	*sigInit*( )	1-345
	initialize SLIP interface.	*slipInit*( )	1-135
semaphore.	initialize static binary	*semInit*( )	1-337
	initialize stdioLib support.	*stdioInit*( )	1-371
library.	initialize symbol table	*symLibInit*( )	1-396
facility.	initialize task variables	*taskVarInit*( )	1-439
specified address.	initialize task with stack at	*taskInit*( )	1-420
RPC package.	initialize task's access to	*rpcTaskInit*( )	1-286
	initialize task's registers.	*taskRegsInit*( )	1-403
	initialize telnet daemon.	*telnetInit*( )	1-444
descriptor.	initialize tty device	*tyDevInit*( )	1-466
fields in SBIC structure.	initialize user-specified	*wd33c93CtrlInit*( )	2-31
semaphore. create and	initialize version 4.x binary	*semCreate*( )	1-336
set shell's default	input/output/error fds.	*shellOrigStdSet*( )	1-341
device. issue	INQUIRY command to SCSI	*scsiInquiry*( )	1-308
display specified number of	instructions. disassemble and	*l*( )	1-31
Internet address to long	integer. convert dot notation	*inet_addr*( )	1-140
absolute value of	integer.	*abs*( )	1-1
read next word (32-bit	integer) from stream.	*getw*( )	1-380
append word (32-bit	integer) to output stream.	*putw*( )	1-381
look up host in host table by	Internet address.	*hostGetByAddr*( )	1-112
address (host number) from	Internet address. get local	*inet_lnaof*( )	1-141
return network number from	Internet address.	*inet_netof*( )	1-143
extract netmask field from	Internet address.	*bootNetmaskExtract*( )	1-17
Ethernet address for given	Internet address. resolve	*etherAddrResolve*( )	1-72
and host numbers. form	Internet address from network	*inet_makeaddr*( )	1-142
and host numbers. form	Internet address from network	*inet_makeaddr_b*( )	1-142
routines.	Internet address manipulation	inetLib	1-139
interface. get	Internet address of network	*ifAddrGet*( )	1-116
point-to-point peer. get	Internet address of	*ifDstAddrGet*( )	1-119
integer. convert dot notation	Internet address to long	*inet_addr*( )	1-140
string to address. convert	Internet network number from	*inet_network*( )	1-145
/all active connections for	Internet protocol sockets.	*inetstatShow*( )	1-244
connect C routine to hardware	interrupt.	*intConnect*( )	1-149
routine. restrict	interrupt context from using	*intRestrict*( )	1-152
determine if we are in	interrupt context or task/	*intContext*( )	1-152
routine. construct	interrupt handler for C	*intHandlerCreate*( )	1-150
set	interrupt level.	*intLevelSet*( )	1-147
	interrupt level input.	*tyIRd*( )	1-473
language routines.	interrupt library assembly	intALib	1-147
set current	interrupt lock-out level.	*intLockLevelSet*( )	1-151

get current	interrupt lock-out level.	*intLockLevelGet*( )	1-152
get current	interrupt nesting depth.	*intCount*( )	1-153
user-defined system clock	interrupt routine.	*usrClock*( )	1-476
	interrupt subroutine library.	intLib	1-148
	interrupt-level output.	*tyITx*( )	1-472
lock out	interrupts.	*intLock*( )	1-153
cancel effect of	intLock.	*intUnlock*( )	1-154
find	I/O device in device list.	*iosDevFind*( )	1-176
install	I/O driver.	*iosDrvInstall*( )	1-173
remove	I/O driver.	*iosDrvRemove*( )	1-174
Network File System	I/O driver.	nfsDrv	2-12
pipe	I/O driver.	pipeDrv	2-18
network remote file	I/O driver.	netDrv	2-8
	I/O interface library.	ioLib	1-158
standard	I/O library.	stdioLib	1-368
formatted	I/O library.	fioLib	1-80
Ethernet raw	I/O routines and hooks.	etherLib	1-68
initialize floating-point	I/O support.	*floatInit*( )	1-91
	I/O system.	iosLib	1-172
initialize	I/O system.	*iosInit*( )	1-173
add device to	I/O system.	*iosDevAdd*( )	1-175
delete device from	I/O system.	*iosDevDelete*( )	1-175
	jump program support routines.	picLib	3-8
associated boards. show	jumpering of target and	jump	3-4
initialize	kernel.	*kernelInit*( )	1-179
task management routines for	kernel. architecture specific	taskArchLib	1-403
management library for Vx960	kernel. task	taskLib	1-416
announce clock tick to	kernel.	*tickAnnounce*( )	1-446
Vx960	kernel library.	kernelLib	1-178
return	kernel revision string.	*kernelVersion*( )	1-180
set value of	kernel's tick counter.	*tickSet*( )	1-447
get value of	kernel's tick counter.	*tickGet*( )	1-448
driver.	LANCE Ethernet interface	if_ln	1-132
read line with	line-editing.	*ledRead*( )	1-185
	line-editing library.	ledLib	1-182
library. doubly	linked list subroutine	lstLib	1-207
find I/O device in device	list.	*iosDevFind*( )	1-176
add node to end of	list.	*lstAdd*( )	1-209
report number of nodes in	list.	*lstCount*( )	1-210
delete specified node from	list.	*lstDelete*( )	1-210
extract sublist from	list.	*lstExtract*( )	1-211
find first node in	list.	*lstFirst*( )	1-211
and return first node from	list. delete	*lstGet*( )	1-212
find last node in	list.	*lstLast*( )	1-213

find next node in	list.	*lstNext*( )	1-213
find Nth node in	list.	*lstNth*( )	1-214
find previous node in	list.	*lstPrevious*( )	1-214
find node in	list.	*lstFind*( )	1-215
free up	list.	*lstFree*( )	1-216
node to *select*( ) wake-up	list. add wake-up	*selNodeAdd*( )	1-316
all tasks on *select*( ) wake-up	list. wake up	*selWakeupAll*( )	1-316
and delete node on wake-up	list. find	*selNodeDelete*( )	1-317
initialize *select*( ) wake-up	list.	*selWakeupListInit*( )	1-317
of nodes on *select*( ) wake-up	list. get number	*selWakeupListLen*( )	1-318
insert node in	list after specified node.	*lstInsert*( )	1-212
	list all system-known devices.	*devs*( )	1-504
	list contents of directory.	*ls*( )	1-500
or rsh directory.	list contents of RT-11, ftp,	*lsOld*( )	1-502
initialize	list descriptor.	*lstInit*( )	1-208
	list global symbols.	*lkup*( )	1-505
/string with variable argument	list in specified buffer.	*vsprintf*( )	1-85
from specified node. find	list node nStep nodes away	*lstNStep*( )	1-215
get	list of active task IDs.	*taskIdListGet*( )	1-436
display	list of devices in system.	*iosDevShow*( )	1-176
display	list of fd names in system.	*iosFdShow*( )	1-177
timed. clear	list of function calls to be	*timexClear*( )	1-451
timed. display	list of function calls to be	*timexShow*( )	1-456
display	list of system drivers.	*iosDrvShow*( )	1-174
show	list of task create routines.	*taskCreateHookShow*( )	1-412
blocked on semaphore. get	list of task IDs that are	*semInfo*( )	1-330
task. get	list of task variables of	*taskVarInfo*( )	1-443
to SCSI controller.	list physical devices attached	*scsiShow*( )	1-305
doubly linked	list subroutine library.	lstLib	1-207
near given value.	list symbols whose values are	*lkAddr*( )	1-505
/string with variable argument	list to specified fd.	*vfdprintf*( )	1-85
/string with variable argument	list to standard out.	*vprintf*( )	1-84
concatenate two	lists.	*lstConcat*( )	1-209
memory.	load object module into	*loadModule*( )	1-188
memory.	load object module into	*loadModuleAt*( )	1-188
memory.	load object module into	*ld*( )	1-499
object module	loader.	loadLib	1-187
from Internet address. get	local address (host number)	*inet_lnaof*( )	1-141
TCB.. move i960	local registers from stack to	*taskRegsStackToTcb*( )	1-408
stack.. move i960	local registers from TCB to	*taskRegsTcbToStack*( )	1-409
	lock access to shell.	*shellLock*( )	1-342
	lock out interrupts.	*intLock*( )	1-153
	log formatted error message.	*logMsg*( )	1-194
	log in to remote FTP server.	*ftpLogin*( )	1-104

	log in to remote host.	*rlogin( )*	1-272
	log out of Vx960 system.	*logout( )*	1-510
add	logging fd.	*logFdAdd( )*	1-196
set primary	logging fd.	*logFdSet( )*	1-196
delete	logging fd.	*logFdDelete( )*	1-197
message	logging library.	logLib	1-192
initialize message	logging library.	*logInit( )*	1-194
Vx960 remote	login daemon.	*rlogind( )*	1-271
initialize remote	login facility.	*rlogInit( )*	1-270
remote	login library.	rlogLib	1-270
entry. display	login prompt and validate user	*loginPrompt( )*	1-203
change	login string.	*loginStringSet( )*	1-204
initialize	login table.	*loginInit( )*	1-200
add user to	login table.	*loginUserAdd( )*	1-200
delete user entry from	login table.	*loginUserDelete( )*	1-201
display user	login table.	*loginUserShow( )*	1-202
user name and password in	login table. verify	*loginUserVerify( )*	1-202
print Vx960	logo.	*printLogo( )*	1-510
print Vx960	manual entries.	vwman	3-10
set signal	mask.	*sigsetmask( )*	1-347
get subnet	mask for network interface.	*ifMaskGet( )*	1-120
Controller (SPC)/ Fujitsu	mb87030 SCSI Protocol	mb87030Lib	2-1
create control structure for	mb87030 SPC.	*mb87030CtrlCreate( )*	2-1
control structure for	mb87030 SPC. initialize	*mb87030CtrlInit( )*	2-3
display values of all readable	mb87030 (SPC) registers.	*spcShow( )*	2-4
report	mbuf statistics.	*mbufShow( )*	1-245
load object module into	memory.	*loadModule( )*	1-188
load object module into	memory.	*loadModuleAt( )*	1-188
free block of	memory.	*free( )*	1-226
reallocate block of	memory.	*realloc( )*	1-226
free block of	memory.	*cfree( )*	1-227
modify	memory.	*m( )*	1-493
display	memory.	*d( )*	1-494
load object module into	memory.	*ld( )*	1-499
create	memory device.	*memDevCreate( )*	2-6
pseudo	memory device driver.	memDrv	2-5
install	memory driver.	*memDrv( )*	2-6
allot	memory from caller's stack.	*taskStackAllot( )*	1-407
partition. allocate block of	memory from specified	*memPartAlloc( )*	1-221
partition. allocate block of	memory from system memory	*malloc( )*	1-225
free block of	memory in specified partition.	*memPartFree( )*	1-222
reallocate block of	memory in specified partition.	*memPartRealloc( )*	1-222
Vx960	memory manager.	memLib	1-217
create	memory partition.	*memPartCreate( )*	1-219

add memory to	memory partition.	*memPartAddToPool*( )	1-220
set debug options for	memory partition.	*memPartOptionsSet*( )	1-221
add memory to system	memory partition.	*memAddToPool*( )	1-224
set debug options for system	memory partition.	*memOptionsSet*( )	1-224
block of memory from system	memory partition. allocate	*malloc*( )	1-225
largest free block in system	memory partition. find	*memFindMax*( )	1-227
statistics. show system	memory partition blocks and	*memShow*( )	1-228
add	memory to memory partition.	*memPartAddToPool*( )	1-220
partition. add	memory to system memory	*memAddToPool*( )	1-224
log formatted error	message.	*logMsg*( )	1-194
receive	message from message queue.	*msgQReceive*( )	1-234
receive	message from socket.	*recvfrom*( )	1-356
receive	message from socket.	*recvmsg*( )	1-358
	message logging library.	logLib	1-192
initialize	message logging library.	*logInit*( )	1-194
create and initialize	message queue.	*msgQCreate*( )	1-231
delete	message queue.	*msgQDelete*( )	1-232
send message to	message queue.	*msgQSend*( )	1-233
receive message from	message queue.	*msgQReceive*( )	1-234
number of messages queued to	message queue. get	*msgQNumMsgs*( )	1-235
get information about	message queue.	*msgQInfoGet*( )	1-236
show information about	message queue.	*msgQShow*( )	1-238
	message queue library.	msgQLib	1-230
send	message to message queue.	*msgQSend*( )	1-233
send	message to socket.	*sendto*( )	1-355
send	message to socket.	*sendmsg*( )	1-356
exported by specified host.	mount all file systems	*nfsMountAll*( )	2-15
	mount NFS file system.	*nfsMount*( )	2-15
display	mounted NFS devices.	*nfsDevShow*( )	2-16
create and initialize	mutual-exclusion semaphore.	*semMCreate*( )	1-334
library.	mutual-exclusion semaphore	semMLib	1-331
without restrictions. give	mutual-exclusion semaphore	*semMGiveForce*( )	1-335
address. extract	netmask field from Internet	*bootNetmaskExtract*( )	1-17
attach bp interface to	network.	*bpattach*( )	1-127
information about backplane	network. show	*bpShow*( )	1-128
attach enp interface to	network.	*enpattach*( )	1-130
attach ex interface to	network.	*exattach*( )	1-131
attach ln interface to	network.	*lnattach*( )	1-132
attach SLIP interface to	network.	*slattach*( )	1-137
notation. extract	network address in dot	*inet_netof_string*( )	1-144
form Internet address from	network and host numbers.	*inet_makeaddr*( )	1-142
form Internet address from	network and host numbers.	*inet_makeaddr_b*( )	1-142
long (32 bit) from host to	network byte order.. convert	*htonl*( )	1-392
short (16 bit) from host to	network byte order.. convert	*htons*( )	1-393

shut down	network connection.	*shutdown*( )	1-361
control to boot ROMs. reset	network devices and transfer	*reboot*( )	1-262
to ASCII. convert	network dot notation address	*inet_ntoa_b*( )	1-145
to ASCII. convert	network dot notation address	*inet_ntoa*( )	1-146
driver.	Network File System I/O	nfsDrv	2-12
	Network File System library.	nfsLib	1-248
initialize	network host table.	*hostTblInit*( )	1-110
get Internet address of	network interface.	*ifAddrGet*( )	1-116
set interface address for	network interface.	*ifAddrSet*( )	1-116
set broadcast address for	network interface.	*ifBroadcastSet*( )	1-117
get broadcast address for	network interface.	*ifBroadcastGet*( )	1-118
define subnet for	network interface.	*ifMaskSet*( )	1-119
get subnet mask for	network interface.	*ifMaskGet*( )	1-120
specify flags for	network interface.	*ifFlagSet*( )	1-121
get metric for	network interface.	*ifMetricGet*( )	1-123
delete routes associated with	network interface.	*ifRouteDelete*( )	1-123
serial line IP (SLIP)	network interface driver.	if_sl	1-134
change	network interface flags.	*ifFlagChange*( )	1-120
get	network interface flags.	*ifFlagGet*( )	1-122
specify	network interface hop count.	*ifMetricSet*( )	1-122
	network interface library.	ifLib	1-115
	network interface library.	netLib	1-240
display attached	network interfaces.	*ifShow*( )	1-242
address. return	network number from Internet	*inet_netof*( )	1-143
address. convert Internet	network number from string to	*inet_network*( )	1-145
initialize	network package.	*netLibInit*( )	1-240
display routines.	network related information	netShow	1-242
install	network remote file driver.	*netDrv*( )	2-10
driver.	network remote file I/O	netDrv	2-8
library.	network route manipulation	routeLib	1-281
print synopsis of	network routines.	*netHelp*( )	1-483
display host and	network routing tables.	*routeShow*( )	1-283
initialize	network show routines.	*netShowInit*( )	1-245
	network task entry point.	*netTask*( )	1-241
convert long (32 bit) from	network to host byte order..	*ntohl*( )	1-393
convert short (16 bit) from	network to host byte order..	*ntohs*( )	1-394
unmount	NFS device.	*nfsUnmount*( )	2-16
display mounted	NFS devices.	*nfsDevShow*( )	2-16
install	NFS driver.	*nfsDrv*( )	2-14
mount	NFS file system.	*nfsMount*( )	2-15
display	NFS help menu.	*nfsHelp*( )	1-249
parameters. modify	NFS UNIX authentication	*nfsAuthUnixPrompt*( )	1-251
parameters. display	NFS UNIX authentication	*nfsAuthUnixShow*( )	1-251
parameters. get	NFS UNIX authentication	*nfsAuthUnixGet*( )	1-252

parameters. set	NFS UNIX authentication	***nfsAuthUnixSet( )***	1-252
parameters. set ID number of	NFS UNIX authentication	***nfsIdSet( )***	1-253
node. find list node nStep	nodes away from specified	***lstNStep( )***	1-215
report number of	nodes in list.	***lstCount( )***	1-210
list. get number of	nodes on ***select( )*** wake-up	***selWakeupListLen( )***	1-318
/disassemble i960	object code..	***dsm960Inst( )***	1-62
extract symbol table from	object module.	**xsym**	3-12
load	object module into memory.	***loadModule( )***	1-188
load	object module into memory.	***loadModuleAt( )***	1-188
load	object module into memory.	***ld( )***	1-499
	object module loader.	loadLib	1-187
return current	offset in stream.	*ftell( )*	1-378
	open directory for searching.	*opendir( )*	1-36
driver-specific/ validate	open fd and return	*iosFdValue( )*	1-177
	open file.	*open( )*	1-162
	open socket.	*socket( )*	1-352
port bound to it.	open socket with privileged	*rresvport( )*	1-266
named file in place of	open stream. substitute	*freopen( )*	1-377
	open stream on file.	*fopen( )*	1-374
bit) from host to network byte	order.. convert long (32	*htonl( )*	1-392
bit) from host to network byte	order.. convert short (16	*htons( )*	1-393
bit) from network to host byte	order.. convert long (32	*ntohl( )*	1-393
bit) from network to host byte	order.. convert short (16	*ntohs( )*	1-394
invert	order of bytes in buffer.	*binvert( )*	1-4
send	packet on Ethernet interface.	*etherOutput( )*	1-69
to receive all Ethernet input	packets. add routine	*etherInputHookAdd( )*	1-70
to receive all Ethernet output	packets. add routine	*etherOutputHookAdd( )*	1-71
create memory	partition.	*memPartCreate( )*	1-219
add memory to memory	partition.	*memPartAddToPool( )*	1-220
block of memory from specified	partition. allocate	*memPartAlloc( )*	1-221
set debug options for memory	partition.	*memPartOptionsSet( )*	1-221
block of memory in specified	partition. free	*memPartFree( )*	1-222
block of memory in specified	partition. reallocate	*memPartRealloc( )*	1-222
add memory to system memory	partition.	*memAddToPool( )*	1-224
options for system memory	partition. set debug	*memOptionsSet( )*	1-224
of memory from system memory	partition. allocate block	*malloc( )*	1-225
free block in system memory	partition. find largest	*memFindMax( )*	1-227
fields in SCSI logical	partition. initialize	*scsiBlkDevInit( )*	1-306
statistics. show	partition blocks and	*memPartShow( )*	1-223
show system memory	partition blocks and/	*memShow( )*	1-228
device. define logical	partition on SCSI block	*scsiBlkDevCreate( )*	1-306
get current user name and	password.	*remCurIdGet( )*	1-267
set remote user name and	password.	*remCurIdSet( )*	1-267
set remote user name and	password.	*iam( )*	1-268

default	password encryption routine.	*loginDefaultEncrypt*( )	1-205
verify user name and	password in login table.	*loginUserVerify*( )	1-202
set current default	path.	*ioDefPathSet*( )	1-166
set current default	path.	*chdir*( )	1-167
get current default	path.	*ioDefPathGet*( )	1-167
get current default	path.	*getcwd*( )	1-168
get current default	path.	*getwd*( )	1-168
address of point-to-point	peer. get Internet	*ifDstAddrGet*( )	1-119
get name of connected	peer.	*getpeername*( )	1-360
reports. run	periodic task activity	*spyTask*( )	1-365
reports. begin	periodic task activity	*spy*( )	1-366
spawn task to call function	periodically.	*period*( )	1-485
call function	periodically.	*periodRun*( )	1-485
create	pipe device.	*pipeDevCreate*( )	2-21
initialize	pipe driver.	*pipeDrv*( )	2-20
	pipe I/O driver.	pipeDrv	2-18
from task's TCB.	print complete information	*ti*( )	1-491
directory.	print current default	*pwd*( )	1-496
error status value.	print definition of specified	*printErrno*( )	1-509
specified fd.	print formatted string to	*fdprintf*( )	1-83
standard error.	print formatted string to	*printErr*( )	1-83
standard out.	print formatted string to	*printf*( )	1-81
stream.	print formatted string to	*fprintf*( )	1-375
variable argument list to/	print formatted string with	*vfdprintf*( )	1-85
variable argument list to/	print formatted string with	*vprintf*( )	1-84
	print stack trace of task.	*tt*( )	1-32
stack usage.	print summary of each task's	*checkStack*( )	1-488
TCB.	print summary of each task's	*i*( )	1-489
routines.	print synopsis of network	*netHelp*( )	1-483
routines.	print synopsis of selected	*help*( )	1-481
from stack.	print trace of function calls	*trcStack*( )	1-458
	print Vx960 logo.	*printLogo*( )	1-510
	print Vx960 manual entries.	vwman	3-10
information.	print Vx960 version	*version*( )	1-492
change	priority of task.	*taskPrioritySet*( )	1-425
examine	priority of task.	*taskPriorityGet*( )	1-426
change shell	prompt.	*shellPromptSet*( )	1-341
entry. display login	prompt and validate user	*loginPrompt*( )	1-203
parameters.	prompt for boot line	*bootParamsPrompt*( )	1-16
	pseudo memory device driver.	memDrv	2-5
create	pseudo terminal.	*ptyDevCreate*( )	2-23
	pseudo-terminal driver.	ptyDrv	2-22
initialize	pseudo-terminal driver.	*ptyDrv*( )	2-23
create	RAM disk device.	*ramDevCreate*( )	2-26

	RAM disk driver.	ramDrv	2-25
(optional). prepare	RAM disk driver for use	*ramDrv*( )	2-26
library.	raw block device file system	rawFsLib	1-254
modify mode of	raw device volume.	*rawFsModeChange*( )	1-260
disable	raw device volume.	*rawFsVolUnmount*( )	1-261
Ethernet	raw I/O routines and hooks.	etherLib	1-68
associate block device with	raw volume functions.	*rawFsDevInit*( )	1-258
prepare to use	raw volume library.	*rawFsInit*( )	1-259
perform buffered	read.	*fread*( )	1-377
if end-of-file has been	read. determine	*feof*( )	1-386
from input stream.	read and convert characters	*fscanf*( )	1-382
from standard in stream.	read and convert characters	*scanf*( )	1-382
	read buffer.	*fioRead*( )	1-89
device.	read bytes from file or	*read*( )	1-164
do task-level	read for tty device.	*tyRead*( )	1-472
	read line with line-editing.	*ledRead*( )	1-185
integer) from stream.	read next word (32-bit	*getw*( )	1-380
	read one entry from directory.	*readdir*( )	1-36
SENSE command to device and	read results. issue REQUEST	*scsiReqSense*( )	1-311
device.	read sectors from SCSI block	*scsiRdSecs*( )	1-310
	read string from file.	*fioRdString*( )	1-89
	read string from input stream.	*fgets*( )	1-373
stream.	read string from standard in	*gets*( )	1-379
set file	read/write pointer.	*lseek*( )	1-165
notify rawFsLib of change in	ready status.	*rawFsReadyChange*( )	1-260
notify rt11FsLib of change in	ready status.	*rt11FsReadyChange*( )	1-296
notify dosFsLib of change in	ready status.	*dosFsReadyChange*( )	1-58
check if task is	ready to run.	*taskIsReady*( )	1-434
	reallocate block of memory.	*realloc*( )	1-226
specified partition.	reallocate block of memory in	*memPartRealloc*( )	1-222
add routine to be called at	reboot.	*rebootHookAdd*( )	1-263
	reboot support library.	rebootLib	1-262
packets. add routine to	receive all Ethernet input	*etherInputHookAdd*( )	1-70
packets. add routine to	receive all Ethernet output	*etherOutputHookAdd*( )	1-71
	receive data from socket.	*recv*( )	1-357
queue.	receive message from message	*msgQReceive*( )	1-234
	receive message from socket.	*recvfrom*( )	1-356
	receive message from socket.	*recvmsg*( )	1-358
return contents of PCW	register.	*pcw*( )	1-507
return contents of TCW	register.	*tcw*( )	1-507
return contents of ACW	register.	*acw*( )	1-508
r3-r15,/ return contents of	register pfp (also sp, rip,	*pfp*( )	1-506
initialize task's	registers.	*taskRegsInit*( )	1-403
display contents of task's	registers.	*taskRegsShow*( )	1-405

set task's	registers.	*taskRegsSet*( )	1-431
modify	registers.	*mRegs*( )	1-508
of all readable wd33c93 chip	registers. display values	*sbicShow*( )	2-32
of all readable mb87030 (SPC)	registers. display values	*spcShow*( )	2-4
move i960 local	registers from stack to TCB.	*taskRegsStackToTcb*( )	1-408
get floating-point	registers from task TCB.	*fppTaskRegsGet*( )	1-95
get task's	registers from TCB.	*taskRegsGet*( )	1-431
move i960 local	registers from TCB to stack.	*taskRegsTcbToStack*( )	1-409
display floating-point	registers of task.	*fppTaskRegsShow*( )	1-95
set floating-point	registers of task.	*fppTaskRegsSet*( )	1-96
	remote command library.	remLib	1-265
create	remote file device.	*netDevCreate*( )	2-10
install network	remote file driver.	*netDrv*( )	2-10
network	remote file I/O driver.	netDrv	2-8
log in to	remote FTP server.	*ftpLogin*( )	1-104
exported file systems of	remote host. display	*nfsExportShow*( )	1-250
log in to	remote host.	*rlogin*( )	1-272
display current	remote identity.	*whoami*( )	1-268
Vx960	remote login daemon.	*rlogind*( )	1-271
initialize	remote login facility.	*rlogInit*( )	1-270
	remote login library.	rlogLib	1-270
execute shell command on	remote machine.	*rcmd*( )	1-265
set	remote user name and password.	*remCurIdSet*( )	1-267
set	remote user name and password.	*iam*( )	1-268
	remove directory.	*rmdir*( )	1-503
	remove file.	*rm*( )	1-503
	remove I/O driver.	*iosDrvRemove*( )	1-174
table.	remove symbol from symbol	*symRemove*( )	1-399
task.	remove task variable from	*taskVarDelete*( )	1-441
disable task	rescheduling.	*taskLock*( )	1-426
enable task	rescheduling.	*taskUnlock*( )	1-427
indicators.	reset error and end-of-file	*clearerr*( )	1-385
transfer control to boot/	reset network devices and	*reboot*( )	1-262
directory.	reset position to start of	*rewinddir*( )	1-37
pulse	reset signal on SCSI bus.	*scsiBusReset*( )	1-307
	restart task.	*taskRestart*( )	1-425
	resume task.	*taskResume*( )	1-424
	resume task.	*tr*( )	1-490
create empty	ring buffer.	*rngCreate*( )	1-274
delete	ring buffer.	*rngDelete*( )	1-275
get characters from	ring buffer.	*rngBufGet*( )	1-276
put bytes into	ring buffer.	*rngBufPut*( )	1-276
number of free bytes in	ring buffer. determine	*rngFreeBytes*( )	1-278
determine number of bytes in	ring buffer.	*rngNBytes*( )	1-278

make	ring buffer empty.	*rngFlush*( )	1-275
test if	ring buffer is empty.	*rngIsEmpty*( )	1-277
room). test if	ring buffer is full (no more	*rngIsFull*( )	1-277
library.	ring buffer subroutine	rngLib	1-273
ring/ put byte ahead in	ring buffer without moving	*rngPutAhead*( )	1-279
advance	ring pointer by n bytes.	*rngMoveAhead*( )	1-279
in ring buffer without moving	ring pointers. put byte ahead	*rngPutAhead*( )	1-279
generic	ROM initialization.	*romStart*( )	1-11
	ROM initialization module.	bootInit	1-11
boot	ROM subroutine library.	bootLib	1-13
configuration module for boot	ROMs. system	bootConfig	1-10
and transfer control to boot	ROMs. reset network devices	*reboot*( )	1-262
	root task.	*usrRoot*( )	1-475
enable	round-robin selection.	*kernelTimeSlice*( )	1-181
add	route.	*routeAdd*( )	1-281
delete	route.	*routeDelete*( )	1-282
network	route manipulation library.	routeLib	1-281
display host and network	routing tables.	*routeShow*( )	1-283
initialize	RPC package.	*rpcInit*( )	1-286
initialize task's access to	RPC package.	*rpcTaskInit*( )	1-286
	RPC support library.	rpcLib	1-285
initialize	RT-11 device descriptor.	*rt11FsDevInit*( )	1-292
initialize device and create	RT-11 file system.	*rt11FsMkfs*( )	1-294
list contents of	RT-11, ftp, or rsh directory.	*lsOld*( )	1-502
prepare to use	RT-11 library.	*rt11FsInit*( )	1-294
system library.	RT-11 media-compatible file	rt11FsLib	1-287
modify mode of	RT-11 volume.	*rt11FsModeChange*( )	1-296
fragmented free space on	RT-11 volume. reclaim	*squeeze*( )	1-499
make calling task	safe from deletion.	*taskSafe*( )	1-428
and partially initialize	SBIC structure. create	*wd33c93CtrlCreate*( )	2-29
user-specified fields in	SBIC structure. initialize	*wd33c93CtrlInit*( )	2-31
define logical partition on	SCSI block device.	*scsiBlkDevCreate*( )	1-306
read sectors from	SCSI block device.	*scsiRdSecs*( )	1-310
write sectors to	SCSI block device.	*scsiWrtSecs*( )	1-311
pulse reset signal on	SCSI bus.	*scsiBusReset*( )	1-307
all devices connected to	SCSI controller. configure	*scsiAutoConfig*( )	1-304
physical devices attached to	SCSI controller. list	*scsiShow*( )	1-305
TEST_UNIT_READY command to	SCSI device. issue	*scsiTestUnitRdy*( )	1-307
issue FORMAT_UNIT command to	SCSI device.	*scsiFormatUnit*( )	1-308
issue INQUIRY command to	SCSI device.	*scsiInquiry*( )	1-308
issue MODE SELECT command to	SCSI device.	*scsiModeSelect*( )	1-309
issue MODE SENSE command to	SCSI device.	*scsiModeSense*( )	1-309
issue READ_CAPACITY command to	SCSI device.	*scsiReadCapacity*( )	1-310
Computer System Interface	(SCSI) library. Small	scsiLib	1-298

initialize fields in	SCSI logical partition.	*scsiBlkDevInit*( )	1-306
configure	SCSI peripheral (example).	*usrScsiConfig*( )	1-476
structure. create	SCSI physical device	*scsiPhysDevCreate*( )	1-303
structure. delete	SCSI physical device	*scsiPhysDevDelete*( )	1-303
structure. return pointer to	SCSI physical device	*scsiPhysDevIdGet*( )	1-304
Library. Fujitsu mb87030	SCSI Protocol Controller (SPC)	mb87030Lib	2-1
wake up task pended in	*select*( ).	*selWakeup*( )	1-315
issue MODE	SELECT command to SCSI device.	*scsiModeSelect*( )	1-309
UNIX BSD 4.3	select library.	selectLib	1-313
initialize	*select*( ) library.	*selectInit*( )	1-314
add wake-up node to	*select*( ) wake-up list.	*selNodeAdd*( )	1-316
wake up all tasks on	*select*( ) wake-up list.	*selWakeupAll*( )	1-316
initialize	*select*( ) wake-up list.	*selWakeupListInit*( )	1-317
get number of nodes on	*select*( ) wake-up list.	*selWakeupListLen*( )	1-318
create and initialize binary	semaphore.	*semBCreate*( )	1-322
create and initialize counting	semaphore.	*semCCreate*( )	1-325
give	semaphore.	*semGive*( )	1-328
take	semaphore.	*semTake*( )	1-328
delete	semaphore.	*semDelete*( )	1-329
unblock every task pended on	semaphore.	*semFlush*( )	1-329
task IDs that are blocked on	semaphore. get list of	*semInfo*( )	1-330
initialize mutual-exclusion	semaphore. create and	*semMCreate*( )	1-334
initialize version 4.x binary	semaphore. create and	*semCreate*( )	1-336
initialize static binary	semaphore.	*semInit*( )	1-337
available. take version 4.x	semaphore if semaphore is	*semClear*( )	1-337
take version 4.x semaphore if	semaphore is available.	*semClear*( )	1-337
binary	semaphore library.	semBLib	1-320
counting	semaphore library.	semCLib	1-324
general	semaphore library.	semLib	1-326
mutual-exclusion	semaphore library.	semMLib	1-331
version 4.x binary	semaphore library.	semOLib	1-336
give mutual-exclusion	semaphore without/[illegible]	*semMGiveForce*( )	1-335
	send data to socket.	*send*( )	1-355
reply.	send FTP command and get	*ftpCommand*( )	1-99
	send message to message queue.	*msgQSend*( )	1-233
	send message to socket.	*sendto*( )	1-355
	send message to socket.	*sendmsg*( )	1-356
interface.	send packet on Ethernet	*etherOutput*( )	1-69
	send signal to task.	*kill*( )	1-349
	send signal to task.	*sigRaise*( )	1-349
log in to remote FTP	server.	*ftpLogin*( )	1-104
ARPA File Transfer Protocol	server.	ftpdLib	1-106
FTP	server daemon task.	*ftpdTask*( )	1-107
get control connection to FTP	server on specified host.	*ftpHookup*( )	1-103

clean up and finalize FTP	server task.	*ftpdDelete*( )	1-108
initialize FTP	server task.	*ftpdInit*( )	1-108
start	shell.	*shellInit*( )	1-338
lock access to	shell.	*shellLock*( )	1-342
machine. execute	shell command on remote	*rcmd*( )	1-265
	shell entry point.	*shell*( )	1-339
	shell execution routines.	**shellLib**	1-338
display (or set)	shell history.	*shellHistory*( )	1-340
display (or set)	shell history.	*h*( )	1-510
change	shell prompt.	*shellPromptSet*( )	1-341
script. signal	shell to stop processing	*shellScriptAbort*( )	1-340
input/output/error fds. set	shell's default	*shellOrigStdSet*( )	1-341
	shut down network connection.	*shutdown*( )	1-361
sleep until occurrence of	signal.	*pause*( )	1-348
initialize	signal facilities.	*sigInit*( )	1-345
software	signal facility library.	**sigLib**	1-343
install	signal handler.	*sigvec*( )	1-346
set	signal mask.	*sigsetmask*( )	1-347
pulse reset	signal on SCSI bus.	*scsiBusReset*( )	1-307
processing script.	signal shell to stop	*shellScriptAbort*( )	1-340
install separate	signal stack.	*sigstack*( )	1-347
send	signal to task.	*kill*( )	1-349
send	signal to task.	*sigRaise*( )	1-349
add to set of blocked	signals.	*sigblock*( )	1-348
	single-step.	*s*( )	1-30
subroutine.	single-step, but step over	*so*( )	1-31
block. find	size of largest available free	*memPartFindMax*( )	1-223
signal.	sleep until occurrence of	*pause*( )	1-348
initialize	SLIP interface.	*slipInit*( )	1-135
set baud rate for	SLIP interface.	*slipBaudSet*( )	1-136
delete	SLIP interface.	*slipDelete*( )	1-137
attach	SLIP interface to network.	*slattach*( )	1-137
driver. serial line IP	(SLIP) network interface	**if_sl**	1-134
bind name to	socket.	*bind*( )	1-352
open	socket.	*socket*( )	1-352
enable connections to	socket.	*listen*( )	1-353
accept connection from	socket.	*accept*( )	1-354
initiate connection to	socket.	*connect*( )	1-354
send data to	socket.	*send*( )	1-355
send message to	socket.	*sendto*( )	1-355
receive message from	socket.	*recvfrom*( )	1-356
send message to	socket.	*sendmsg*( )	1-356
receive data from	socket.	*recv*( )	1-357
receive message from	socket.	*recvmsg*( )	1-358

UNIX BSD 4.3 compatible	socket library.	sockLib	1-351
get	socket name.	*getsockname*( )	1-359
set	socket options.	*setsockopt*( )	1-358
get	socket options.	*getsockopt*( )	1-359
bind	socket to privileged IP port.	*bindresvport*( )	1-269
bound to it. open	socket with privileged port	*rresvport*( )	1-266
	spawn task.	*taskSpawn*( )	1-419
periodically.	spawn task to call function	*period*( )	1-485
repeatedly.	spawn task to call function	*repeat*( )	1-487
parameters.	spawn task with default	*sp*( )	1-487
control structure for mb87030	SPC. create	*mb87030CtrlCreate*( )	2-1
control structure for mb87030	SPC. initialize	*mb87030CtrlInit*( )	2-3
	spy CPU activity library.	spyLib	1-362
stop	spying and reporting.	*spyStop*( )	1-365
install separate signal	stack.	*sigstack*( )	1-347
allot memory from caller's	stack.	*taskStackAllot*( )	1-407
local registers from TCB to	stack.. move i960	*taskRegsTcbToStack*( )	1-409
trace of function calls from	stack. print	*trcStack*( )	1-458
initialize task with	stack at specified address.	*taskInit*( )	1-420
move i960 local registers from	stack to TCB..	*taskRegsStackToTcb*( )	1-408
	stack trace library.	trcLib	1-458
print	stack trace of task.	*tt*( )	1-32
print summary of each task's	stack usage.	*checkStack*( )	1-488
print formatted string to	standard error.	*printErr*( )	1-83
read string from	standard in stream.	*gets*( )	1-379
and convert characters from	standard in stream. read	*scanf*( )	1-382
return next character in	standard in stream.	*getchar*( )	1-387
get fd for global	standard in/out/error.	*ioGlobalStdGet*( )	1-169
set fd for global	standard in/out/error.	*ioGlobalStdSet*( )	1-169
get fd for task	standard in/out/error.	*ioTaskStdGet*( )	1-170
set fd for task	standard in/out/error.	*ioTaskStdSet*( )	1-170
	standard I/O library.	stdioLib	1-368
print formatted string to	standard out.	*printf*( )	1-81
with variable argument list to	standard out. /string	*vprintf*( )	1-84
append character to	standard out stream.	*putchar*( )	1-388
show partition blocks and	statistics.	*memPartShow*( )	1-223
memory partition blocks and	statistics. show system	*memShow*( )	1-228
display IP	statistics.	*ipstatShow*( )	1-244
report mbuf	statistics.	*mbufShow*( )	1-245
display	statistics for ICMP.	*icmpstatShow*( )	1-243
display all	statistics for TCP protocol.	*tcpstatShow*( )	1-246
display	statistics for UDP protocol.	*udpstatShow*( )	1-246
buffering for either stdout or	stderr. set line	*setlinebuf*( )	1-384
initialize	stdioLib support.	*stdioInit*( )	1-371

copy in (or stdin) to out (or	stdout).	*copy*( )	1-496
set line buffering for either	stdout or stderr.	*setlinebuf*( )	1-384
return next character in input	stream.	*fgetc*( )	1-373
read string from input	stream.	*fgets*( )	1-373
out any buffers on output	stream. write	*fflush*( )	1-374
print formatted string to	stream.	*fprintf*( )	1-375
append character to output	stream.	*fputc*( )	1-376
string to output	stream. copy NULL-terminated	*fputs*( )	1-376
named file in place of open	stream. substitute	*freopen*( )	1-377
reposition	stream.	*fseek*( )	1-378
return current offset in	stream.	*ftell*( )	1-378
read string from standard in	stream.	*gets*( )	1-379
word (32-bit integer) from	stream. read next	*getw*( )	1-380
string to output	stream. copy NULL-terminated	*puts*( )	1-380
(32-bit integer) to output	stream. append word	*putw*( )	1-381
convert characters from input	stream. read and	*fscanf*( )	1-382
characters from standard in	stream. read and convert	*scanf*( )	1-382
specify buffer to be used on	stream.	*setbuf*( )	1-383
set buffer to be used on	stream.	*setbuffer*( )	1-383
push character back into input	stream.	*ungetc*( )	1-385
get fd associated with	stream.	*fileno*( )	1-387
next character in standard in	stream. return	*getchar*( )	1-387
return next character in input	stream.	*getc*( )	1-388
return next character in input	stream.	*getc*( )	1-388
character to standard out	stream. append	*putchar*( )	1-388
append character to output	stream.	*putc*( )	1-389
position	stream at beginning.	*rewind*( )	1-381
empty	stream buffers and close file.	*fclose*( )	1-372
open	stream on file.	*fopen*( )	1-374
associate	stream with fd.	*fdopen*( )	1-372
copy from / to specified	streams.	*copyStreams*( )	1-497
return kernel revision	string.	*kernelVersion*( )	1-180
change login	string.	*loginStringSet*( )	1-204
get task's status as	string.	*taskStatusString*( )	1-437
convert format	string.	*fioFormatV*( )	1-86
for arguments from ASCII	string. obtain values	*sscanf*( )	1-87
read	string from file.	*fioRdString*( )	1-89
read	string from input stream.	*fgets*( )	1-373
stream. read	string from standard in	*gets*( )	1-379
put formatted	string in specified buffer.	*sprintf*( )	1-84
	string subroutine library.	strLib	1-390
Internet network number from	string to address. convert	*inet_network*( )	1-145
copy NULL-terminated	string to output stream.	*fputs*( )	1-376
copy NULL-terminated	string to output stream.	*puts*( )	1-380

print formatted	string to specified fd.	*fdprintf*( )	1-83
print formatted	string to standard error.	*printErr*( )	1-83
print formatted	string to standard out.	*printf*( )	1-81
print formatted	string to stream.	*fprintf*( )	1-375
list in/ put formatted	string with variable argument	*vsprintf*( )	1-85
list to/ print formatted	string with variable argument	*vfdprintf*( )	1-85
list to/ print formatted	string with variable argument	*vprintf*( )	1-84
define	subnet for network interface.	*ifMaskSet*( )	1-119
interface. get	subnet mask for network	*ifMaskGet*( )	1-120
	suspend task.	*taskSuspend*( )	1-424
	suspend task.	*ts*( )	1-490
check if task is	suspended.	*taskIsSuspended*( )	1-435
	swap buffers.	*bswap*( )	1-4
	swap bytes.	*swab*( )	1-5
to be called at every task	switch. add routine	*taskSwitchHookAdd*( )	1-413
delete previously added task	switch routine.	*taskSwitchHookDelete*( )	1-413
show	switch routines.	*taskSwitchHookShow*( )	1-414
look up	symbol by name.	*symFindByName*( )	1-399
look up	symbol by name and type.	*symFindByNameAndType*( )	1-400
look up	symbol by value.	*symFindByValue*( )	1-400
look up	symbol by value and type.	*symFindByValueAndType*( )	1-401
remove	symbol from symbol table.	*symRemove*( )	1-399
create	symbol table.	*symTblCreate*( )	1-397
create and add symbol to	symbol table.	*symAdd*( )	1-398
delete	symbol table.	*symTblDelete*( )	1-398
remove symbol from	symbol table.	*symRemove*( )	1-399
to examine each entry in	symbol table. call routine	*symEach*( )	1-402
module. extract	symbol table from object	xsym	3-12
initialize	symbol table library.	*symLibInit*( )	1-396
library.	symbol table subroutine	symLib	1-395
create and add	symbol to symbol table.	*symAdd*( )	1-398
list global	symbols.	*lkup*( )	1-505
make table of	symbols.	makeSymTbl	3-6
given value. list	symbols whose values are near	*lkAddr*( )	1-505
host and network routing	tables. display	*routeShow*( )	1-283
	take semaphore.	*semTake*( )	1-328
semaphore is available.	take version 4.x semaphore if	*semClear*( )	1-337
FTP server daemon	task.	*ftpdTask*( )	1-107
up and finalize FTP server	task. clean	*ftpdDelete*( )	1-108
initialize FTP server	task.	*ftpdInit*( )	1-108
message-logging support	task.	*logTask*( )	1-197
print stack trace of	task.	*tt*( )	1-32
send signal to	task.	*kill*( )	1-349
send signal to	task.	*sigRaise*( )	1-349

spawn	task.	*taskSpawn*( )	1-419
exit	task.	*exit*( )	1-422
delete	task.	*taskDelete*( )	1-423
resume	task.	*taskResume*( )	1-424
suspend	task.	*taskSuspend*( )	1-424
change priority of	task.	*taskPrioritySet*( )	1-425
restart	task.	*taskRestart*( )	1-425
examine priority of	task.	*taskPriorityGet*( )	1-426
get task ID of running	task.	*taskIdSelf*( )	1-433
verify existence of	task.	*taskIdVerify*( )	1-433
get information about	task.	*taskInfoGet*( )	1-436
add task variable to	task.	*taskVarAdd*( )	1-440
remove task variable from	task.	*taskVarDelete*( )	1-441
get list of task variables of	task.	*taskVarInfo*( )	1-443
root	task.	*usrRoot*( )	1-475
resume	task.	*tr*( )	1-490
suspend	task.	*ts*( )	1-490
delete	task.	*td*( )	1-491
error status value of calling	task. get	*errnoGet*( )	1-65
status value of specified	task. get error	*errnoOfTaskGet*( )	1-65
status value of specified	task. set error	*errnoOfTaskSet*( )	1-66
error status value of calling	task. set	*errnoSet*( )	1-66
floating-point registers of	task. display	*fppTaskRegsShow*( )	1-95
floating-point registers of	task. set	*fppTaskRegsSet*( )	1-96
start collecting	task activity data.	*spyClkStart*( )	1-363
stop collecting	task activity data.	*spyClkStop*( )	1-364
display	task activity data.	*spyReport*( )	1-364
run periodic	task activity reports.	*spyTask*( )	1-365
begin periodic	task activity reports.	*spy*( )	1-366
set	task arithmetic control word.	*taskACWSet*( )	1-406
we are in interrupt context or	task context. determine if	*intContext*( )	1-152
ID. get	task control block for task	*taskTcb*( )	1-435
routine to be called at every	task create. add	*taskCreateHookAdd*( )	1-411
delete previously added	task create routine.	*taskCreateHookDelete*( )	1-412
show list of	task create routines.	*taskCreateHookShow*( )	1-412
routine to be called at every	task delete. add	*taskDeleteHookAdd*( )	1-414
delete previously added	task delete routine.	*taskDeleteHookDelete*( )	1-415
network	task entry point.	*netTask*( )	1-241
delay	task from executing.	*taskDelay*( )	1-429
	task hook library.	taskHookLib	1-410
get name associated with	task ID.	*taskName*( )	1-432
set default	task ID.	*taskIdDefault*( )	1-434
get task control block for	task ID.	*taskTcb*( )	1-435
name. look up	task ID associated with task	*taskNameToId*( )	1-432

get	task ID of running task.	***taskIdSelf***( )	1-433
get list of active	task IDs.	***taskIdListGet***( )	1-436
semaphore. get list of	task IDs that are blocked on	***semInfo***( )	1-330
check if	task is ready to run.	***taskIsReady***( )	1-434
check if	task is suspended.	***taskIsSuspended***( )	1-435
Vx960 kernel.	task management library for	**taskLib**	1-416
kernel. architecture specific	task management routines for	**taskArchLib**	1-403
display	task monitoring help menu.	***spyHelp***( )	1-367
up task ID associated with	task name. look	***taskNameToId***( )	1-432
change	task options.	***taskOptionsSet***( )	1-429
examine	task options.	***taskOptionsGet***( )	1-430
wake up	task pended in *select*( ).	***selWakeup***( )	1-315
unblock every	task pended on semaphore.	***semFlush***( )	1-329
set	task proccessor control word.	***taskPCWSet***( )	1-405
disable	task rescheduling.	***taskLock***( )	1-426
enable	task rescheduling.	***taskUnlock***( )	1-427
make calling	task safe from deletion.	***taskSafe***( )	1-428
get fd for	task standard in/out/error.	***ioTaskStdGet***( )	1-170
set fd for	task standard in/out/error.	***ioTaskStdSet***( )	1-170
routine to be called at every	task switch. add	***taskSwitchHookAdd***( )	1-413
delete previously added	task switch routine.	***taskSwitchHookDelete***( )	1-413
floating-point registers from	task TCB. get	***fppTaskRegsGet***( )	1-95
initialized. activate	task that has been	***taskActivate***( )	1-422
periodically. spawn	task to call function	***period***( )	1-485
repeatedly. spawn	task to call function	***repeat***( )	1-487
set	task trace control word.	***taskTCWSet***( )	1-406
make calling	task unsafe from deletion.	***taskUnsafe***( )	1-428
get value of	task variable.	***taskVarGet***( )	1-442
set value of	task variable.	***taskVarSet***( )	1-442
remove	task variable from task.	***taskVarDelete***( )	1-441
add	task variable to task.	***taskVarAdd***( )	1-440
initialize	task variables facility.	***taskVarInit***( )	1-439
get list of	task variables of task.	***taskVarInfo***( )	1-443
library.	task variables support	**taskVarLib**	1-438
spawn	task with default parameters.	***sp***( )	1-487
address. initialize	task with stack at specified	***taskInit***( )	1-420
delete	task without restriction.	***taskDeleteForce***( )	1-423
handle	task-level exceptions.	***excTask***( )	1-79
device. do	task-level read for tty	***tyRead***( )	1-472
device. do	task-level write for tty	***tyWrite***( )	1-471
initialize	task's access to RPC package.	***rpcTaskInit***( )	1-286
get	task's arguments.	***taskArgsGet***( )	1-404
set	task's arguments.	***taskArgsSet***( )	1-404
initialize	task's registers.	***taskRegsInit***( )	1-403

display contents of	task's registers.	*taskRegsShow*( )	1-405
set	task's registers.	*taskRegsSet*( )	1-431
get	task's registers from TCB.	*taskRegsGet*( )	1-431
print summary of each	task's stack usage.	*checkStack*( )	1-488
get	task's status as string.	*taskStatusString*( )	1-437
print summary of each	task's TCB.	*i*( )	1-489
complete information from	task's TCB. print	*ti*( )	1-491
local registers from stack to	TCB.. move i960	*taskRegsStackToTcb*( )	1-408
get task's registers from	TCB.	*taskRegsGet*( )	1-431
print summary of each task's	TCB.	*i*( )	1-489
information from task's	TCB. print complete	*ti*( )	1-491
registers from task	TCB. get floating-point	*fppTaskRegsGet*( )	1-95
move i960 local registers from	TCB to stack..	*taskRegsTcbToStack*( )	1-409
initialize	telnet daemon.	*telnetInit*( )	1-444
Vx960	telnet daemon.	*telnetd*( )	1-445
	telnet library.	telnetLib	1-444
C-callable atomic	test-and-set primitive.	*vxTas*( )	1-512
set value of kernel's	tick counter.	*tickSet*( )	1-447
get value of kernel's	tick counter.	*tickGet*( )	1-448
clock	tick support library.	tickLib	1-446
announce clock	tick to kernel.	*tickAnnounce*( )	1-446
set current	time.	*dosFsTimeSet*( )	1-59
function or group of/	time repeated executions of	*timexN*( )	1-454
function or functions.	time single execution of	*timex*( )	1-453
list of function calls to be	timed. clear	*timexClear*( )	1-451
specify functions to be	timed.	*timexFunc*( )	1-452
list of function calls to be	timed. display	*timexShow*( )	1-456
create watchdog	timer.	*wdCreate*( )	1-516
delete watchdog	timer.	*wdDelete*( )	1-516
start watchdog	timer.	*wdStart*( )	1-517
execution	timer facilities.	timexLib	1-449
display synopsis of execution	timer facilities.	*timexHelp*( )	1-452
include execution	timer library.	*timexInit*( )	1-451
watchdog	timer library.	wdLib	1-515
functions to be called after	timing. specify	*timexPost*( )	1-455
to be called prior to	timing. specify functions	*timexPre*( )	1-456
set task	trace control word.	*taskTCWSet*( )	1-406
stack	trace library.	trcLib	1-458
stack. print	trace of function calls from	*trcStack*( )	1-458
print stack	trace of task.	*tt*( )	1-32
change	trap-to-monitor character.	*tyMonitorTrapSet*( )	1-470
whether underlying driver is	tty device. return	*isatty*( )	1-371
do task-level write for	tty device.	*tyWrite*( )	1-471
do task-level read for	tty device.	*tyRead*( )	1-472

initialize	tty device descriptor.	*tyDevInit*( )	1-466
	tty driver support library.	tyLib	1-461
parameters. modify NFS	UNIX authentication	*nfsAuthUnixPrompt*( )	1-251
parameters. display NFS	UNIX authentication	*nfsAuthUnixShow*( )	1-251
parameters. get NFS	UNIX authentication	*nfsAuthUnixGet*( )	1-252
parameters. set NFS	UNIX authentication	*nfsAuthUnixSet*( )	1-252
set ID number of NFS	UNIX authentication/	*nfsIdSet*( )	1-253
library.	UNIX BSD 4.3 compatible socket	sockLib	1-351
	UNIX BSD 4.3 select library.	selectLib	1-313
	unmount dosFs volume.	*dosFsVolUnmount*( )	1-60
	unmount NFS device.	*nfsUnmount*( )	2-16
make calling task	unsafe from deletion.	*taskUnsafe*( )	1-428
login prompt and validate	user entry. display	*loginPrompt*( )	1-203
delete	user entry from login table.	*loginUserDelete*( )	1-201
library.	user interface subroutine	usrLib	1-479
display	user login table.	*loginUserShow*( )	1-202
library.	user login/password subroutine	loginLib	1-198
get current	user name and password.	*remCurIdGet*( )	1-267
set remote	user name and password.	*remCurIdSet*( )	1-267
set remote	user name and password.	*iam*( )	1-268
login table. verify	user name and password in	*loginUserVerify*( )	1-202
add	user to login table.	*loginUserAdd*( )	1-200
get value of task	variable.	*taskVarGet*( )	1-442
set value of task	variable.	*taskVarSet*( )	1-442
put formatted string with	variable argument list in/	*vsprintf*( )	1-85
print formatted string with	variable argument list to/	*vfdprintf*( )	1-85
print formatted string with	variable argument list to/	*vprintf*( )	1-84
remove task	variable from task.	*taskVarDelete*( )	1-441
add task	variable to task.	*taskVarAdd*( )	1-440
get	vector.	*intVecGet*( )	1-156
set CPU	vector.	*intVecSet*( )	1-156
get	vector base address.	*intVecBaseGet*( )	1-155
set	vector base address.	*intVecBaseSet*( )	1-155
initialize exception/interrupt	vectors.	*excVecInit*( )	1-74
	verify existence of task.	*taskIdVerify*( )	1-433
in login table.	verify user name and password	*loginUserVerify*( )	1-202
create and initialize	version 4.x binary semaphore.	*semCreate*( )	1-336
library.	version 4.x binary semaphore	semOLib	1-336
semaphore is available. take	version 4.x semaphore if	*semClear*( )	1-337
print Vx960	version information.	*version*( )	1-492
create Vx960	version module.	makeVersion	3-7
wake-up list.	wake up all tasks on *select*( )	*selWakeupAll*( )	1-316
select( ).	wake up task pended in	*selWakeup*( )	1-315
add wake-up node to *select*( )	wake-up list.	*selNodeAdd*( )	1-316

wake up all tasks on *select*( )	wake-up list.	*selWakeupAll*( )	1-316
find and delete node on	wake-up list.	*selNodeDelete*( )	1-317
initialize *select*( )	wake-up list.	*selWakeupListInit*( )	1-317
number of nodes on *select*( )	wake-up list. get	*selWakeupListLen*( )	1-318
wake-up list. add	wake-up node to *select*( )	*selNodeAdd*( )	1-316
cancel currently counting	watchdog.	*wdCancel*( )	1-517
create	watchdog timer.	*wdCreate*( )	1-516
delete	watchdog timer.	*wdDelete*( )	1-516
start	watchdog timer.	*wdStart*( )	1-517
	watchdog timer library.	wdLib	1-515
display values of all readable	wd33c93 chip registers.	*sbicShow*( )	2-32
Controller/ library for	wd33c93 SCSI-Bus Interface	wd33c93Lib	2-29
set task processor control	word.	*taskPCWSet*( )	1-405
set task arithmetic control	word.	*taskACWSet*( )	1-406
set task trace control	word.	*taskTCWSet*( )	1-406
stream. read next	word (32-bit integer) from	*getw*( )	1-380
output stream. append	word (32-bit integer) to	*putw*( )	1-381
one buffer to another one long	word at a time. copy	*bcopyLongs*( )	1-7
copy one buffer to another one	word at a time.	*bcopyWords*( )	1-7
perform buffered	write.	*fwrite*( )	1-379
	write bytes to file.	*write*( )	1-164
do task-level	write for tty device.	*tyWrite*( )	1-471
output stream.	write out any buffers on	*fflush*( )	1-374
device.	write sectors to SCSI block	*scsiWrtSecs*( )	1-311
has occurred while reading or	writing. determine if error	*ferror*( )	1-386
	zero out buffer.	*bzero*( )	1-5