HEWLETT PACKARD

HP 9000
Computers

HP-UX Reference
Volume 2

HP-UX Reference

Volume 2: Sections 2 and 3

HP 9000 Computers

HP-UX Release 9.0

HP Part No. B2355-90033
Printed in USA August 1992

Third Edition
E0892

Legal Notices

The information contained in this document is subject to change without notice.

Hewlett-Packard Company makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard Company shall not be liable for errors contained herein or direct, indirect, special, incidental, or consequential damages in connection with the furnishing, performance, or use of this material.

Warranty: A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

Hewlett-Packard Company
3000 Hanover Street
Palo Alto, CA 94304 U.S.A.

Printing History

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. However, minor changes may be made at reprint without changing the printing date. The manual part number changes when extensive changes are made.

To ensure that you receive new editions of this manual when changes occur, you may subscribe to the appropriate product support service, available through your HP sales representative.

August 1992. Third Edition. This edition is an update to the Second Edition and is valid for HP-UX Release 9.0 on all HP 9000 systems. Replaces Second Edition, HP part number B2355-90004.

June 1991. Second Edition. Update to the First Edition for HP-UX Release 8.05 on Series 700 systems. Also valid for HP-UX Release 8.0 on Series 300/400 and Series 800 systems. Replaces First Edition, HP part number B1864-90000.

January 1991. First Edition. Replaces manual part number 09000-90013. Valid for HP-UX Release 8.0 on Series 300/400, 700, and Series 800 systems. The *Networking Reference* was merged into this manual at Release 8.0.

New Features

This edition contains several new features.

Typography has been changed to conform to style used in other HP manuals as well as industry standards (conversion complete execpt for parts of Volume 3). Command names, argument names, and such appear on the printed page in exactly the same form as when they are typed in commands or applications, eliminating much confusion regarding capitalization of letters, which items are literals or otherwise, etc.

Progressive bleed tabs in each section are positioned vertically on the page edge according to the first letter in the name of the manual entry for easier access.

As part of an on-going effort to improve the quality and usability of this manual, several entries have been expanded and rewritten for better clarity and many examples have been added or expanded in many entries. Many changes are a direct result of comments, requests, and suggestions from users outside of HP.

Manual is expanded considerably to conver new functionality from Open Software Foundation and several other sources as well as newer versions of NFS Services and other software contained in previous releases.

Do You Have Comments or Suggestions?

Comments and suggestions from users about this manual are always welcome because they are an important part of our on-going process of improving the *HP-UX Reference*.

Internal HP users send electronic mail to:

```
hpuxref@fc.hp.com
```

Other users, please use the reply card provided in the manual or send a note or letter by ordinary mail to:

HP-UX Reference Comments, MS 11
Hewlett-Packard Company
3404 East Harmony Road
Fort Collins, CO 80525-9988, U.S.A.

Notes

Table of Contents for Volume 2

Table of Contents
Volume 2

Section 2: System Calls

Entry Name(Section): *name* — **Description**

intro(2): ..... introduction to system calls
accept(2): `accept()` ..... accept connection on a socket
access(2): `access()` ..... determine accessibility of a file
acct(2): `acct()` ..... enable or disable process accounting
alarm(2): `alarm()` ..... set a process's alarm clock
atexit(2): `atexit()` ..... register a function to be called at program termination
audctl(2): `audctl()` ..... start or halt auditing system; set or get audit files
audswitch(2): `audswitch()` ..... suspend or resume auditing on current process
audwrite(2): `audwrite()` ..... write audit record for self-auditing process
bind(2): `bind()` ..... bind address to a socket
brk(2): `brk()`, `sbrk()` ..... change data segment space allocation
bsdproc(2): `killpg()`, `getpgrp()`, `setpgrp()`, `sigvec()`, `signal()` ..... 4.2 BSD-compatible process control facilities
chdir(2): `chdir()` ..... change working directory
chmod(2): `chmod()`, `fchmod()` ..... change access mode of file
chown(2): `chown()`, `fchown()` ..... change owner and group of a file
chroot(2): `chroot()` ..... change root directory
close(2): `close()` ..... close a file descriptor
cnodeid(2): `cnodeid()` ..... get the cnode ID of the local machine
cnodes(2): `cnodes()` ..... get a list of active nodes in cluster
connect(2): `connect()` ..... initiate connection on a socket
creat(2): `creat()` ..... create a new file or rewrite an existing one
dup2(2): `dup2()` ..... duplicate an open file descriptor to a specific slot
dup(2): `dup()` ..... duplicate an open file descriptor
errno(2): `errno()` ..... error indicator for system calls
exec(2): `execl()`, `execv()`, `execle()`, `execve()`, `execlp()`, `execvp()` ..... execute a file
`execle()`: execute a file ..... see **exec(2)**
`execl()`: execute a file ..... see **exec(2)**
`execlp()`: execute a file ..... see **exec(2)**
`execve()`: execute a file ..... see **exec(2)**
`execv()`: execute a file ..... see **exec(2)**
`execvp()`: execute a file ..... see **exec(2)**
exit(2): `exit()`, `_exit()` ..... terminate process
`fchdir(2)`: change working directory ..... see **chdir(2)**
`fchmod()`: change access mode of file ..... see **chmod(2)**
`fchown()`: change owner and group of a file ..... see **chown(2)**
fcntl(2): `fcntl()` ..... file control
`fgetacl()`: get access control list (ACL) information ..... see **getacl(2)**
fork(2): `fork()` ..... create a new process
`fpathconf()`: get configurable pathname variables ..... see **pathconf(2)**
fsctl(2): `fsctl()` ..... file system control
`fsetacl()`: set access control list (ACL) information ..... see **setacl(2)**
`fstatfs()`: get file system statistics ..... see **statfs(2)**
`fstat()`: get file status ..... see **stat(2)**
fsync(2): `fsync()` ..... synchronize a file's in-core state with its state on disk
ftime(2): `ftime()` ..... get date and time more precisely
`ftruncate()`: truncate a file to a specified length ..... see **truncate(2)**
getaccess(2): `getaccess()` ..... get a user's effective access rights to a file
getacl(2): `getacl()`, `fgetacl()` ..... get access control list (ACL) information
getaudid(2): `getaudid()` ..... get the audit ID (`aid()`) for the current process
getaudproc(2): `getaudproc()` ..... get audit process flag for calling process
getcontext(2): `getcontext()` ..... return the process context for context dependent file search
getdirentries(2): `getdirentries()` ..... get entries from a directory in a filesystem-independent format
getdomainname(2): `getdomainname()`, `setdomainname()` ..... get/set name of current NIS domain
`getegid()`: get effective group ID ..... see **getuid(2)**
`geteuid()`: get effective user group ID ..... see **getuid(2)**

Table of Contents
Volume 2

Entry Name(Section): *name* — **Description**

getevent(2): `getevent()` ... get events and system calls currently being audited
getfh(2): `getfh()` ... return file handle for file on remote node.
`getgid()`: get real group ID ... see **getuid(2)**
getgroups(2): `getgroups()` ... get group access list
gethostname(2): `gethostname()` ... get name of current host
getitimer(2): `getitimer()`, `setitimer()` ... get/set value of interval timer
getpeername(2): `getpeername()` ... get address of connected peer
`getpgrp2`: get process group ID of specified process ... see **getpid(2)**
`getpgrp()`: 4.2 BSD-compatible process control facilities ... see **bsdproc(2)**
`getpgrp()`: get process group ID ... see **getpid(2)**
getpid(2): `getpid()`, `getpgrp()`, `getppid()`, `getpgrp2` ... get process, process group, and parent process ID
`getppid()`: get parent process ID ... see **getpid(2)**
getpriority(2): `getpriority`, `setpriority` ... get or set process priority
`getpriority`: get process priority ... see **getpriority(2)**
getrlimit(2): `getrlimit()`, `setrlimit()` ... control consumption of system resources
getsockname(2): `getsockname()` ... get socket address
getsockopt(2): `getsockopt()`, `setsockopt()` ... get or set options on sockets
gettimeofday(2): `gettimeofday()`, `settimeofday()` ... get/set date and time
getuid(2): `getuid()`, `geteuid()`, `getgid()`, `getegid()` ... get real user, effective user, real group, and effective group IDs
`gtty()`: control device ... see **stty(2)**
ioctl(2): `ioctl()` ... control device
ipcconnect(2): `ipcconnect()()` ... request connection to another process
ipccontrol(2): `ipccontrol()()` ... perform special operations on NetIPC sockets
ipccreate(2): `ipccreate()()` ... create a call socket
ipcdest(2): `ipcdest()()` ... create a destination descriptor
ipcgetnodename(2): `ipcgetnodename()` ... obtain NetIPC node name of current host
ipclookup(2): `ipclookup()()` ... obtain a destination descriptor
ipcname(2): `ipcname()()` ... associate name with call socket or destination call socket
ipcnamerase(2): `ipcnamerase()()` ... delete name associated with a call socket or destination call socket
ipcrecv(2): `ipcrecv()()` ... establish or receive data on NetIPC virtual circuit connection
ipcrecvcn(2): `ipcrecvcn()()` ... receive connection request on a call socket
ipcselect(2): `ipcselect()()` ... determine status of call socket or VC socket
ipcsend(2): `ipcsend()()` ... send data on a virtual circuit connection
ipcsetnodename(2): `ipcsetnodename()` ... set NetIPC node name of host CPU
ipcshutdown(2): `ipcshutdown()()` ... release a descriptor
kill(2): `kill()`, `raise()` ... send a signal to a process or a group of processes
`killpg()`: 4.2 BSD-compatible process control facilities ... see **bsdproc(2)**
link(2): `link()` ... link to a file
listen(2): `listen()` ... listen for connections on a socket
lockf(2): `lockf()` ... provide semaphores and record locking on files
lseek(2): `lseek()` ... move read/write file pointer; seek
`lstat()`: get file status ... see **stat(2)**
`lsync()`: update super-block ... see **sync(2)**
madvise(2): `madvise` ... advise system of process' expected paging behavior
mkdir(2): `mkdir()` ... make a directory file
mknod(2): `mknod()` ... make a directory, or a special or ordinary file
`mknod()` – make a cnode-specific special file ... see **mknod(2)**
mmap(2): `mmap` ... map object into virtual memory
mount(2): `mount()` ... mount a file system
mprotect(2): `mprotect` ... modify memory mapping access protections
msem_init(2): `msem_init` ... initialize semaphore in mapped file or anonymous memory region
msem_lock(2): `msem_lock` ... lock a semaphore
msem_remove(2): `msem_remove` ... remove semaphore in mapped file or anonymous region
msem_unlock(2): `msem_unlock` ... unlock a semaphore
msgget(2): `msgget()` ... get message queue
msgop(2): `msgsnd()`, `msgrcv()` ... message operations

Entry Name(Section): *name* — **Description**

msgrcv(): message operations ..... see **msgop(2)**
mstctl(2): **msgctl()** ..... message control operations
msync(2): **msync** ..... synchronize a mapped file
munmap(2): **munmap** ..... unmap a mapped region
nfssvc(2): **nfssvc()**, **async_daemon** ..... NFS daemons
nice(2): **nice()** ..... change priority of a process
open(2): **open()** ..... open file for reading or writing
pathconf(2): **pathconf()**, **fpathconf()** ..... get configurable pathname variables
pause(2): **pause()** ..... suspend process until signal
pipe(2): **pipe()** ..... create an interprocess channel
plock(2): **plock()** ..... lock process, text, or data in memory
poll(2): **poll** – monitor I/O conditions on multiple file descriptors
prealloc(2): **prealloc()** ..... preallocate fast disk storage
profil(2): **profil()** ..... execution time profile
ptrace(2): **ptrace()** ..... process trace
quotactl(2): **quotactl()** ..... manipulate disk quotas
raise() – send a signal to a process or a group of processes ..... see **kill(2)**
read(2): **read()**, **readv()** ..... read input
readlink(2): **readlink()** ..... read value of a symbolic link
readv(): read input ..... see **read(2)**
reboot(2): **reboot()** ..... boot the system
recv(2): **recv()**, **recvfrom()** **recvmsg()** ..... receive message from a socket
recvfrom(): receive message from a socket ..... see **recv(2)**
recvmsg(): receive message from a socket ..... see **recv(2)**
rename(2): **rename()** ..... change the name of a file
rmdir(2): **rmdir()** ..... remove a directory file
rtprio(2): **rtprio()** ..... change or read real-time priority
sbrk(): change data segment space allocation ..... see **brk(2)**
select(2): **select()** ..... synchronous I/O multiplexing
semctl(2): **semctl()** ..... semaphore control operations
semget(2): **semget()** ..... get set of semaphores
semop(2): **semop()** ..... semaphore operations
send(2): **send()**, **sendto()** ..... send message to a socket
sendmsg(): send message to a socket ..... see **send(2)**
sendto(): send message to a socket ..... see **send(2)**
setacl(2): **setacl()**, **fsetacl()** ..... set access control list (ACL) information
setaudid(2): **setaudid()** ..... set audit ID (**aid()**) for current process
setaudproc(2): **setaudproc()** ..... set or clear auditing on calling process
setevent(2): **setevent()** ..... set current events and system calls to be audited
setgid(): set group ID ..... see **setuid(2)**
setgroups(2): **setgroups()** ..... set group access list
sethostname(2): **sethostname()** ..... set name of host cpu
setitimer(): set value of interval timer ..... see **getitimer(2)**
setpgid(2): **setpgid()**, **setpgrp2** ..... set process group ID for job control
setpgrp2: set process group ID ..... see **setpgid(2)**
setpgrp(): 4.2 BSD-compatible process control facilities ..... see **bsdproc(2)**
setpgrp() – create session and set process group ID ..... see **setsid(2)**
setpriority: set process priority ..... see **getpriority(2)**
setresgid(): set real, effective, and saved group IDs ..... see **setresuid(2)**
setresuid(2): **setresuid()**, **setresgid()** ..... set real, effective, and saved user and group IDs
setrlimit() – control consumption of system resources ..... see **getrlimit(2)**
setsid(2): **setsid()**, **setpgrp()** ..... create session and set process group ID
setsockopt(): set options on sockets ..... see **getsockopt(2)**
settimeofday(): set date and time ..... see **gettimeofday(2)**
setuid(2): **setuid()**, **setgid()** ..... set user and group IDs
shmctl(2): **shmctl()** ..... shared memory control operations
shmdt(): shared memory operations ..... see **shmop(2)**

Table of Contents
Volume 2

Entry Name(Section): *name*	Description
shmget(2): `shmget()`	get shared memory segment
shmop(2): `shmat()`, `shmdt()`	shared memory operations
shutdown(2): `shutdown()`	shut down a socket
sigaction(2): `sigaction()`	examine and change signal action
sigblock(2): `sigblock()`	block signals
`sighold()`: signal management	see **sigset(2V)**
`sigignore()`: signal management	see **sigset(2V)**
signal(2): `signal()`	specify what to do upon receipt of a signal
`signal()`: 4.2 BSD-compatible process control facilities	see **bsdproc(2)**
sigpause(2): `sigpause()`	atomically release blocked signals and wait for interrupt
`sigpause()`: signal management	see **sigset(2V)**
sigpending(2): `sigpending()`	examine pending signals
sigprocmask(2): `sigprocmask()`	examine and change blocked signals
`sigrelse()`: signal management	see **sigset(2V)**
sigset(2V): `sigset()`, `sighold()`, `sigrelse()`, `sigignore()`, `sigpause()`	signal management
sigsetmask(2): `sigsetmask()`	set current signal mask
sigspace(2): `sigspace()`	assure sufficient signal stack space
sigstack(2): `sigstack()`	set and/or get signal stack context
sigsuspend(2): `sigsuspend()`	wait for a signal
`sigvec()`: 4.2 BSD-compatible process control facilities	see **bsdproc(2)**
sigvector(2): `sigvector()`	software signal facilities
socket(2): `socket()`	create an endpoint for communication
socketpair(2): `socketpair()`	create a pair of connected sockets
stat(2): `stat()`, `lstat()`, `fstat()`	get file status
statfs(2): `statfs()`, `fstatfs()`	get file system statistics
stime(2): `stime()`	set time and date
stty(2): `stty()`, `gtty()`	control device
swapon(2): `swapon()`	add a swap device for interleaved paging/swapping
symlink(2): `symlink()`	make symbolic link to a file
sync(2): `sync()`, `lsync()`	update super-block
sysconf(2): `sysconf`	get configurable system variables
time(2): `time()`	get time
times(2): `times()`	get process and child process times
truncate(2): `truncate()`, `ftruncate()`	truncate a file to a specified length
ulimit(2): `ulimit()`	get and set user limits
umask(2): `umask()`	set and get file creation mask
umount(2): `umount()`	unmount a file system
uname(2): `uname()`	get name of current HP-UX system
unlink(2): `unlink`	remove directory entry; delete file
ustat(2): `ustat()`	get file system statistics
utime(2): `utime()`	set file access and modification times
vfork(2): `vfork()`	spawn new process (use `fork()` intead)
vfsmount(2): `vfsmount()`	mount a file system
wait(2): `wait()`, `wait3()`	wait for child or traced process to stop or terminate
`wait3()`: wait for child or traced process to stop or terminate	see **wait(2)**
`waitpid()`: wait for child or traced process to stop or terminate	see **wait(2)**
write(2): `write()`, `writev()`	write on a file
`writev()`: write on a file	see **write(2)**

Section 3: Library Routines

Entry Name(Section): *name* **Description**

a64l(3C): `a64l()`, `l64a()` ..... convert between long integer and base-64 ASCII string
intro(3): `intro()` ..... introduction to subroutines and libraries
AAudioString(3X): `AAudioString()` ..... get name of audio controller (string) passed to AOpenAudio()
ABestAudioAttributes(3X): `ABestAudioAttributes()` ..... get best audio attributes for specified controller
abort(3C): `abort()` ..... generate a software abort fault
abs(3C): `abs()`, `abs()` ..... return integer absolute value
ACalculateLength(3X): `ACalculateLength()` ..... return the size in bytes of converted data
ACheckEvent(3X): `ACheckEvent()` ..... get first event found in audio event queue
ACheckMaskEvent(3X): `ACheckMaskEvent()` ..... get first event in audio event queue that matches mask
AChooseAFileAttributes(3X): `AChooseAFileAttributes()` ..... select attributes for creating new file
AChoosePlayAttributes(3X): `AChoosePlayAttributes()` ..... select attributes for playing file or stream
AChooseSourceAttributes(3X): ..... select attributes associated with existing file or stream
`aclentrystart()`: convert pattern string form to access control list (ACL) structure ..... see **strtoacl(3C)**
ACloseAudio(3X): `ACloseAudio()` ..... close connection to specific audio server
acltostr(3C): `acltostr()` ..... convert access control list (ACL) structure to string form
AConnectionNumber(3X): `AConnectionNumber()` ..... get audio server connection number
AConnectRecordStream(3X): `AConnectRecordStream()` ..... connect socket to TCP socket address
AConvertAFile(3X): `AConvertAFile()` ..... convert audio file data format
AConvertBuffer(3X): `AConvertBuffer()` ..... convert a buffer of data
`acosdf()`: trigonometric arccosine function (float, degrees) ..... see **trigd(3M)**
`acosd()`: trigonometric arccosine function (degrees) ..... see **trigd(3M)**
`acosf()`: trigonometric arccosine function (float) ..... see **trig(3M)**
`acosh()`: inverse hyperbolic cosine function ..... see **sinh(3M)**
`acos()`: trigonometric arccosine function ..... see **trig(3M)**
ACreateSBucket(3X): `ACreateSBucket()` ..... create empty sound bucket and return pointer to it
ADataFormats(3X): `ADataFormats()` ..... get list of data formats supported by audio controller
`addexportent()` – access exported file system information ..... see **exportent(3N)**
`addmntent()`: get file system descriptor file entry ..... see **getmntent(3X)**
addopt(3N): `addopt()` ..... add argument and data to NetIPC option buffer
ADestroySBucket(3X): `ADestroySBucket()` ..... destroy specified sound bucket
`ADVANCE()`: process 16-bit characters ..... see **nl_tools_16(3C)**
`advance()`: regular expression compile and match routines ..... see **regexp(3X)**
AEndConversion(3X): `AEndConversion()` ..... finish stream data conversion
AEventsQueued(3X): `AEventsQueued()` ..... get number of events in queue for specified server connection
AGetAFileAttributes(3X): `AGetAFileAttributes()` ..... get file attributes of specified file
AGetASilenceValue(3X): `AGetSilenceValue()` ..... get a silence value
AGetChannelGain(3X): `AGetChannelGain` ..... get transaction channel gain
AGetDataFormats(3X): `AGetDataFormats()` ..... get data formats for a specified file format
AGetErrorText(3X): `AGetErrorText()` ..... copy error description into specified buffer
AGetGain(3X): `AGetGain()` ..... get play volume or record gain of specified transaction
AGetSBucketData(3X): `AGetSBucketData` ..... copy audio data in sound bucket to buffer; return number of bytes
AGetSystemChannelGain(3X): `AGetSystemChannelGain()` ..... get system or monitor channel gain
AGetTransStatus(3X): `AGetTransStatus()` ..... get status of specified transaction
AGMGainRestricted(3X): `AGMGainRestricted()` ..... find out if audio controller restricts gain entries
AGrabServer(3X): `AGrabServer()` ..... acquire exclusive use of audio server
AInputChannels(3X): `AInputChannels()` ..... get list of A/D input channels on current hardware
AInputSources(3X): `AInputSources()` ..... get types of input sources existing on current hardware
almanac(3X): `almanac()` ..... return numeric date information in MPE format
ALoadAFile(3X): `ALoadAFile()` ..... copy audio file into new sound bucket with data conversion
`alphasort()` – sort a directory pointer array ..... see **scandir(3C)**
AMaskEvent(3X): `AMaskEvent()` ..... get first matching event in audio event queue
AMaxInputGain(3X): `AMaxInputGain()` ..... get maximum input gain supported by audio controller
AMaxOutputGain(3X): `AMaxOutputGain()` ..... get maximum output gain supported by audio controller
AMinInputGain(3X): `AMinInputGain()` ..... get minimum input gain supported by audio controller
AMinOutputGain(3X): `AMinOutputGain()` ..... get minimum output gain supported by audio controller

Table of Contents
Volume 2

Entry Name(Section): *name* **Description**

ANextEvent(3X):`ANextEvent()` ..... dequeue and return first event in audio event queue
ANumDataFormats(3X): `ANumDataFormats()` ..... data formats, number supported by audio controller
ANumSamplingRates(3X): `ANumSamplingRates()` number of sampling rates supported by audio controller
AOpenAudio(3X):`AOpenAudio()` ..... open connection to specified audio server
AOutputChannels(3X):`AOutputChannels()` ..... get D/A output channels existing on current hardware
AOutputDestinations(3X):`AOutputDestinations()` ..... output destinations types on current hardware
APauseAudio(3X): `APauseAudio()` ..... pause the specified audio transaction
APeekEvent(3X): `APeekEvent()` ..... return but do not dequeue first event in audio event queue
APlaySBucket(3X): `APlaySBucket()` ..... play specified sound bucket and return transaction ID
APlaySStream(3X):`APlaySStream()` ..... initiate transaction and return transaction ID and SStream structure
AProtocolRevision(3X):`AProtocolRevision()` .... get minor revision number of protocol used by audio server
AProtocolVersion(3X): `AProtocolVersion()` ..... get major version number of protocol used by audio server
APutBackEvent(3X): `APutBackEvent()` ..... push event onto head of audio event queue
APutSBucketData(3X): `APutSBucketData()` ..... copy audio data from buffer to sound bucket
AQLength(3X): `AQLength()` ..... return number of events on audio event queue
AQueryAFile(3X):`AQueryAFile()` ..... get file format of specified file
ARecordAData(3X): `ARecordAData()` ..... read audio data into sound bucket
ARecordSStream(3X):`ARecordSStream()` . initiate transaction; return transaction ID and SStreams structure
AResumeAudio(3X): `AResumeAudio()` ..... resume specified audio transaction
ASamplingRates(3X):`ASamplingRates()` ..... return list of sampling rates supported by audio controller
ASaveSBucket(3X):`ASaveSBucket()` ..... write sound bucket data into file with data conversion
`asctime()`: convert date and time to string ..... see **ctime(3C)**
ASelectInput(3X):`ASelectInput()` ..... request report of specified audio events
AServerVendor(3X): `AServerVendor()` ..... get vendor name of audio server for this connection
ASetChannelGain(3X): `ASetChannelGain()` ..... set transaction channel gain
ASetCloseDownMode(3X): .. set close-down mode to destroy or complete transactions on specified connection
ASetErrorHandler(3X): `ASetErrorHandler()` ..... replace default error handler with specified handler
ASetGain(3X):`ASetGain()` ..... set play volume or record gain of specified transaction
ASetIOErrorHandler(3X): `ASetIOErrorHandler()` replace default I/O error handler with specified handler
ASetSystemChannelGain(3X): `ASetSystemChannelGain()` ..... set system or monitor channel gain
ASetSystemPlayGain(3X):`ASetSystemPlayGain()` ..... set system play volume
ASetSystemRecordGain(3X):`ASetSystemRecordGain()` ..... set system record gain
ASetupConversion(3X): `ASetupConversion()` ..... perform setup required for stream data conversion
ASimplePlayer(3X):`ASimplePlayer()` ..... return gain matrix of basic play device
ASimpleRecorder(3X):`ASimpleRecorder()` ..... return gain matrix of basic recording device
`asindf()`: trigonometric arcsine function (float, degrees) ..... see **trigd(3M)**
`asind()`: trigonometric arcsine function (degrees) ..... see **trigd(3M)**
`asinf()`: trigonometric arcsine function (float) ..... see **trig(3M)**
asinh(3M): `asinh()`, `acosh()`, `atanh()` ..... inverse hyperbolic functions
`asin()`: trigonometric arcsine function ..... see **trig(3M)**
ASoundBitOrder(3X): `ASoundBitOrder()` ..... get bit order used for one-bit-per-sample data
ASoundByteOrder(3X): ..... get byte order of audio data accepted by audio controller for this connection
assert(3X): `assert()` ..... verify program assertion
AStopAudio(3X): `AStopAudio()` ..... stop specified audio transaction
AtAddCallback(3X):`AtAddCallback()` ..... add callback procedure for the toolkit
`atan2df()`: trigonometric arctangent-and-quadrant function (float, degrees) ..... see **trigd(3M)**
`atan2d()`: trigonometric arctangent-and-quadrant function (degrees) ..... see **trigd(3M)**
`atan2f()`: trigonometric arctangent-and-quadrant function (float) ..... see **trig(3M)**
`atan2()`: trigonometric arctangent-and-quadrant function ..... see **trig(3M)**
`atandf()`: trigonometric arctangent function (float, degrees) ..... see **trigd(3M)**
`atand()`: trigonometric arctangent function (degrees) ..... see **trigd(3M)**
`atanf()`: trigonometric arctangent function (float) ..... see **trig(3M)**
`atanh()`: inverse hyperbolic tangent function ..... see **sinh(3M)**
`atan()`: trigonometric arctangent function ..... see **trig(3M)**
AtInitialize(3X):`AtInitialize()` ..... add audio event handler for this connection
`atof()`: convert string to double-precision number ..... see **strtod(3C)**
AtRemoveCallback(3X): `AtRemoveCallback()` ..... set callback to NULL

Entry Name(Section): *name* **Description**

AuCreatePlay(3X): `AuCreatePlay()` ..... create an audio play widget
AuCreateRecord(3X): `AuCreateRecord()` ..... create an audio record widget
AuInvokePlay(3X): `AuInvokePlay()` ..... initiate a widget play operation
AuInvokeRecord(3X): `AuInvokeRecord()` ..... initiate an audio widget record operation
AUngrabServer(3X): `AUngrabServer()` ..... release server from exclusive use by this connection
AUpdateDataLength(3X): `AUpdateDataLength()` ..... update a file's header
AuPlayWidget(3X): `AuPlayWidget()` ..... audio play widget
AuRecordWidget(3X): `AuRecordWidget()` ..... audio record widget
AuSaveFile(3X): `AuSaveFile()` ..... save sound bucket data created by record widget
AVendorRelease(3X): `AVendorRelease()` ..... get vendor release number of audio server for this connection
AWriteAHeader(3X): `AWriteAHeader()` ..... write a header for an audio file
`bcmp()`: memory operations ..... see **memory(3C)**
`bcopy()`: memory operations ..... see **memory(3C)**
bessel(3M): `j0()`, `j1()`, `jn()`, `y0()`, `y1()`, `yn()` ..... Bessel functions
bindresvport(3N): `bindresvport()` ..... bind a socket to a privileged IP port
`blclose()` – terminal block-mode library interface ..... see **blmode(3C)**
`blget()` – terminal block-mode library interface ..... see **blmode(3C)**
blmode(3C): `blmode()` ..... terminal block-mode library interface
`blopen()` – terminal block-mode library interface ..... see **blmode(3C)**
`blread()` – terminal block-mode library interface ..... see **blmode(3C)**
`blset()` – terminal block-mode library interface ..... see **blmode(3C)**
bsearch(3C): `bsearch()` ..... binary search a sorted table
byteorder(3N): `htonl()`, `htons()`, `ntohl()`, `ntohs()` .. convert values between host and network byte order
`byte_status()`, `BYTE_STATUS()`: process 16-bit characters ..... see **nl_tools_16(3C)**
`bzero()`: memory operations ..... see **memory(3C)**
`cabs()` – complex absolute value function ..... see **hypot(3M)**
cachectl(3C): `cachectl()` ..... flush and/or purge the cache
calendar(3X): `calendar()` ..... return the MPE calendar date
`calloc`: main memory allocator ..... see **malloc(3C)**
`catclose()`: close NLS message catalog for reading ..... see **catopen(3C)**
catgetmsg(3C): `catgetmsg()` ..... get message from a message catalog
catgets(3C): `catgets()` ..... get a program message
catopen(3C): `catopen()`, `catclose()` ..... open or close NLS message catalog for reading
catread(3C): `catread()` ..... MPE/RTE-style message catalog support
`cbrt()`: cube root function ..... see **exp(3M)**
`cbrtf()`: cube root function (float version) ..... see **exp(3M)**
`c_colwidth()`, `C_COLWIDTH()`: process 16-bit characters ..... see **nl_tools_16(3C)**
`ceil()`: ceiling function ..... see **floor(3M)**
`cfgetispeed()`: get tty intput baud rate ..... see **cfspeed(3C)**
`cfgetospeed()`: get tty output baud rate ..... see **cfspeed(3C)**
`cfsetispeed()`: set tty intput baud rate ..... see **cfspeed(3C)**
`cfsetospeed()`: set tty output baud rate ..... see **cfspeed(3C)**
cfspeed(3C): `cfgetospeed()`, `cfsetospeed()`, `cfgetispeed()`, `cfsetispeed()` ... tty baud rate functions
`CHARADV()`: process 16-bit characters ..... see **nl_tools_16(3C)**
`CHARAT()`: process 16-bit characters ..... see **nl_tools_16(3C)**
chownacl(3C): `chownacl()` ..... change owner and/or group in access control list (ACL)
clearenv(3C): `clearenv` ..... clear the process environment
`clearerr`: stream status inquiries ..... see **ferror(3S)**
clock(3C): `clock()` ..... report CPU time used
clock(3X): `clock()` ..... return the MPE clock value
`closedir()`: directory operations ..... see **directory(3C)**
`closelog()`: control system log ..... see **syslog(3C)**
`compile()`: regular expression compile and match routines ..... see **regexp(3X)**
confstr(3C): `confstr()` ..... get string-valued configuration values
conv(3C): `toupper()`, `tolower()`, `_toupper`, `_tolower`, `toascii()` ..... translate characters
`copysign()`, `copysignf()`: copysign manipulations ..... see **ieee(3M)**
`copysignf()`, `copysign()`: copysign manipulations ..... see **ieee(3M)**

Table of Contents
Volume 2

Entry Name(Section): *name* — **Description**

cosdf(): trigonometric cosine function (float, degrees) ..... see **trigd(3M)**
cosd(): trigonometric cosine function (degrees) ..... see **trigd(3M)**
cosf(): trigonometric cosine function (float) ..... see **trig(3M)**
coshf(): hyperbolic cosine function (float version) ..... see **sinh(3M)**
cosh(): hyperbolic cosine function ..... see **sinh(3M)**
cos(): trigonometric cosine function ..... see **trig(3M)**
cpacl(3C): **cpacl()**, **fcpacl()** ..... copy access control list (ACL) to another file
crt0(3): **crt0.o**, **mcrt0.o**, **frt0.o**, **mfrt0.o** ..... execution startup routines
crt0.o: execution startup routines ..... see **crt0(3)**
crypt(3C): **crypt()**, **setkey()**, **encrypt()** ..... generate hashing encryption
ctermid(3S): **ctermid()** ..... generate file name for terminal
ctime(3C): **ctime()**, **nl_cxtime()**, **localtime()**, **gmtime()**, **asctime()**, **nl_ascxtime()**, **timezone()**, **daylight()**, **tzname()**, **tzset()**, **nl_ctime()**, **nl_asctime()** ..... convert date and time to string
ctime(): convert date and time to string ..... see **ctime(3C)**
ctype(3C): **isalpha()**, **isupper()**, **islower()**, **isdigit()**, **isxdigit()**, **isalnum()**, **isspace()**, **ispunct()**, **isprint()**, **isgraph()**, **iscntrl()**, **isascii()** ..... classify characters
currlangid(): NLS information about native languages ..... see **langinfo(3C)**
curses(3X): **curses()** ..... CRT screen handling and optimization package
cuserid(3S): **cuserid()** ..... get character login name of the user
cvtnum(3C): **cvtnum()** ..... convert string to floating point number
datalock(3C): **datalock()** ..... lock process into memory after allocating data and stack space
daylight(): convert date and time to string ..... see **ctime(3C)**
dbm(3X): **dbminit()**, **fetch()**, **store()**, **delete()**, **firstkey()**, **nextkey()**, **dbmclose()** ..... database subroutines
dbm_clearerr: database subroutines ..... see **ndbm(3X)**
dbmclose(): database subroutines ..... see **dbm(3X)**
dbm_close: database subroutines ..... see **ndbm(3X)**
dbm_delete: database subroutines ..... see **ndbm(3X)**
dbm_error: database subroutines ..... see **ndbm(3X)**
dbm_fetch: database subroutines ..... see **ndbm(3X)**
dbm_firstkey: database subroutines ..... see **ndbm(3X)**
dbminit(): database subroutines ..... see **dbm(3X)**
dbm_nextkey: database subroutines ..... see **ndbm(3X)**
dbm_open: database subroutines ..... see **ndbm(3X)**
dbm_store: database subroutines ..... see **ndbm(3X)**
delete(): database subroutines ..... see **dbm(3X)**
devnm(3): **devnm()** ..... map device ID to file path
dial(3C): **dial()**, **undial()** ..... establish an out-going terminal line connection
difftime(): difference between calendar times ..... see **ctime(3C)**
directory(3C): **opendir()**, **readdir()**, **telldir()**, **seekdir()**, **rewinddir()**, **closedir()** ..... directory operations
div(3C): **div()**, **ldiv()** ..... integer division and remainder
dn_comp, **dn_expand**, – resolver routines ..... see **resolver(3N)**
drand48(3C): **drand48()**, **erand48()**, **lrand48()**, **nrand48()**, **mrand48()**, **jrand48()**, **srand48()**, **seed48()**, **lcong48()** ..... generate uniformly distributed pseudo-random numbers
drem(): remainder manipulations ..... see **ieee(3M)**
ecvt(3C): **ecvt()**, **fcvt()**, **gcvt()**, **nl_gcvt()** ..... convert floating-point number to string
edata: last locations in program ..... see **end(3C)**
encrypt(): generate hashing encryption ..... see **crypt(3C)**
end(3C): **end**, **etext**, **edata** ..... last locations in program
endccent(): get cluster configuration entry ..... see **getccent(3C)**
endexportent() – access exported file system information ..... see **exportent(3N)**
endfsent(): get file system descriptor file entry ..... see **getfsent(3X)**
endgrent(): get group file entry ..... see **getgrent(3C)**
endhostent(): get network host entry ..... see **gethostent(3N)**
endmntent(): get file system descriptor file entry ..... see **getmntent(3X)**
endnetent(): get network entry ..... see **getnetent(3N)**

Entry Name(Section): *name* — **Description**

endprotoent(): get protocol entry ..... see **getprotoent(3N)**
endpwent(): get password file entry ..... see **getpwent(3C)**
endpwent(): get secure password file entry ..... see **getspwent(3C)**
endservent(): get service entry ..... see **getservent(3N)**
endusershell() – close legal user shells file ..... see **getusershell(3C)**
endutent(): access utmp file entry ..... see **getut(3C)**
erand48(): generate pseudo-random numbers ..... see **drand48(3C)**
erf(3M): **erf()**, **erfc()** ..... error function and complementary error function
erfc(): error function and complementary error function ..... see **erf(3M)**
errno: system error messages ..... see **perror(3C)**
error_$intro(3): **error_$intro** ..... error text database operations
error_$c_get_text(3): **error_$c_get_text()** ..... return subsystem, module, and error texts for a status code
error_$c_text(3): **error_$c_text()** ..... return an error message for a status code
etext: last locations in program ..... see **end(3C)**
exp(3M): **exp()**, **log()**, **log10()**, **log2()**, **pow()**, **sqrt()**, **cbrt()**, **expf()**, **logf()**, **log10f()**, **log2f()**, **powf()**, **sqrtf()** ... exponential, logarithm, power, square root, cube root functions
expf(): exponential function (float version) ..... see **exp(3M)**
exportent(3N): **exportent()**, **getexportent()**, **setexportent()**, **addexportent()**, **remexportent()**, **endexportent()**, **getexportopt()** ..... access exported file system information
fabs(): absolute value function ..... see **floor(3M)**
fabsf(): absolute value function (float version) ..... see **floor(3M)**
fclose(3S): **fclose()**, **fflush()** ..... close or flush a stream
fcpacl(): copy access control list (ACL) to another file ..... see **cpacl(3C)**
fcvt(): convert floating-point number to string ..... see **ecvt(3C)**
fdopen(): associate a stream with a file descriptor ..... see **fopen(3S)**
feof: stream status inquiries ..... see **ferror(3S)**
ferror(3S): **ferror**, **feof**, **clearerr**, **fileno** ..... stream status inquiries
fetch(): database subroutines ..... see **dbm(3X)**
fflush(): flush a stream ..... see **fclose(3S)**
ffs(): memory operations ..... see **memory(3C)**
fgetccent(): get cluster configuration entry ..... see **getccent(3C)**
fgetc(): get character from a stream file ..... see **getc(3S)**
fgetgrent(): get group file entry ..... see **getgrent(3C)**
fgetpos(3S): **fgetpos()**, **fsetpos()** ..... save or restore file position indicator for a stream
fgetpwent(): get password file entry ..... see **getpwent(3C)**
fgetpwent(): get secure password file entry ..... see **getspwent(3C)**
fgets(): get a string from a stream ..... see **gets(3S)**
fgetwc(): get wide character from a stream file ..... see **getwc(3C)**
fgetws(): get a wide string from a stream ..... see **getws(3C)**
fileno(3S): **fileno()** ..... map stream pointer to file descriptor
finitef(), **finite()**: floating-point classification functions ..... see **ieee(3M)**
finite(), **finitef()**: floating-point classification functions ..... see **ieee(3M)**
firstkey(): database subroutines ..... see **dbm(3X)**
firstof4(), **FIRSTof2()**: process 16-bit characters ..... see **nl_tools_16(3C)**
floor(3M): **floor()**, **ceil()**, **fmod()**, **fabs()**, **fabsf()**, **rint()** ..... floor, ceiling, remainder, absolute value functions
fmodf(): remainder function (float version) ..... see **floor(3M)**
fmod(): remainder function ..... see **floor(3M)**
fnmatch(3C): **fnmatch()** ..... match filename patterns
fopen(3S): **fopen()**, **freopen()**, **fdopen()** ..... open or re-open a stream file; convert file to stream
fpclassify(3M): **fpclassify()**, **fpclassifyf()** ..... floating-point classification functions
fpclassifyf(): floating-point classification function (float version) ..... see **fpclassify(3M)**
fpgetcontrol(), **fpsetcontrol()**: floating-point control register functions ..... see **fpgetround(3M)**
fpgetfastmode(), **fpsetfastmode()**: floating-point underflow mode functions ..... see **fpgetround(3M)**
fpgetmask(), **fpsetmask()**: floating-point exception trap enables functions ..... see **fpgetround(3M)**
fpgetround(3M): **fpgetround()**, **fpsetround()**, **fpgetmask()**, **fpsetmask()**, **fpgetsticky()**, **fpsetsticky()**, **fpgetcontrol()**, **fpsetcontrol()**, **fpgetfastmode()**,

Table of Contents
Volume 2

Entry Name(Section): *name* **Description**

`fpsetfastmode()`, `fpsetdefaults()` ..... floating-point mode control functions
`fpgetsticky()`, `fpsetsticky()`: floating-point exception flags functions ..... see **fpgetround(3M)**
`fprintf()`: print formatted output ..... see **printf(3S)**
`fprintmsg()`: print formatted output with numbered arguments ..... see **printmsg(3C)**
`fpsetcontrol()`, `fpgetcontrol()`: floating-point control register functions ..... see **fpgetround(3M)**
`fpsetdefaults()`: floating-point control register defaults functions ..... see **fpgetround(3M)**
`fpsetfastmode()`, `fpgetfastmode()`: floating-point underflow mode functions ..... see **fpgetround(3M)**
`fpsetmask()`, `fpgetmask()`: floating-point exception trap enables functions ..... see **fpgetround(3M)**
`fpsetround()`, `fpgetround()`: floating-point rounding mode functions ..... see **fpgetround(3M)**
`fpsetsticky()`, `fpgetsticky()`: floating-point exception flags functions ..... see **fpgetround(3M)**
`fputc()`: put character on a stream ..... see **putc(3S)**
`fputs()`: put a string on a stream ..... see **puts(3S)**
`fputwc()`: put wide character on a stream ..... see **putwc(3C)**
`fputws()`: put a wide string on a stream ..... see **putws(3C)**
fread(3S): `fread()`, `fwrite()` ..... buffered binary input/output to a stream file
`free`: main memory allocator ..... see **malloc(3C)**
`freopen()`: re-open a stream file; convert file to stream ..... see **fopen(3S)**
frexp(3C): `frexp`, `ldexp`, `modf` ..... split floating-point into mantissa and exponent
`frt0.o`: execution startup routines ..... see **crt0(3)**
`fscanf()`: formatted input conversion, read from stream file ..... see **scanf(3S)**
fseek(3S): `fseek`, `rewind`, `ftell` ..... reposition a file pointer in a stream
`fsetaclentry()`: add, modify, or delete access control list entry ..... see **setaclentry(3C)**
`fsetpos()` – restore file position indicator for a stream ..... see **fgetpos(3S)**
`fstatfsdev()`: get file system statistics ..... see **statfsdev(3C)**
`ftell`: reposition a file pointer in a stream ..... see **fseek(3S)**
`ftok()` – standard interprocess communication package ..... see **stdipc(3C)**
ftw(3C): `ftw`, `ftwh` ..... walk a file tree
`ftwh`: walk a file tree ..... see **ftw(3C)**
`fwrite()`: buffered binary output to a stream file ..... see **fread(3S)**
gamma(3M): `gamma()`, `lgamma()`, `signgam()` ..... log gamma function
`gcrt0.o`: execution startup routines ..... see **crt0(3)**
`gcvt()`: convert floating-point number to string ..... see **ecvt(3C)**
getc(3S): `getc()`, `getchar()`, `fgetc()`, `getw()` ..... get character or word from a stream file
`getcccid()`: get cluster configuration entry ..... see **getccent(3C)**
getccent(3C): `getccent()`, `getcccid()`, `getccnam()`, `setccent()`, `endccent()`, `fgetccent()` ..... get cluster configuration entry
`getccnam()`: get cluster configuration entry ..... see **getccent(3C)**
getcdf(3C): `getcdf()`, `hidecdf()` ..... manipulate CDF path names
`getchar()`: get character from a stream file ..... see **getc(3S)**
getclock(3C): `getclock` ..... get current value of system-wide clock
getcwd(3C): `getcwd()`, `gethcwd()` ..... get path-name of current working directory
getdate(3C): `getdate()` ..... convert user format date and time
getdiskbyname(3C) :`getdiskbyname()` ..... get disk description by its name
getenv(3C): `getenv()` ..... return value for environment name
`getexportent()` – access exported file system information ..... see **exportent(3N)**
`getexportopt()` – access exported file system information ..... see **exportent(3N)**
getfsent(3X): `getfsent()`, `getfsspec()`, `getfsfile()`, `getfstype()`, `setfsent()`, `endfsent()` ..... get file system descriptor file entry
`getfsent()`: get file system descriptor file entry ..... see **getfsent(3X)**
`getfsfile()`: get file system descriptor file entry ..... see **getfsent(3X)**
`getfsspec()`: get file system descriptor file entry ..... see **getfsent(3X)**
`getfstype()`: get file system descriptor file entry ..... see **getfsent(3X)**
getgrent(3C): `getgrent()`, `getgrgid()`, `getgrnam()`, `setgrent()`, `endgrent()`, `fgetgrent()` ..... get group file entry
`getgrgid()`, `getgrnam()`: get group file entry ..... see **getgrent(3C)**
`gethcwd()`: get path-name of current working directory ..... see **getcwd(3C)**
`gethostbyaddr()`: get network host entry ..... see **gethostent(3N)**

Entry Name(Section): *name* **Description**

`gethostbyname()`: get network host entry ..... see **gethostent(3N)**
gethostent(3N): `gethostent()`, `gethostbyaddr()`, `gethostbyname()`, `sethostent()`, `endhostent()` ..... get network host entry
`gethostent()`: get network host entry ..... see **gethostent(3N)**
`getlocale()`: get the locale of a program ..... see **setlocale(3C)**
getlogin(3C): `getlogin()` ..... get login name
getmntent(3X): `getmntent()`, `setmntent()`, `addmntent()`, `endmntent()`, `hasmntopt()` ..... get file system descriptor file entry
`getnetbyaddr()`: get network entry ..... see **getnetent(3N)**
`getnetbyname()`: get network entry ..... see **getnetent(3N)**
getnetent(3N): `getnetent()`, `getnetbyaddr()`, `getnetbyname()`, `setnetent()`, `endnetent()` ..... get network entry
`getnetent()`: get network entry ..... see **getnetent(3N)**
getnetgrent(3C): `getnetgrent()`, `setnetgrent()`, `endnetgrent()`, `innetgr()` ..... get network group entry
getopt(3C): `getopt()`, `optarg`, `optind`, `opterr` ..... get option letter from argument vector
getpass(3C): `getpass()` ..... read a password
`getprotobyname()`: get protocol entry ..... see **getprotoent(3N)**
`getprotobynumber()`: get protocol entry ..... see **getprotoent(3N)**
getprotoent(3N): `getprotoent()`, `getprotobynumber()`, `getprotobyname()`, `setprotoent()`, `endprotoent()` ..... get protocol entry
`getprotoent()`: get protocol entry ..... see **getprotoent(3N)**
getpw(3C): `getpw()` ..... get name from UID
getpwent(3C): `getpwent()`, `getpwuid()`, `getpwnam()`, `setpwent()`, `endpwent()`, `fgetpwent()` ..... get password file entry
`getpwent()`: get password file entry ..... see **getpwent(3C)**
`getpwent()`: get secure password file entry ..... see **getspwent(3C)**
getrpcent(3C): `getrpcent()`, `getrpcbyname()`, `getrpcbynumber()` ..... get rpc entry
getrpcport(3N): `getrpcport()` ..... get RPC port number
gets(3S): `gets()`, `fgets()` ..... get a string from a stream
`getservbyname()`: get service entry ..... see **getservent(3N)**
`getservbyport()`: get service entry ..... see **getservent(3N)**
getservent(3N): `getservent()`, `getservbyport()`, `getservbyname()`, `setservent()`, `endservent()` ..... get service entry
`getservent()`: get service entry ..... see **getservent(3N)**
`getspwaid()`: get secure password file entry ..... see **getspwent(3C)**
getspwent(3C): `getpwent()`, `getpwuid()`, `getpwnam()`, `setpwent()`, `endpwent()`, `fgetpwent()` ..... get secure password file entry
getsubopt(3C): `getsubopt()` ..... parse suboptions from a string.
gettimer(3C): `gettimer` ..... get value of a per-process timer
getusershell(3C): `getusershell()`, `setusershell()`, `endusershell()` ..... get legal user shells
getut(3C): `getutent()`, `getutid()`, `getutline()`, `pututline()`, `setutent()`, `endutent()`, `utmpname()` ..... access utmp file entry
`getutent()`: access utmp file entry ..... see **getut(3C)**
getwc(3C): `getwc()`, `getwchar()`, `fgetwc()` ..... get wide character from a stream file
`getwchar()`: get wide character from a stream file ..... see **getwc(3C)**
`getw()`: get word from a stream file ..... see **getc(3S)**
getws(3C): `getws()`, `fgetws()` ..... get a wide string from a stream
glob(3C): `glob()`, `globfree()` ..... file name generation function
`globfree()` – file name generation function ..... see **glob(3C)**
`gmtime()`: convert date and time to string ..... see **ctime(3C)**
gpio_get_status(3I): `gpio_get_status` ..... return status lines of GPIO card
gpio_set_ctl(3I): `gpio_set_ctl` ..... set control lines on GPIO card
`gsignal()`: software signals ..... see **ssignal(3C)**
`hasmntopt()`: get file system descriptor file entry ..... see **getmntent(3X)**
`hcreate()`: manage hash search tables ..... see **hsearch(3C)**
`hdestroy()`: manage hash search tables ..... see **hsearch(3C)**
`herror` – resolver routines ..... see **resolver(3N)**

Table of Contents
Volume 2

Entry Name(Section): *name* — **Description**

hidecdf() – manipulate context-dependent file path names ..... see **getcdf(3C)**
hpib_abort(3I): **hpib_abort()** ..... stop activity on specified HP-IB bus
hpib_address_ctl(3I): **hpib_address_ctl()** ..... set HP-IB bus address for an interface
hpib_atn_ctl(3I): **hpib_atn_ctl()** ..... control Attention line on HP-IB
hpib_bus_status(3I): **hpib_bus_status()** ..... return status of HP-IB interface
hpib_card_ppoll_resp(3I): **hpib_card_ppoll_resp()** ..... control response to parallel poll on HP-IB
hpib_eoi_ctl(3I): **hpib_eoi_ctl()** ..... control EOI mode for HP-IB file
hpib_io(3I): **hpib_io()** ..... perform I/O with an HP-IB channel from buffers
hpib_parity_ctl(3I): **hpib_parity_ctl()** ..... enable/disable odd parity on ATN commands
hpib_pass_ctl(3I): **hpib_pass_ctl()** ..... change active controllers on HP-IB
hpib_ppoll(3I): **hpib_ppoll()** ..... conduct parallel poll on HP-IB bus
hpib_ppoll_resp_ctl(3I): **hpib_ppoll_resp_ctl()** ..... define interface parallel poll response
hpib_ren_ctl(3I): **hpib_ren_ctl()** ..... control the Remote Enable line on HP-IB
hpib_rqst_srvce(3I): **hpib_rqst_srvce()** ..... allow interface to enable SRQ line on HP-IB
hpib_send_cmnd(3I): **hpib_send_cmnd()** ..... send command bytes over HP-IB
hpib_spoll(3I): **hpib_spoll()** ..... conduct a serial poll on HP-IB bus
hpib_status_wait(3I): **hpib_status_wait()** ..... wait until the requested status condition becomes true
hpib_wait_on_ppoll(3I): **hpib_wait_on_ppoll()** ..... wait until a particular parallel poll value occurs
hppac(3X) ..... Series 800 HP 3000-mode packed decimal library
hsearch(3C): **hsearch()**, **hcreate()**, **hdestroy()** ..... manage hash search tables
htonl(), **htons()**: convert values from host to network byte order ..... see **byteorder(3N)**
hypot(3M): **hypot()**, **cabs()** ..... Euclidean distance, complex absolute value function
iconv(3C): **iconvclose()**, **iconvopen()**, **iconvsize()**, **iconvlock()**, **ICONV**, **ICONV1**, **ICONV2** ..... code set conversion routines
idtolang(): NLS information about native languages ..... see **langinfo(3C)**
ieee(3M): **copysign()**, **copysignf()**, **drem()**, **finite()**, **finitef()**, **logb()**, **scalb()** ..... copysign, remainder, classification, exponent manipulations
index(): BSD portability string routine ..... see **string(3C)**
inet(3N): **inet_addr()**, **inet_network()**, **inet_ntoa()**, **inet_makeaddr()**, **inet_lnaof()**, **inet_netof()** ..... Internet address manipulation routines
inet_addr(): Internet address manipulation routines ..... see **inet(3N)**
inet_lnaof(): Internet address manipulation routines ..... see **inet(3N)**
inet_makeaddr(): Internet address manipulation routines ..... see **inet(3N)**
inet_netof(): Internet address manipulation routines ..... see **inet(3N)**
inet_network(): Internet address manipulation routines ..... see **inet(3N)**
inet_ntoa(): Internet address manipulation routines ..... see **inet(3N)**
initgroups(3C): **initgroups()** ..... initialize group access list
initopt(3N): **initopt()** ..... initialize a NetIPC option buffer
io_burst(3I): **io_burst()** ..... perform low-overhead I/O on an HP-IB/GPIO/parallel channel
io_dma_ctl(3I): **io_dma_ctl()** ..... control DMA allocation for an interface
io_eol_ctl(3I): **io_eol_ctl** ..... set up read termination character on special file
io_get_term_reason(3I): **io_get_term_reason()** ..... determine how last read terminated
io_interrupt_ctl(3I): **io_interrupt_ctl()** ..... enable/disable interrupts for the associated eid
io_lock(3I): **io_lock**, **io_unlock** ..... lock and unlock an interface
io_on_interrupt(3I): **io_on_interrupt()** ..... device interrupt (fault) control
io_reset(3I): **io_reset()** ..... reset an I/O interface
io_speed_ctl(3I): **io_speed_ctl()** ..... inform system of required transfer speed
io_timeout_ctl(3I): **io_timeout_ctl()** ..... establish a time limit for I/O operations
io_unlock: lock and unlock an interface ..... see **io_lock(3I)**
io_width_ctl(3I): **io_width_ctl()** ..... set width of data path
ipcerrmsg(3N): **ipcerrmsg()**, **ipcerrstr()** ..... provide text describing NetIPC error number
ipcerrstr() – provide text describing NetIPC error number ..... see **ipcerrmsg(3N)**
is_68010_present: check for presence of hardware capabilities ..... see **is_hw_present(3C)**
is_68881_present: check for presence of hardware capabilities ..... see **is_hw_present(3C)**
is_98248A_present: check for presence of hardware capabilities ..... see **is_hw_present(3C)**
is_98635A_present: check for presence of hardware capabilities ..... see **is_hw_present(3C)**
isalnum(): classify characters ..... see **ctype(3C)**

Entry Name(Section): *name* — **Description**

`isalpha()`: classify characters ..... see **ctype(3C)**
`isascii()`: classify characters ..... see **ctype(3C)**
`isatty()`: find name of a terminal ..... see **ttyname(3C)**
`iscntrl()`: classify characters ..... see **ctype(3C)**
`isdigit()`: classify characters ..... see **ctype(3C)**
`isgraph()`: classify characters ..... see **ctype(3C)**
is_hw_present(3C): `is_68010_present`, `is_68881_present`, `is_98635A_present`, `is_98248A_present` ..... check for presence of hardware capabilities
isinf(3M): `isinf()`, `isinff()` ..... test for INFINITY
`isinff()`: test for INFINITY (float version) ..... see **isinf(3M)**
`islower()`: classify characters ..... see **ctype(3C)**
isnan(3M): `isnan()`, `isnanf()` ..... test for NaN
`isnanf()`: test for NaN (float version) ..... see **isnan(3M)**
`isprint()`: classify characters ..... see **ctype(3C)**
`ispunct()`: classify characters ..... see **ctype(3C)**
`isspace()`: classify characters ..... see **ctype(3C)**
`isupper()`: classify characters ..... see **ctype(3C)**
`iswalnum`: classify wide characters ..... see **wctype(3C)**
`iswalpha`: classify wide characters ..... see **wctype(3C)**
`iswcntrl`: classify wide characters ..... see **wctype(3C)**
`iswdigit`: classify wide characters ..... see **wctype(3C)**
`iswgraph`: classify wide characters ..... see **wctype(3C)**
`iswlower`: classify wide characters ..... see **wctype(3C)**
`iswprint`: classify wide characters ..... see **wctype(3C)**
`iswpunct`: classify wide characters ..... see **wctype(3C)**
`iswspace`: classify wide characters ..... see **wctype(3C)**
`iswupper`: classify wide characters ..... see **wctype(3C)**
`iswxdigit`: classify wide characters ..... see **wctype(3C)**
`isxdigit()`: classify characters ..... see **ctype(3C)**
`j0()`: Bessel function ..... see **bessel(3M)**
`j1()`: Bessel function ..... see **bessel(3M)**
`jn()`: Bessel function ..... see **bessel(3M)**
`jrand48()`: generate pseudo-random numbers ..... see **drand48(3C)**
l3tol(3C): `l3tol()`, `ltol3()` ..... convert between 3-byte integers and long integers
`l64a`: convert between long integer and base-64 ASCII string ..... see **a64l(3C)**
langinfo(3C): `langinfo()`, `langtoid()`, `idtolang()`, `currlangid()` ..... native language NLS information
`langinit()`: initialize the NLS environment of a program ..... see **nl_init(3C)**
`langtoid()`: NLS information about native languages ..... see **langinfo(3C)**
`lcong48()`: generate pseudo-random numbers ..... see **drand48(3C)**
ldcvt(3C): `_ldecvt()`, `_ldfcvt()`, `_ldgcvt()` ..... convert long double floating-point number to string
`_ldecvt()` – convert long double floating-point number to string ..... see **ldcvt(3C)**
`ldecvt()` (`_ldecvt()`) – convert long double floating-point number to string ..... see **ldcvt(3C)**
`ldexp`: split floating-point into mantissa and exponent ..... see **frexp(3C)**
`_ldfcvt()` – convert long double floating-point number to string ..... see **ldcvt(3C)**
`ldfcvt()` (`_ldfcvt()`) – convert long double floating-point number to string ..... see **ldcvt(3C)**
`_ldgcvt()` – convert long double floating-point number to string ..... see **ldcvt(3C)**
`ldgcvt()` (`_ldgcvt()`) – convert long double floating-point number to string ..... see **ldcvt(3C)**
`ldiv()`: long integer division and remainder ..... see **div(3C)**
`lfind()`: linear search and update ..... see **lsearch(3C)**
`lgamma()`: log gamma function ..... see **gamma(3M)**
`localtime()`: convert date and time to string ..... see **ctime(3C)**
`log10()`: common logarithm function ..... see **exp(3M)**
`log10f()`: common logarithm function (float version) ..... see **exp(3M)**
`log2()`: base 2 logarithm function ..... see **exp(3M)**
`log2f()`: base 2 logarithm function (float version) ..... see **exp(3M)**
`logb()`, `scalb()`: exponent manipulations ..... see **ieee(3M)**
`logf()`: natural logarithm function (float version) ..... see **exp(3M)**

Table of Contents
Volume 2

Entry Name(Section): *name* — **Description**

logname(3C): `logname()` ..... return login name of user
`log()`: natural logarithm function ..... see **exp(3M)**
`longjmp()`: restore stack environment for non-local goto ..... see **setjmp(3C)**
`lrand48()`: generate pseudo-random numbers ..... see **drand48(3C)**
lsearch(3C): `lsearch()`, `lfind()` ..... linear search and update
`ltoa()`: long to ASCII decimal ..... see **ltostr(3C)**
`ltol3()`: convert between 3-byte integers and long integers ..... see **l3tol(3C)**
ltostr(3C): `ltostr()`, `ultostr()`, `ltoa()`, `ultoa()` ..... convert long integers to strings
`mallinfo`: main memory allocator ..... see **malloc(3C)**
malloc(3C): `malloc`, `free`, `realloc`, `calloc`, `mallopt`, `mallinfo`, memorymap ..... main memory allocator
`mallopt`: main memory allocator ..... see **malloc(3C)**
matherr(3M): `matherr()` ..... error-handling function
`mblen()`: multibyte characters and strings conversions ..... see **multibyte(3C)**
`mbstowcs()`: multibyte characters and strings conversions ..... see **multibyte(3C)**
`mbtowc()`: multibyte characters and strings conversions ..... see **multibyte(3C)**
`mcrt0.o`: execution startup routines ..... see **crt0(3)**
`memccpy()`: memory operations ..... see **memory(3C)**
`memchr()`: memory operations ..... see **memory(3C)**
`memcmp()`: memory operations ..... see **memory(3C)**
`memcpy()`: memory operations ..... see **memory(3C)**
`memmove()`: memory operations ..... see **memory(3C)**
memory(3C): `memccpy()`, `memchr()`, `memcmp()`, `memcpy()`, `memset()` ..... memory operations
`memorymap`: main memory allocator ..... see **malloc(3C)**
`memset()`: memory operations ..... see **memory(3C)**
`mfrt0.o`: execution startup routines ..... see **crt0(3)**
mkfifo(3C): `mkfifo()` ..... make a FIFO special file
mktemp(3C): `mktemp()` ..... make a unique file name
`mktime()`: create calendar time value ..... see **ctime(3C)**
mktimer(3C): `mktimer` ..... allocate a per-process timer
`modf`: split floating-point into mantissa and exponent ..... see **frexp(3C)**
monitor(3C): `monitor()` ..... prepare execution profile
mount(3N): `mount()` ..... keep track of remotely mounted filesystems
`mrand48()`: generate pseudo-random numbers ..... see **drand48(3C)**
multibyte(3C): `mblen()`, `mbtowc()`, `mbstowcs()`, `wctomb()`, `wcstombs()` ..... multibyte characters and strings conversions
ndbm(3X): `dbm_open`, `dbm_close`, `dbm_fetch`, `dbm_store`, `dbm_delete`, `dbm_firstkey`, `dbm_nextkey`, `dbm_error`, `dbm_clearerr` ..... database subroutines
net_aton(3C): `net_aton()`, `net_ntoa()` ..... network station address string conversion routines
`net_ntoa()`: network station address string conversion routines ..... see **net_aton(3C)**
`nextkey()`: database subroutines ..... see **dbm(3X)**
nlappend(3X): `nlappend()` ..... append appropriate language identification to valid MPE file name
`nl_asctime()`: convert date and time to string ..... see **ctime(3C)**
`nl_ascxtime()`: convert date and time to string ..... see **ctime(3C)**
`nl_atof`: convert string to double-precision number ..... see **strtod(3C)**
nlcollate(3X): `nlcollate()` ..... compare strings using MPE language-dependent collating sequence
nl_conv(3C): `nl_toupper()`, `nl_tolower()` ..... translate characters for use with NLS
nlconvclock(3X): `nlconvclock()` ..... check and convert time string to MPE internal format
nlconvcustdate(3X): `nlconvcustdate()` ..... convert date string to MPE packed date format
nlconvnum(3X): `nlconvnum()` ..... convert MPE native language formatted number to ASCII number
`nl_ctime()`: convert date and time to string ..... see **ctime(3C)**
nl_ctype(3C): `nl_isalpha()`, `nl_isupper()`, `nl_islower()`, `nl_isdigit()`, `nl_isxdigit()`, `nl_isalnum()`, `nl_isspace()`, `nl_ispunct()`, `nl_isprint()`, `nl_isgraph()`, `nl_iscntrl()` ..... classify characters for use with NLS
`nl_cxtime()`: convert date and time to string ..... see **ctime(3C)**
nlfindstr(3X): `nlfindstr()` ..... search for string in another string using MPE character set definition
nlfmtcal(3X): `nlfmtcalendar()` ..... format MPE packed date using localized format

Entry Name(Section): *name* **Description**

nlfmtclock(3X): `nlfmtclock()` ..... format MPE time of day using localized format
nlfmtcustdate(3X): `nlfmtcustdate()` ..... format MPE packed date using custom date
nlfmtdate(3X): `nlfmtdate()` ..... format MPE date and time in localized format
nlfmtlongcal(3X): `nlfmtlongcal()` ..... format MPE packed date using long calendar format
nlfmtnum(3X): `nlfmtnum()` ..... convert ASCII number to MPE language-specific formatted number
`nl_fprintf()`: print formatted output ..... see **printf(3S)**
`nl_fscanf`: formatted input conversion, read from stream file ..... see **scanf(3S)**
`nl_gcvt()`: convert floating-point number to string ..... see **ecvt(3C)**
nlgetlang(3X): `nlgetlang()` ..... return current user, data, or system default language
nlinfo(3X): `nlinfo()` ..... return MPE language-dependent information
nl_init(3C): `nl_init()`, `langinit()` ..... initialize the NLS environment of a program
`nl_isalnum()`: classify characters for use with NLS ..... see **nl_ctype(3C)**
`nl_isalpha()`: classify characters for use with NLS ..... see **nl_ctype(3C)**
`nl_iscntrl()`: classify characters for use with NLS ..... see **nl_ctype(3C)**
`nl_isdigit()`: classify characters for use with NLS ..... see **nl_ctype(3C)**
`nl_isgraph()`: classify characters for use with NLS ..... see **nl_ctype(3C)**
`nl_islower()`: classify characters for use with NLS ..... see **nl_ctype(3C)**
`nl_isprint()`: classify characters for use with NLS ..... see **nl_ctype(3C)**
`nl_ispunct()`: classify characters for use with NLS ..... see **nl_ctype(3C)**
`nl_isspace()`: classify characters for use with NLS ..... see **nl_ctype(3C)**
nlist(3C): `nlist()` ..... get entries from name list
`nl_isupper()`: classify characters for use with NLS ..... see **nl_ctype(3C)**
`nl_isxdigit()`: classify characters for use with NLS ..... see **nl_ctype(3C)**
nljudge(3X): `nljudge()` ..... judge whether character is one- or multi-byte Asian using MPE character table
nlkeycompare(3X):
`nlkeycompare()` ..... compare character arrays (key1, key2) using MPE collation table
nl_nl_langinfo(3C): `nl_langinfo()` ..... NLS information about native languages
nlnumspec(3X): `nlnumspec()` ..... return number convert/format information for MPE routines
`nl_printf()`: print formatted output ..... see **printf(3S)**
nlrepchar(3X): `nlrepchar()` ..... replace non-displayable characters MPE character set table
`nl_scanf`: formatted input conversion, read from stream file ..... see **scanf(3S)**
nlscanmove(3X): `nlscanmove()` .... move, scan and case shift character strings using MPE character set table
`nl_sprintf()`: print formatted output ..... see **printf(3S)**
`nl_sscanf`: formatted input conversion, read from stream file ..... see **scanf(3S)**
`nl_strcmp`, `nl_strncmp`: character string operations ..... see **string(3C)**
nl_string(3C): `strcmp8()`, `strncmp8()`, `strcmp16()`, `strncmp16()` ..... non-ASCII string collation
`nl_strtod`: convert string to double-precision number ..... see **strtod(3C)**
nlsubstr(3X): `nlsubstr()` ..... extract substring using MPE character set table
nlswitchbuf(3X): `nlswitchbuf()` ..... convert string screen order using MPE character set table
`nl_tolower()`: translate characters for use with NLS ..... see **nl_conv(3C)**
nl_tools_16(3C): `firstof2()`, `secof2()`, `byte_status()`, `c_colwidth()`,
`FIRSTof2()`, `SECof2()`, `BYTE_STATUS()`, `C_COLWIDTH()`, `CHARAT()`,
`ADVANCE()`, `CHARADV()`, `WCHAR()`, `WCHARADV()` ..... tools to process 16-bit characters
`nl_toupper()`: translate characters for use with NLS ..... see **nl_conv(3C)**
nltranslate(3X): `nltranslate()` ..... translate ASCII EBCDIC using MPE conversion table
`nrand48()`: generate pseudo-random numbers ..... see **drand48(3C)**
`ntohl()`, `ntohs()`: convert values from network to host byte order ..... see **byteorder(3N)**
`opendir()`: directory operations ..... see **directory(3C)**
`openlog()`: control system log ..... see **syslog(3C)**
`optarg`: get option letter from argument vector ..... see **getopt(3C)**
`opterr`: get option letter from argument vector ..... see **getopt(3C)**
`optind`: get option letter from argument vector ..... see **getopt(3C)**
optoverhead(3N): `optoverhead()` ..... return number of bytes needed by a NetIPC option
`pclose()`: initiate pipe I/O to/from a process ..... see **popen(3S)**
perror(3C): `perror()`, `errno()`, `sys_errlist()`, `sys_nerr()` ..... system error messages
pfm_$intro(3): pfm_$intro ..... fault management
pfm_$cleanup(3): `pfm_$cleanup()` ..... establish a cleanup handler

Table of Contents
Volume 2

Entry Name(Section): *name* — **Description**

pfm_$enable(3): `pfm_$enable()` ..... enable asynchronous faults
pfm_$enable_faults(3): `pfm_$enable_faults()` ..... enable asynchronous faults
pfm_$inhibit(3): `pfm_$inhibit()` ..... inhibit asynchronous faults
pfm_inhibit(3): `pfm_inhibit` ..... pointer entry for conflicting online manual entries
pfm_$inhibit_faults(3): `pfm_$inhibit_faults` ..... inhibit asynchronous faults; allow time-sliced task switching
pfm_$init(3): `pfm_$init()` ..... initialize the process fault manager package
pfm_$reset_cleanup(3): `pfm_$reset_cleanup` ..... reset a cleanup handler
pfm_$rls_cleanup(3): `pfm_$rls_cleanup()` ..... release a cleanup handler
pfm_$signal(3): `pfm_$signal()` ..... signal the calling process
pgm_$intro(3): pgm_$intro ..... program management
pgm_$exit(3): `pgm_$exit()` ..... exit a program
popen(3S): `popen()`, `pclose()` ..... initiate pipe I/O to/from a process
`powf()`: power function (float version) ..... see **exp(3M)**
`pow()`: power function ..... see **exp(3M)**
printf(3S): `printf()`, `nl_printf()`, `fprintf()`, `nl_fprintf()`, `sprintf()`, `nl_sprintf()` ..... print formatted output
printmsg(3C): `printmsg()`, `fprintmsg()`, `sprintmsg()` print formatted output with numbered arguments
ptsname(3C): `ptsname` ..... get the name of a slave pty
putc(3S): `putc()`, `putchar()`, `fputc()`, `putw()` ..... put character or word on a stream
`putchar()`: put character on a stream ..... see **putc(3S)**
putenv(3C): `putenv()` ..... change or add value to environment
putpwent(3C): `putpwent()` ..... write password file entry
puts(3S): `puts()`, `fputs()` ..... put a string on a stream
putspwent(3C): `putspwent()` ..... write secure password file entry
`_pututline()`: access utmp file entry ..... see **getut(3C)**
`pututline()`: access utmp file entry ..... see **getut(3C)**
putwc(3C): `putwc()`, `putwchar()`, `fputwc()`, `putw()` ..... put wide character on a stream
`putwchar()`: put wide character on a stream ..... see **putwc(3C)**
`putw()`: put word on a stream ..... see **putc(3S)**
putws(3C): `putws()`, `fputws()` ..... put a wide string on a stream
qsort(3C): `qsort()` ..... quicker sort
rand(3C): `rand()`, `srand()` ..... simple random-number generator
rcmd(3N): `rcmd()`, `rresvport()`, `ruserok()` ..... return a stream to a remote command
`readdir()`: directory operations ..... see **directory(3C)**
readopt(3N): `readopt()` ..... obtain option code and data from NetIPC option buffer
`realloc`: main memory allocator ..... see **malloc(3C)**
regcmp(3X): `regcmp()`, `regex()` ..... compile and execute regular expression
regcomp(3C): `regcomp()`, `regerror()`, `regexec()`, `regfree()` ..... regular expression matching routines
`regerror()` – regular expression matching routines ..... see **regcomp(3C)**
`regex()`: compile and execute regular expression ..... see **regcmp(3X)**
`regexec()` – regular expression matching routines ..... see **regcomp(3C)**
regexp(3X): `compile()`, `step()`, `advance()` ..... regular expression compile and match routines
`regfree()` – regular expression matching routines ..... see **regcomp(3C)**
reltimer(3C): `reltimer` ..... relatively arm a per-process timer
`remexportent()` – access exported file system information ..... see **exportent(3N)**
remove(3C): `remove()` ..... remove a file
`res_init`, `res_mkquery`, `res_query`, `res_search`, `res_send`, – resolver routines ..... see **resolver(3N)**
resolver(3N): `res_init`, `res_mkquery`, `res_query`, `res_search`, `res_send`, `dn_comp`, `dn_expand`, `herror` ..... resolver routines
`rewinddir()`: directory operations ..... see **directory(3C)**
`rewind`: reposition a file pointer in a stream ..... see **fseek(3S)**
rexec(3N): `rexec()` ..... return stream to a remote command
`rindex()`: BSD portability string routine ..... see **string(3C)**
`rint()`: round-to-nearest function ..... see **floor(3M)**
rmtimer(3C): `rmtimer` ..... free a per-process timer
rnusers(3N): `rnusers()`, `rusers()` ..... return information about users on remote machines
rpc(3C): `rpc()` ..... library routines for remote procedure calls

Entry Name(Section): *name* — **Description**

rresvport() – return a stream to a remote command ..... see **rcmd(3N)**
rstat(3N): **rstat()**, **havedisk()** ..... get performance data from remote kernel
ruserok() return a stream to a remote command ..... see **rcmd(3N)**
rwall(3N): **rwall()** ..... write to specified remote machines
scalb(), **logb()**: exponent manipulations ..... see **ieee(3M)**
scandir(3C): **scandir()**, **alphasort()** ..... scan a directory
scanf(3S): **scanf()**, **fscanf()**, **sscanf()**, **nl_scanf**, **nl_fscanf**, **nl_sscanf** ..... formatted input conversion, read from stream file
secof2(), **SECof2()**: process 16-bit characters ..... see **nl_tools_16(3C)**
seed48(): generate pseudo-random numbers ..... see **drand48(3C)**
seekdir(): directory operations ..... see **directory(3C)**
setaclentry(3C): **setaclentry()**, **fsetaclentry()** ..... add, modify, or delete access control list entry
setbuf(3S): **setbuf()**, **setvbuf()** ..... assign buffering to a stream file
setccent(): get cluster configuration entry ..... see **getccent(3C)**
setclock(3C): **setclock** ..... set value of system-wide clock
setexportent() – access exported file system information ..... see **exportent(3N)**
setfsent(): get file system descriptor file entry ..... see **getfsent(3X)**
setgrent(): get group file entry ..... see **getgrent(3C)**
sethostent(): get network host entry ..... see **gethostent(3N)**
setjmp(3S): **setjmp()**, **longjmp()** ..... save/restore stack environment for non-local goto
_setjmp(): save stack environment for non-local goto ..... see **setjmp(3C)**
setkey(): generate hashing encryption ..... see **crypt(3C)**
setlocale(3C): **setlocale()**, **getlocale()** ..... set and get the locale of a program
setlogmask(): control system log ..... see **syslog(3C)**
setmntent(): get file system descriptor file entry ..... see **getmntent(3X)**
setnetent(): get network entry ..... see **getnetent(3N)**
setprotoent(): get protocol entry ..... see **getprotoent(3N)**
setpwent(): get password file entry ..... see **getpwent(3C)**
setpwent(): get secure password file entry ..... see **getspwent(3C)**
setservent(): get service entry ..... see **getservent(3N)**
setusershell() – rewind legal user shells file ..... see **getusershell(3C)**
setutent(): access utmp file entry ..... see **getut(3C)**
setvbuf(): assign buffering to a stream file ..... see **setbuf(3S)**
sgetl(): access long integer data in a machine-independent fashion ..... see **sputl(3X)**
shl_definesym() – define new symbol for shared libraries ..... see **shl_load(3X)**
shl_findsym() – explicit load of shared libraries ..... see **shl_load(3X)**
shl_findsym() – get information about shared libraries ..... see **shl_load(3X)**
shl_gethandle() – get shared library information ..... see **shl_load(3X)**
shl_load(3X): **shl_load()**, **shl_findsym()**, **shl_unload()**, **shl_get()** ..... explicit load of shared libraries
shl_load() – explicit load of shared libraries ..... see **shl_load(3X)**
shl_unload() – unload shared libraries ..... see **shl_load(3X)**
sigaddset(): initialize, manipulate, and test signal sets ..... see **sigsetops(3C)**
sigdelset(): initialize, manipulate, and test signal sets ..... see **sigsetops(3C)**
sigemptyset(): initialize, manipulate, and test signal sets ..... see **sigsetops(3C)**
sigfillset(): initialize, manipulate, and test signal sets ..... see **sigsetops(3C)**
sigismember(): initialize, manipulate, and test signal sets ..... see **sigsetops(3C)**
signgam(): log gamma function ..... see **gamma(3M)**
sigsetops(3C): **sigemptyset()**, **sigfillset()**, **sigaddset()**, **sigdelset()**, **sigismember()** ..... initialize, manipulate, and test signal sets
sindf(): trigonometric sine function (float, degrees) ..... see **trigd(3M)**
sind(): trigonometric sine function (degrees) ..... see **trigd(3M)**
sinf(): trigonometric sine function (float) ..... see **trig(3M)**
sinh(3M): **sinh()**, **cosh()**, **tanh()**, sinhf(), **coshf()**, **tanhf()** ..... hyperbolic functions
sinhf(): hyperbolic cosine function (float version) ..... see **sinh(3M)**
sin(): trigonometric sine function ..... see **trig(3M)**
sleep(3C): **sleep()** ..... suspend execution for interval
spray(3N): **spray** ..... scatter data for network checking

Table of Contents
Volume 2

Entry Name(Section): *name* — **Description**

sprintf(): print formatted output ..... see **printf(3S)**
sprintmsg(): print formatted output with numbered arguments ..... see **printmsg(3C)**
sputl(3X): **sputl()**, **sgetl()** ..... access long integer data in a machine-independent fashion
sqrtf(): square root function (float version) ..... see **exp(3M)**
sqrt(): square root function ..... see **exp(3M)**
srand48(): generate pseudo-random numbers ..... see **drand48(3C)**
srand(): simple random-number generator ..... see **rand(3C)**
sscanf(): formatted input conversion, read from stream file ..... see **scanf(3S)**
ssignal(3C): **ssignal()**, **gsignal()** ..... software signals
statfsdev(3C): **statfsdev()**, **fstatfsdev()** ..... get file system statistics
stdio(3S): **stdio()** ..... standard buffered input/output stream file package
stdipc(3C): **ftok()** ..... standard interprocess communication package
step(): regular expression compile and match routines ..... see **regexp(3X)**
store(): database subroutines ..... see **dbm(3X)**
strcat(), **strncat()**: character string operations ..... see **string(3C)**
strchr(), **strrchr()**: character string operations ..... see **string(3C)**
strcmp8(), **strcmp16()**: non-ASCII string collation ..... see **nl_string(3C)**
strcmp(), **strncmp()**: character string operations ..... see **string(3C)**
strcoll(): character string operations ..... see **string(3C)**
strcpy(), **strncpy()**: character string operations ..... see **string(3C)**
strerror(): system error messages ..... see **perror(3C)**
strftime(3C): **strftime()** ..... convert date and time to string
string(3C): **strcat()**, **strncat()**, **strcmp()**, **strncmp()**, **strcpy()**, **strncpy()**, **strlen()**, **strchr()**, **strrchr()**, **strpbrk()**, **strspn()**, **strcspn()**, **strtok()**, **nl_strcmp**, **nl_strncmp** ..... character string operations
strlen(): character string operations ..... see **string(3C)**
strncmp8(), **strncmp16()**: non-ASCII string collation ..... see **nl_string(3C)**
strord(3C): **strord()** ..... convert string data order
strpbrk(): character string operations ..... see **string(3C)**
strrstr(): character string operations ..... see **string(3C)**
strspn(), **strcspn()**: character string operations ..... see **string(3C)**
strstr(): character string operations ..... see **string(3C)**
strtoacl(3C): **strtoacl()**, **strtoaclpatt()** ..... convert string form to access control list structure
strtoaclpatt(): convert pattern string form to access control list (ACL) structure ..... see **strtoacl(3C)**
strtod(3C): **strtod()**, **atof()**, **nl_strtod**, **nl_atof** ..... convert string to double-precision number
strtok(): character string operations ..... see **string(3C)**
strtold(3C): **strtold()** ..... convert string to long double-precision number
strxfrm(): character string operations ..... see **string(3C)**
swab(3C): **swab()** ..... swap bytes
sys_errlist: system error messages ..... see **perror(3C)**
syslog(3C): **syslog()**, **openlog()**, **closelog()**, **setlogmask()** ..... control system log
sys_nerr: system error messages ..... see **perror(3C)**
system(3S): **system()** ..... issue a shell command
tandf(): trigonometric tangent function (float, degrees) ..... see **trigd(3M)**
tand(): trigonometric tangent function (degrees) ..... see **trigd(3M)**
tanf(): trigonometric tangent function (float) ..... see **trig(3M)**
tanhf(): hyperbolic tangent function (float version) ..... see **sinh(3M)**
tanh(): hyperbolic tangent function ..... see **sinh(3M)**
tan(): trigonometric tangent function ..... see **trig(3M)**
tcattribute(3C): **tcgetattr()**, **tcsetattr()** ..... control tty device
tccontrol(3C): **tcsendbreak()**, **tcdrain()**, **tcflush()**, **tcflow()** ..... tty line control functions
tcdrain(): tty line control functions ..... see **tccontrol(3C)**
tcflow(): tty line control functions ..... see **tccontrol(3C)**
tcflush(): tty line control functions ..... see **tccontrol(3C)**
tcgetattr(): get tty device attributes ..... see **tcattribute(3C)**
tcgetpgrp(3C): **tcgetpgrp()** ..... get foreground process group ID
tcsendbreak(): tty line control functions ..... see **tccontrol(3C)**

Entry Name(Section): *name* **Description**

`tcsetattr()`: set tty device attributes ..... see **tcattribute(3C)**
tcsetpgrp(3C): `tcsetpgrp()` ..... get foreground process group ID
`tdelete()`: manage binary search trees ..... see **tsearch(3C)**
`telldir()`: directory operations ..... see **directory(3C)**
`tempnam()`: create a name for a temporary file ..... see **tmpnam(3S)**
termcap(3X): `tgetent()`, `tgetnum()`, `tgetflag()`, `tgetstr()`, `tgoto()`, `tputs()` ..... emulate `/etc/termcap` access routines
`tfind()`: manage binary search trees ..... see **tsearch(3C)**
`tgetent()`: emulate `/etc/termcap` access routines ..... see **termcap(3X)**
`tgetflag()`: emulate `/etc/termcap` access routines ..... see **termcap(3X)**
`tgetnum()`: emulate `/etc/termcap` access routines ..... see **termcap(3X)**
`tgetstr()`: emulate `/etc/termcap` access routines ..... see **termcap(3X)**
`tgoto()`: emulate `/etc/termcap` access routines ..... see **termcap(3X)**
`timezone()`: convert date and time to string ..... see **ctime(3C)**
tmpfile(3S): `tmpfile()` ..... create a temporary file
tmpnam(3S): `tmpnam()`, `tempnam()` ..... create a name for a temporary file
`toascii()`: translate characters ..... see **conv(3C)**
`tolower()`, `_tolower`: translate characters ..... see **conv(3C)**
`toupper()`, `_toupper`: translate characters ..... see **conv(3C)**
`towlower()`: translate wide characters ..... see **wconv(3C)**
`towupper()`: translate wide characters ..... see **wconv(3C)**
`tputs()`: emulate `/etc/termcap` access routines ..... see **termcap(3X)**
trig(3M): `sin()`, `cos()`, `tan()`, `asin()`, `acos()`, `atan()`, `atan2()`, `sinf()`, `cosf()`, `tanf()`, `asinf()`, `acosf()`, `atanf()`, `atan2f()` ..... trigonometric functions
trigd(3M): `sind()`, `cosd()`, `tand()`, `asind()`, `acosd()`, `atand()`, `atan2d()`, `sindf()`, `cosdf()`, `tandf()`, `asindf()`, `acosdf()`, `atandf()`, `atan2df()` ..... degree-valued trigonometric functions
tsearch(3C): `tsearch()`, `tfind()`, `tdelete()`, `twalk()` ..... manage binary search trees
ttyname(3C): `ttyname()`, `isatty()` ..... find name of a terminal
ttyslot(3C): `ttyslot()` ..... find the slot in the utmp file of the current user
`twalk()`: manage binary search trees ..... see **tsearch(3C)**
`tzname()`: convert date and time to string ..... see **ctime(3C)**
`tzset()`: convert date and time to string ..... see **ctime(3C)**
`ultoa()`: unsigned long to ASCII decimal ..... see **ltostr(3C)**
`ultostr()`: unsigned long to ASCII ..... see **ltostr(3C)**
`undial()`: establish an out-going terminal line connection ..... see **dial(3C)**
ungetc(3S): `ungetc()` ..... push character back into input stream
ungetwc(3C): `ungetwc()` ..... push wide character back into input stream
utmp file entry ..... see **getut(3C)**
`utmpname()`: access utmp file entry ..... see **getut(3C)**
`vfprintf()`: print formatted output of a varargs argument list ..... see **vprintf(3S)**
`vfscanf()`: formatted input conversion to a varargs argument ..... see **vscanf(3S)**
vprintf(3S): `vprintf()`, `vfprintf()`, `vsprintf()` ..... print formatted output of a varargs argument list
vscanf(3S): `vscanf()`, `vfscanf()`, `vsscanf()` ..... formatted input conversion to a varargs argument
`vsprintf()`: print formatted output of a varargs argument list ..... see **vprintf(3S)**
`vsscanf()`: formatted input conversion to a varargs argument ..... see **vscanf(3S)**
`WCHARADV()`: process 16-bit characters ..... see **nl_tools_16(3C)**
`WCHAR()`: process 16-bit characters ..... see **nl_tools_16(3C)**
wconv(3C): `towupper()`, `towlower()` ..... translate wide characters
wcscat, wcsncat: wide character string operations ..... see **wcstring(3C)**
wcschr, wcsrchr: wide character string operations ..... see **wcstring(3C)**
wcscmp, wcsncmp: wide character string operations ..... see **wcstring(3C)**
wcscoll: wide character string operations ..... see **wcstring(3C)**
wcscpy, wcsncpy: wide character string operations ..... see **wcstring(3C)**
wcsftime(3C): `wcsftime()` ..... convert date and time to wide-character string
wcslen: wide character string operations ..... see **wcstring(3C)**
wcspbrk: wide character string operations ..... see **wcstring(3C)**

Table of Contents
Volume 2

Entry Name(Section): *name* **Description**

wcsspn, wcscspn: wide character string operations ..... see **wcstring(3C)**
wcstod(3C): `wcstod()` ..... convert wide character string to double-precision number
wcstok: wide character string operations ..... see **wcstring(3C)**
wcstring(3C): `wcscat`, `wcsncat`, `wcscmp`, `wcsncmp`, `wcscpy`, `wcsncpy`, `wcslen`, `wcschr`, `wcsrchr`, `wcspbrk`, `wcsspn`, `wcscspn`, `wcstok`, `nl_wcscmp`, `nl_wcsncmp` ..... wide character string operations
wcswcs: wide character string operations ..... see **wcstring(3C)**
wcswidth: wide character string operations ..... see **wcstring(3C)**
wctomb(), wctombs(): multibyte characters and strings conversions ..... see **multibyte(3C)**
wctype(3C): `iswalpha`, `iswupper`, `iswlower`, `iswdigit`, `iswxdigit`, `iswalnum`, `iswspace`, `iswpunct`, `iswprint`, `iswgraph`, `iswcntrl` ..... classify wide characters
wcwidth: wide character string operations ..... see **wcstring(3C)**
wordexp 3C: `wordexp`, `wordfree` – perform word expansions
xdr(3C): `xdr()` ..... library routines for external data representation
y0(): Bessel function ..... see **bessel(3M)**
y1(): Bessel function ..... see **bessel(3M)**
yn(): Bessel function ..... see **bessel(3M)**
yp_all() – Network Information Service client interface ..... see **ypclnt(3C)**
yp_bind() – Network Information Service client interface ..... see **ypclnt(3C)**
ypclnt(3C): `ypclnt()`, `yp_all()`, `yp_bind()`, `yp_first()`, `yp_get_default_domain()`, `yp_master()`, `yp_match()`, `yp_next()`, `yp_order()`, `yp_unbind()`, `yperr_string()`, `ypprot_err()` ..... Network Information Service client interface
yperr_string() – Network Information Service client interface ..... see **ypclnt(3C)**
yp_first() – Network Information Service client interface ..... see **ypclnt(3C)**
yp_get_default_domain() – Network Information Service client interface ..... see **ypclnt(3C)**
yp_master() – Network Information Service client interface ..... see **ypclnt(3C)**
yp_match() – Network Information Service client interface ..... see **ypclnt(3C)**
yp_next() – Network Information Service client interface ..... see **ypclnt(3C)**
yp_order() – Network Information Service client interface ..... see **ypclnt(3C)**
yppasswd(3N): `yppasswd()` ..... update user password in Network Information Service
ypprot_err() – Network Information Service client interface ..... see **ypclnt(3C)**
yp_unbind() – Network Information Service client interface ..... see **ypclnt(3C)**

Section 2:
System Calls

NAME

intro - introduction to system calls

DESCRIPTION

This section describes all of the system calls. All of these calls return a function result. This result indicates the status of the call. Typically, a zero or positive result indicates that the call completed successfully, and -1 indicates an error. The individual descriptions specify the details. An error number is also made available in the external variable `errno` (see *errno*(2)). Note: `errno` is not cleared on successful calls. Therefore, it should be tested only after an error has been indicated.

SEE ALSO

intro(3), errno(2), hier(5).

The introduction to this manual.

NAME

accept - accept a connection on a socket

SYNOPSIS

```
#include <sys/socket.h>

AF_CCITT only:
#include <x25/x25addrstr.h>

int accept(int s, void *addr, int *addrlen);
```

DESCRIPTION

`accept()` is used with connection-based socket types, such as `SOCK_STREAM`. Argument *s* is a socket descriptor created using `socket()`, bound to an address by `bind()`, and listening for connections after a `listen()`. `accept()` extracts the first connection on the queue of pending connections, creates a new socket with the same properties as *s*, and allocates a new file descriptor, *ns*, for the socket. If no pending connections are present on the queue and non-blocking mode has not been enabled using the `O_NONBLOCK` or `O_NDELAY` `fcntl()` flags or the `FIOSNBIO` `ioctl()` request, `accept()` blocks the caller until a connection is present. (`O_NONBLOCK` and `O_NDELAY` are defined in `<sys/fcntl.h>`; see *fcntl*(2) *fcntl*(5), and *socket*(7). `FIOSNBIO` and the equivalent request `FIONBIO` are defined in `<sys/ioctl.h>`, although use of `FIONBIO` is not recommended; see *ioctl*(2), *ioctl*(5), and *socket*(7).) If the socket is marked non-blocking and no pending connections are present on the queue, `accept()` returns an error as described below. The accepted socket, *ns*, cannot be used to accept more connections. The original socket *s* remains open. It is possible to determine whether a listening socket has pending connection requests ready for an `accept()` call by using `select()` for reading.

The argument *addr* should point to a local socket address structure. The `accept()` call fills in this structure with the address of the connecting entity, as known to the underlying protocol. The format of the address depends upon the protocol and the address-family of the socket *s*. *addrlen* is a pointer to an int; it should initially contain the size of the structure pointed to by *addr*. On return, it contains the actual length (in bytes) of the address returned. If the memory pointed to by *addr* is not large enough to contain the entire address, only the first *addrlen* bytes of the address are returned.

Since both the `fcntl()` `O_NONBLOCK` flag and `FIOSNBIO` `ioctl()` request are supported, some clarification on how these features interact is necessary. If the `O_NONBLOCK` flag has been set, `accept()` requests behave accordingly, regardless of any `FIOSNBIO` requests. If the `O_NONBLOCK` flag has not been set, `FIOSNBIO` requests control the behavior of `accept()`. **AF_CCITT only:** The *addr* parameter to `accept()` returns addressing information for the connecting entity, except for the `x25ifname[]` field of *addr* which contains the name of the local X.25 interface through which the connection request arrived. Call-acceptance can be controlled with the `X25_CALL_ACPT_APPROVAL` `ioctl()` call described in *socketx25*(7).

RETURN VALUE

Upon successful competion, `accept()` returns a non-negative integer which is a descriptor for the accepted socket. If an error occurs, `accept()` returns –1 and sets `errno` to indicate the cause.

DIAGNOSTICS

`accept()` fails if any of the following conditions are encountered:

[EBADF]	The file descriptor *s* is invalid.
[ENOTSOCK]	The file descriptor *s* references a file, not a socket.
[EOPNOTSUPP]	The socket referenced by *s* is not of type SOCK_STREAM.
[EFAULT]	The *addr* parameter is not in a valid pointer.
[EWOULDBLOCK]	Non-blocking I/O is enabled using `O_NDELAY` or `FIOSNBIO` and no connections are present to be accepted.
[EMFILE]	The maximum number of file descriptors for this process are already currently open.
[ENFILE]	The system's table of open files is full and no more `accepts` can be accepted at this time.

[ENOBUFS]	No buffer space is available. The **accept()** cannot complete. The queued socket connect request is aborted.
[EINVAL]	The socket referenced by *s* is not currently a listen socket or has been **shutdown()**. A **listen()** must be done before an **accept()** is allowed.
[EAGAIN]	Non-blocking I/O is enabled using **O_NONBLOCK** and no connections are present to be accepted.
[EINTR]	The call was interrupted by a signal before a valid connection arrived.

AUTHOR

accept() was developed by the University of California, Berkeley.

SEE ALSO

bind(2), connect(2), listen(2), select(2), socket(2) socketx25(7).

NAME

access - determine accessibility of a file

SYNOPSIS

```
#include <unistd.h>

int access(char *path, int amode);
```

DESCRIPTION

path points to a path name naming a file. `access()` checks the named file for accessibility according to the bit pattern contained in *amode*, using the real user ID instead of the effective user ID and the real group ID instead of the effective group ID. The value of *amode* is either the bit-wise inclusive OR of the access permissions to be checked or the existence test. The following symbolic constants, defined in `<unistd.h>`, test for permissions:

`R_OK`	read
`W_OK`	write
`X_OK`	execute (search)
`F_OK`	check existence of file

Access Control Lists (ACLs)

Read, write and execute/search permissions are checked against the file's access control list. Each mode is checked separately since different ACL entries might grant different permissions. The real user ID is combined with the process's real group ID and each group in its supplementary groups list, and the access control list is searched for a match. Search proceeds in order of specificity and ends when one or more matching entries are found at a specific level. More than one *u.g* or *%.g* entry can match a user if that user has a non-null supplementary groups list. If any matching entry has the appropriate permission bit set, access is permitted.

`access()` reports that a shared text file currently open for execution is not writable, regardless of its access control list. It also reports that a file on a read-only file system is not writable. However, `access()` does not report that a shared text file open for writing is not executable, since the check is not easily done.

RETURN VALUE

If the requested access is permitted, a value of `0` is returned. Otherwise, a value of `-1` is returned and `errno` is set to indicate the error.

ERRORS

Access to the file is denied if one or more of the following is true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	Read, write, or execute (search) permission is requested for a null path name.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.
[EROFS]	Write access is requested for a file on a read-only file system.
[ETXTBSY]	Write access is requested for a pure procedure (shared text) file that is being executed.
[EACCES]	The access control list does not permit the requested access and the real user ID is not a user with appropriate privileges.
[EFAULT]	*path* points outside the allocated address space for the process. The reliable detection of this error is implementation dependent.
[ELOOP]	Too many symbolic links were encountered in translating the path name.
[ENAMETOOLONG]	The length of the specified path name exceeds **PATH_MAX** bytes, or the length of a component of the path name exceeds **NAME_MAX** bytes while `_POSIX_NO_TRUNC` is in effect.

The owner of a file has permission checked with respect to the "owner" read, write, and execute mode bits. Members of the file's group other than the owner have permissions checked with respect to the "group"

mode bits, and all others have permissions checked with respect to the "other" mode bits.

`access()` reports that a file currently open for execution is not writable, regardless of the setting of its mode.

WARNINGS

If the path is valid and the real user ID is super-user, and the access requested is not **X_OK**, `access()` always returns 0. If **X_OK** access is requested for a valid path and the real user ID is super-user and the file is a directory, access always returns 0. If **X_OK** access is requested for a valid path which is not a directory and the real user ID is super-user, access returns 0 only if at least one execute bit (for user, group, or other) is set in the file's mode.

Access Control Lists

Access control list descriptions in this entry apply only to standard HP-UX operating systems. If HP-UX BLS software has been installed, access control lists are handled differently. Refer to HP-UX BLS documentation for information about access control lists in the HP-UX BLS environment.

SEE ALSO

chmod(2), setacl(2), stat(2), acl(5), unistd(5).

STANDARDS CONFORMANCE

`access()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

acct - enable or disable process accounting

SYNOPSIS

```
#include <sys/acct.h>

int acct(const char *path);
```

DESCRIPTION

`acct()` is used to enable or disable the system's process accounting routine. If the routine is enabled, an accounting record is written on an accounting file for each process that terminates. Termination can be caused by one of two things: an `exit()` call or a signal; see *exit*(2) and *signal*(5). The effective user ID of the calling process must be super-user to use this call.

path points to a path name naming the accounting file. The accounting file format is described in *acct*(4).

The accounting routine is enabled if *path* is non-zero and no errors occur during the system call. It is disabled if *path* is zero and no errors occur during the system call.

When the amount of free space on the file system containing the accounting file falls below a configurable threshold, the system prints a message on the console and disables process accounting. Another message is printed and the process accounting is re-enabled when the space reaches a second configurable threshold.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

ERRORS

`acct()` fails if one or more of the following is true:

[EPERM] The effective user ID of the calling process is not super-user.

[EBUSY] An attempt is being made to enable accounting when it is already enabled.

[ENOTDIR] A component of the path prefix is not a directory.

[ENOENT] One or more components of the accounting file path name do not exist.

[EACCES] The file named by *path* is not an ordinary file.

[EROFS] The named file resides on a read-only file system.

[EFAULT] *path* points to an illegal address. The reliable detection of this error simplementation dependent.

[ETXTBSY] *path* points to a text file which is currently open.

[ENAMETOOLONG] The accounting file path name exceeds **PATH_MAX** bytes, or the length of a component of the path name exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect.

[ELOOP] Too many symbolic links were encountered in translating the path name.

DEPENDENCIES

Series 300/400

The system's process accounting routine ignores any locks placed on the process accounting file.

If the size of the process accounting file reaches 5000 blocks, records for processes terminating after that point will be silently lost. However, in that case the `turnacct` command would still sense that process accounting is still enabled. This loss of records can be prevented by the use of `ckpacct` (see *acctsh*(1M)).

SEE ALSO

acct(1M), acctsh(1M), exit(2), acct(4), signal(5).

STANDARDS CONFORMANCE

`acct()`: SVID2, XPG2

NAME

alarm - set a process's alarm clock

SYNOPSIS

```
#include <unistd.h>

unsigned int alarm(unsigned int sec);
```

DESCRIPTION

alarm() instructs the alarm clock of the calling process to send the signal SIGALRM to the calling process after the number of real-time seconds specified by *sec* have elapsed; see *signal*(5). Specific implementations might place limitations on the maximum supported alarm time. The constant MAX_ALARM defined in **<sys/param.h>** specifies the implementation-specific maximum. Whenever *sec* is greater that this maximum, it is silently rounded down to it. On all implementations, **MAX_ALARM** is guaranteed to be at least 31 days (in seconds).

Alarm requests are not stacked; successive calls reset the alarm clock of the calling process.

If *sec* is 0, any previously made alarm request is canceled.

Alarms are not inherited by a child process across a **fork()**, but are inherited across an **exec()**.

On systems that support the **getitimer()** and **setitimer()** system calls, the timer mechanism used by **alarm()** is the same as that used by *ITIMER_REAL*. Thus successive calls to **alarm()**, **getitimer()**, and **setitimer()** set and return the state of a single timer. In addition, **alarm()** sets the timer interval to zero.

RETURN VALUE

alarm() returns the amount of time previously remaining in the alarm clock of the calling process.

WARNINGS

In some implementations, error bounds for alarm are -1, +0 seconds (for the posting of the alarm, not the restart of the process). Thus a delay of 1 second can return immediately. The **setitimer()** routine can be used to create a more precise delay.

SEE ALSO

sleep(1), exec(2), getitimer(2), pause(2), signal(5), sleep(3C).

STANDARDS CONFORMANCE

alarm(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

atexit - register a function to be called at program termination

SYNOPSIS

```
#include <stdlib.h>

int atexit(void (*func)(void));
```

DESCRIPTION

atexit() registers the function *func* to be called, without arguments, at normal program termination. Functions registered by **atexit()** are called in reverse order of registration.

An **atexit()** call during exit processing is always unsuccessful.

The number of registered functions should not exceed **ATEXIT_MAX** as specified in **<limits.h>**.

RETURN VALUE

atexit() returns zero if the registration is successful; non-zero if unsuccessful.

SEE ALSO

exit(2).

STANDARDS CONFORMANCE

atexit(): AES, XPG4, ANSI C

NAME

audctl - start or halt the auditing system and set or get audit files

SYNOPSIS

```
#include <sys/audit.h>

int audctl(int cmd, char *cpath, char *npath, mode_t mode);
```

DESCRIPTION

`audctl()` sets or gets the auditing system "current" and "next" audit files, and starts or halts the auditing system. This call is restricted to superusers. *cpath* and *npath* hold the absolute path names of the "current" and "next" files. *mode* specifies the audit file's permission bits. *cmd* is one of the following specifications:

AUD_ON The caller issues the **AUD_ON** command with the required "current" and "next" files to turn on the auditing system. If the auditing system is currently off, it is turned on; the file specified by the *cpath* parameter is used as the "current" audit file, and the file specified by the *npath* parameter is used as the "next" audit file. If the audit files do not already exist, they are created with the *mode* specified. The auditing system then begins writing to the specified "current" file. An empty string or NULL *npath* can be specified if the caller wants to designate that no "next" file be available to the auditing system. If the auditing system is already on, no action is performed; `-1` is returned and `errno` is set to EBUSY.

AUD_GET The caller issues the **AUD_GET** command to retrieve the names of the "current" and "next" audit files. If the auditing system is on, the names of the "current" and "next" audit files are returned via the *cpath* and *npath* parameters (which must point to character buffers of sufficient size to hold the file names). *mode* is ignored. If the auditing system is on and there is no available "next" file, the "current" audit file name is returned via the *cpath* parameter, *npath* is set to an empty string; `-1` is returned, and `errno` is set to ENOENT. If the auditing system is off, no action is performed; `-1` is returned and `errno` is set to EALREADY.

AUD_SET The caller issues the **AUD_SET** command to change both the "current" and "next" files. If the audit system is on, the file specified by *cpath* is used as the "current" audit file, and the file specified by *npath* is used as the "next" audit file. If the audit files do not already exist, they are created with the specified *mode*. The auditing system begins writing to the specified "current" file. Either an empty string or NULL *npath* can be specified if the caller wants to designate that no "next" file be available to the auditing system. If the auditing system is off, no action is performed; `-1` is returned and `errno` is set to EALREADY.

AUD_SETCURR The caller issues the **AUD_SETCURR** command to change only the "current" audit file. If the audit system is on, the file specified by *cpath* is used as the "current" audit file. If the specified "current" audit file does not exist, it is created with the specified *mode*. *npath* is ignored. The auditing system begins writing to the specified "current" file. If the audit system is off, no action is performed; `-1` is returned and `errno` is set to EALREADY.

AUD_SETNEXT The caller issues the **AUD_SETNEXT** command to change only the "next" audit file. If the auditing system is on, the file specified by *npath* is used as the "next" audit file. *cpath* is ignored. If the "next" audit file specified does not exist, it is created with the specified *mode*. Either an empty string or NULL *npath* can be specified if the caller wants to designate that no "next" file be available to the auditing system. If the auditing system is off, no action is performed; `-1` is returned, and `errno` is set to EALREADY.

AUD_SWITCH The caller issues the **AUD_SWITCH** command to cause auditing system to switch audit files. If the auditing system is on, it uses the "next" file as the new "current" audit file and sets the new "next" audit file to NULL. *cpath*, *npath*,*and mode* are ignored. The auditing system begins writing to the new "current" file. If the auditing system is off, no action is performed; `-1` is returned, and `errno` is set to EALREADY. If the auditing system is on and there is no

available "next" file, no action is performed; `-1` is returned, and `errno` is set to ENOENT.

AUD_OFF The caller issues the **AUD_OFF** command to halt the auditing system. If the auditing system is on, it is turned off and the "current" and "next" audit files are closed. *cpath*, *npath*, and *mode* are ignored. If the audit system is already off, `-1` is returned and `errno` is set to EALREADY.

RETURN VALUE

Upon successful completion, a value of `0` is returned. Otherwise, `-1` is returned and the global variable `errno` is set to indicate the error.

EXAMPLES

In the following example, `audctl()` is used to determine whether the auditing system is on, and to retrieve the names of the audit files that are currently in use by the system.

```
char c_file[PATH_MAX+1], x_file[PATH_MAX+1];
int  mode=0600;

   if (audctl(AUD_GET, c_file, x_file, mode))
      switch ( errno ) {
         case ENOENT:
         strcpy(x_file,"-none-");
         break;
      case EALREADY:
         printf("The auditing system is OFF\n");
         return 0;
      case default:
         fprintf(stderr, "Audctl failed: errno=%d\n", errno);
         return 1;
   }
   printf("The auditing system is ON: c_file=%s x_file=%s\n", c_file, x_file
   return 0;
```

ERRORS

`audctl()` fails if one of the following is true:

[EPERM] The caller does not have superuser privilege, or one or both of the given files are not regular files and cannot be used.

[EALREADY] The **AUD_OFF**, **AUD_SET**, **AUD_SETCURR**, **AUD_SETNEXT**, **AUD_SWITCH**, or **AUD_GET** *cmd* was specified while the auditing system is off.

[EBUSY] User attempt to start the auditing system failed because auditing is already on.

[EFAULT] Bad pointer. One or more of the required function parameters is not accessible.

[EINVAL] The *cpath* or *npath* is greater than **PATH_MAX** in length, the *cpath* or *npath* specified is not an absolute path name.

[ENOENT] No available "next" file when *cmd* is **AUD_GETNEXT** or **AUD_SWITCH**.

AUTHOR

`audctl()` was developed by HP.

SEE ALSO

audit(5), audsys(1M), audomon(1M).

NAME

audswitch - suspend or resume auditing on the current process

SYNOPSIS

```
#include <sys/audit.h>

int audswitch(int aflag);
```

DESCRIPTION

audswitch() suspends or resumes auditing within the current process. This call is restricted to superusers.

One of the following *aflags* must be used:

AUD_SUSPEND	Suspend auditing on the current process.
AUD_RESUME	Resume auditing on the current process.

audswitch() can be used in self-auditing privileged processes to temporarily suspend auditing during intervals where auditing is to be handled by the process itself. Auditing is suspended by a call to **audswitch()** with the **AUD_SUSPEND** parameter and resumed later by a call to **audswitch()** with the **AUD_RESUME** parameter.

An **audswitch()** call to resume auditing serves only to reverse the action of a previous **audswitch()** call to suspend auditing. A call to **audswitch()** to resume auditing when auditing is not suspended has no effect.

audswitch() affects only the current process. For example, **audswitch()** cannot suspend auditing for processes **exec**'ed from the current process. (Use **setaudproc** (see *setaudproc*(2)) to enable or disable auditing for a process and its children).

RETURN VALUE

Upon successful completion, **audswitch()** returns **0**. If an error occurs, **-1** is returned and the global variable **errno** is set to indicate the error.

ERRORS

audswitch() fails if one of the following is true:

[EPERM]	The user is not a superuser.
[EINVAL]	The input parameter is neither **AUD_RESUME** nor **AUD_SUSPEND**.

AUTHOR

audswitch() was developed by HP.

SEE ALSO

audit(5), setaudproc(2), audusr(1M), audevent(1M).

NAME

audwrite - write an audit record for a self-auditing process

SYNOPSIS

```
#include <sys/audit.h>

int audwrite(const struct self_audit_rec *audrec_p);
```

DESCRIPTION

`audwrite()` is called by trusted self-auditing processes, which are capable of turning off the regular auditing (using *audswitch*(2)) and doing higher-level auditing on their own. `audwrite()` is restricted to superusers.

`audwrite()` checks to see if the auditing system is on and the calling process and the event specified are being audited. If these conditions are met, `audwrite()` writes the audit record pointed to by *audrec_p* into the audit file. The record consists of an audit record body and a header with the following fields:

```
u_long ah_time;     /* Date/time (tv_sec of timeval) */
u_short ah_pid;     /* Process ID */
u_short ah_error;   /* Success/failure */
u_short ah_event;   /* Event being audited */
u_short ah_len;     /* Length of variant part */
```

The header has the same format as the regular audit record, while the body contains additional information about the high-level audit event. The header fields `ah_error`, `ah_event`, and `ah_len` are specified by the calling process. `audwrite()` fills in `ah_time` and `ah_pid` fields with the correct values. this is done to reduce the risk of forgery. After the header is completed, the record body is attached and the entire record is written into the current audit file.

RETURN VALUE

If the write is successful, a value of `0` is returned. Otherwise, a value of `-1` is returned and `errno` is set to indicate the reason for the failure.

ERRORS

`audwrite()` fails if one of the following is true:

[EPERM] The caller is not a superuser.

[EINVAL] The event number in the audit record is invalid.

WARNINGS

If *audwrite* causes a file space overflow, the calling process might be suspended until the file space is cleaned up. However a returned call with the return value of 0 indicates that the audit record has been successfully written.

AUTHOR

`audwrite()` was developed by HP.

SEE ALSO

audswitch(2), audit(4).

NAME

bind - bind an address to a socket

SYNOPSIS

```
#include <sys/socket.h>
```

AF_INET only:

```
#include <netinet/in.h>
```

AF_UNIX only:

```
#include <sys/un.h>
```

AF_CCITT only:

```
#include <x25/x25addrstr.h>

int bind(int s, const void *addr, int addrlen);
```

DESCRIPTION

`bind()` assigns an address to an unbound socket. When a socket is created with `socket()`, it exists in an address space (address family) but has no address assigned. `bind()` causes the socket whose descriptor is *s* to become bound to the address specified in the socket address structure pointed to by *addr*.

addrlen must specify the size of the address structure. Since the size of the socket address structure varies between socket address families, the correct socket address structure should be used with each address family (for example, `struct sockaddr_in` for **AF_INET**, and `struct sockaddr_un` for **AF_UNIX**). Typically, the `sizeof()` function is used to pass this value in the `bind()` call (for example, `sizeof(struct sockaddr_in)`).

The rules used in address binding vary between communication domains. For example, when binding an **AF_UNIX** socket to a path name (such as `/tmp/mysocket`), an open file having that name is created in the file system. When the bound socket is closed, that file still exists unless it is removed or unlinked. When binding an **AF_INET** socket, *sin_port* can be a port number, or it can be zero. If *sin_port* is zero, the system assigns an unused port number automatically.

RETURN VALUE

Upon successful completion, `bind()` returns 0; otherwise it returns −1 and sets `errno` to indicate the error.

DIAGNOSTICS

`bind()` fails if any of the following conditions are encountered:

[EBADF]	*s* is not a valid descriptor.
[ENOTSOCK]	*s* is not a socket.
[EADDRNOTAVAIL]	The specified address is bad or not available from the local machine, or for **AF_CCITT** sockets which use "wild card" addressing, the specified address space overlays the address space of an existing bind.
[EADDRINUSE]	The specified address is already in use.
[EINVAL]	The socket is already bound to an address, the socket has been shut down, *addrlen* is a bad value, or an attempt was made to `bind()` an **AF_UNIX** socket to an NFS-mounted (remote) name. **AF_CCITT:** The protocol-ID length is negative or greater than 8, or the X.121 address string contains an illegal character, or the X.121 address string is greater than 15 digits long.
[EAFNOSUPPORT]	Requested address does not match the address family of this socket.
[EACCES]	The requested address is protected, and the current user has inadequate permission to access it. (This error can be returned by **AF_INET** only.)
[EFAULT]	*addr* is not a valid pointer.
[EOPNOTSUPP]	The socket whose descriptor is *s* is of a type that does not support address binding.

[ENOBUFS] Insufficient buffer memory is available. The **bind()** cannot complete.

[ENETUNREACH] The X.25 Level 2 protocol is down. The X.25 link is not working: wires might be broken, or connections are loose on the interface hoods at the modem, or the modem failed, or noise interfered with the line for an extremely long period of time.

[EDESTADDREQ] No *addr* parameter was specified.

[ENODEV] The *x25ifname* field name specifies a non-existent interface. (This error can be returned by **AF_CCITT** only.)

[ENETDOWN] The *x25ifname* field name specifies an interface that was shut down, or never initialized, or whose Level 2 protocol indicates that the link is not working: wires might be broken, the interface hoods on the modem are broken, the modem failed, the phone connection failed (this error can be returned by **AF_CCITT** only), noise interfered with the line for a long period of time.

AUTHOR

bind() was developed by the University of California, Berkeley)

SEE ALSO

connect(2), getsockname(2), listen(2), socket(2), af_ccitt(7F), inet(7F), socketx25(7), tcp(7P), udp(7P), unix(7P).

NAME

brk, sbrk - change data segment space allocation

SYNOPSIS

```
#include <unistd.h>

int brk(const void *endds);

void *sbrk(int incr);
```

DESCRIPTION

`brk()` and `sbrk()` are used to change dynamically the amount of space allocated for the calling process's data segment; see *exec*(2). The change is made by resetting the process's break value and allocating the appropriate amount of space. The break value is the address of the first location beyond the end of the data segment. The amount of allocated space increases as the break value increases. The newly allocated space is set to zero.

`brk()` sets the break value to *endds* and changes the allocated space accordingly.

`sbrk()` adds *incr* bytes to the break value and changes the allocated space accordingly. *incr* can be negative, in which case the amount of allocated space is decreased.

ERRORS

`brk()` and `sbrk()` fail without making any change in the allocated space if one or more of the following are true:

[ENOMEM] Such a change would result in more space being allocated than is allowed by a system-imposed maximum (see *ulimit*(2)).

[ENOMEM] Such a change would cause a conflict between addresses in the data segment and any attached shared memory segment (see *shmop*(2)).

[ENOMEM] Such a change would be impossible as there is insufficient swap space available.

WARNINGS

The pointer returned by `sbrk()` is not necessarily word-aligned. Loading or storing words through this pointer could cause word alignment problems.

Be very careful when using either `brk` or `sbrk` in conjunction with calls to the *malloc*(3C) library routines. There is only one program data segment from which all three of these routines allocate and deallocate program data memory.

RETURN VALUE

Upon successful completion, `brk()` returns a value of 0 and `sbrk()` returns the old break value. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

AUTHOR

`brk()` and `sbrk()` were developed by AT&T and HP.

SEE ALSO

exec(2), shmop(2), ulimit(2), end(3C), malloc(3C).

STANDARDS CONFORMANCE

`brk()`: XPG2

`sbrk()`: XPG2

NAME

killpg, getpgrp, setpgrp, sigvec, signal - 4.2 BSD-compatible process control facilities

SYNOPSIS

```
#include <signal.h>

int killpg(int pgrp, int sig);

int getpgrp(int pid);

int setpgrp(int pid, int pgrp);

int sigvec(
    int sig,
    struct sigvec *vec,
    struct sigvec *ovec
);

void (*signal(int sig, void (*func)(int)))(int);
```

DESCRIPTION

These calls simulate (and are provided for backward compatibility with) functions of the same name in the 4.2 Berkeley Software Distribution.

This version of **setpgrp()** is equivalent to the system call *setpgid*(*pid*, *pgrp*) (see *setpgid*(2)).

This version of **getpgrp()** is equivalent to the system call **getpgrp2**(*pid*) (see *getpid*(2)).

killpg() is equivalent to the system call **kill**(*-pgrp*, *sig*) (see *kill*(2)).

sigvec() is equivalent to the system call **sigvector**(*sig*, *vec*, *ovec*) (see *sigvector*(2)), except for the following:

When SIGCHLD or SIGCLD is used and *vec* specifies a catching function, the routine acts as if the **SV_BSDSIG** flag were included in the **sv_flags** field of *vec*.

The name **sv_onstack** can be used as a synonym for the name of the **sv_flags** field of *vec* and *ovec*.

If *vec* is not a null pointer and the value of (*vec->sv_flags* & 1) is "true", the routine acts as if the **SV_ONSTACK** flag were set.

If *ovec* is not a null pointer, the flag word returned in *ovec->sv_flags* (and therefore the value of *ovec->sv_onstack*) will be equal to 1 if the system was reserving space for processing of that signal because of a call to *sigspace*(2), and 0 if not. The **SV_BSDSIG** bit in the value placed in *ovec->sv_flags* is always clear.

If the reception of a caught signal occurs during certain system calls, the call will always be restarted, regardless of the return value from a catching function installed with **sigvec()**. The affected calls are *wait*(2), *semop*(2), *msgsnd*(2), *msgrcv*(2), and *read*(2) or *write*(2) on a slow device (such as a terminal or pipe, but not a file). Other interrupted system calls are not restarted.

This version of **signal()** has the same effect as **sigvec(sig**, *vec*, *ovec*), where *vec->sv_handler* is equal to *func*, *vec->sv_mask* is equal to 0, and *vec->sv_flags* is equal to 0. **signal()** returns the value that would be stored in *ovec->sv_handler* if the equivalent **sigvec()** call would have succeeded. Otherwise, **signal()** returns -1 and **errno** is set to indicate the reason as it would have been set by the equivalent call to **sigvec()**.

These functions can be linked into a program by giving the **-lBSD** option to *ld*(1).

WARNINGS

While the 4.3 BSD release defined extensions to some of the interfaces described here, only the 4.2 BSD interfaces are emulated by this package.

bsdproc() should not be used in conjunction with the facilities described under *sigset*(2V).

AUTHOR

bsdproc() was developed by HP and the University of California, Berkeley.

SEE ALSO

ld(1), kill(2), getpid(2), msgsnd(2), msgrcv(2), read(2), semop(2), setpgid(2), setsid(2), sigvector(2), wait(2), write(2), sigset(2V), sigstack(2), signal(5).

NAME

chdir, fchdir - change working directory

SYNOPSIS

```
#include <unistd.h>

int chdir(const char *path);

int fchdir(int fildes);
```

DESCRIPTION

`chdir()` and `fchdir()` cause a directory pointed to by *path* or *fildes* to become the current working directory, the starting point for path searches of path names not beginning with `/`. *path* points to the path name of a directory. *fildes* is an open file descriptor of a directory.

For a directory to become the current working directory, a process must have execute (search) access to the directory.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

ERRORS

`chdir()` fails and the current working directory remains unchanged if one or more of the following are true:

[ENOTDIR]	A component of the path name is not a directory.
[ENOENT]	The named directory does not exist.
[EACCES]	Search permission is denied for any component of the path name.
[EFAULT]	*path* points outside the allocated address space of the process. The reliable detection of this error is implementation dependent.
[ENOENT]	*path* is null.
[ENAMETOOLONG]	The length of the specified path name exceeds **PATH_MAX** bytes, or the length of a component of the path name exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect.
[ELOOP]	Too many symbolic links were encountered in translating the path name.

`fchdir()` fails and the current working directory remains unchanged if one or more of the following are true:

[EACCES]	Search permission is denied for *fildes*.
[EBADF]	*fildes* is not an open file descriptor.
[ENOTDIR]	The open file descriptor *fildes* does not refer to a directory.

AUTHOR

`chdir()` and `fchdir()` were developed by AT&T Bell Laboratories and HP.

SEE ALSO

cd(1), chroot(2).

STANDARDS CONFORMANCE

`chdir()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

chmod, fchmod - change access mode of file

SYNOPSIS

```
#include <sys/stat.h>

int chmod(const char *path, mode_t mode);

int fchmod(int fildes, mode_t mode);
```

DESCRIPTION

`chmod()` and `fchmod()` set the access permission portion of the file's mode according to the bit pattern contained in *mode*. *path* points to a path name naming a file. *fildes* is a file descriptor.

The following symbolic constants representing the access permission bits are defined with the indicated values in `<sys/stat.h>` and are used to construct the *mode* argument. The value of *mode* is the bit-wise inclusive OR of the values for the desired permissions.

`S_ISUID`	`04000`	Set user ID on execution.
`S_ISGID`	`02000`	Set group ID on execution.
`S_ENFMT`	`02000`	Record locking enforced.
`S_ISVTX`	`01000`	Save text image after execution.
`S_IRUSR`	`00400`	Read by owner.
`S_IWUSR`	`00200`	Write by owner.
`S_IXUSR`	`00100`	Execute (search) by owner.
`S_IRGRP`	`00040`	Read by group.
`S_IWGRP`	`00020`	Write by group.
`S_IXGRP`	`00010`	Execute (search) by group.
`S_IROTH`	`00004`	Read by others (that is, anybody else).
`S_IWOTH`	`00002`	Write by others.
`S_IXOTH`	`00001`	Execute (search) by others.

To change the mode of a file, the effective user ID of the process must match that of the owner of the file or a user with appropriate privileges.

If the effective user ID of the process is not that of a user with appropriate privileges and the file is a regular file, `S_ISVTX` (mode bit 01000, save text image on execution) is cleared.

If the effective user ID of the process is not that of a user with appropriate privileges, and the effective group ID of the process does not match the group ID of the file and none of the group ID s in the supplementary groups list match the group ID of the file, `S_ISGIDR`, `S_ENFMT` (mode bit 02000, set group ID on execution and enforced file locking mode) is cleared.

The set-group-ID on execution bit is also used to enforce file-locking mode (see *lockf*(2) and *fcntl*(2)) on files that are not group executable. This might affect future calls to `open()`, `creat()`, `read()`, and `write()` on such files (see *open*(2), *creat*(2), *read*(2), and *write*(2)).

If an executable file is prepared for sharing, `S_ISVTX` (mode bit 01000) prevents the system from abandoning the swap-space image of the program-text portion of the file when its last user terminates. Then, when the next user of the file executes it, the text need not be read from the file system but can simply be swapped in, thus saving time.

If mode `S_ISVTX` (mode bit 01000) is set on a directory, an unprivileged user cannot delete or rename others' files in that directory.

Access Control Lists

All optional entries in a file's access control list are deleted when `chmod()` is executed. (This behavior conforms to the IEEE Standard POSIX 1003.1-1988.) To preserve optional entries in a file's access control list, it is necessary to save and restore them using `getacl()` and `setacl()` (see *getacl*(2) and *setacl*(2)).

To set the permission bits of access control list entries, use `setacl()` instead of `chmod()`.

For more information on access control list entries, see *acl*(5).

RETURN VALUE

Upon successful completion, a value of `0` is returned. Otherwise, a value of `-1` is returned and `errno` is

set to indicate the error.

ERRORS

`chmod()` and `fchmod()` fail and the file mode is unchanged if one or more of the following is true:

[EACCES] Search permission is denied on a component of the path prefix.

[EFAULT] *path* points outside the allocated address space of the process. The reliable detection of this error is implementation dependent.

[ELOOP] Too many symbolic links are encountered in translating *path*.

[ENAMETOOLONG] A component of *path* exceeds `NAME_MAX` bytes while `_POSIX_NO_TRUNC` is in effect or *path* exceeds `PATH_MAX` bytes.

[ENOENT] A component of *path* does not exist.

[ENOENT] The file named by *path* does not exist.

[ENOTDIR] A component of the path prefix is not a directory.

[EPERM] The effective user ID does not match that of the owner of the file, and the effective user ID is not that of a user with appropriate privileges.

[EROFS] The named file resides on a read-only file system.

[EINVAL] Attempted to make a root directory into a context-dependent file (see DEPENDENCIES).

WARNINGS

Access Control Lists

Access control list descriptions in this entry apply only to standard HP-UX operating systems. If HP-UX BLS software has been installed, access control lists are handled differently. Refer to HP-UX BLS documentation for information about access control lists in the HP-UX BLS environment.

DEPENDENCIES

HP Clustered Environment:

If the file is a directory, the access permission bit `S_CDF` (04000) indicates a hidden directory (context-dependent file - see *cdf*(4)). A root directory cannot be made into a context-dependent file.

NFS `fchmod()` is not supported on remote files.

AUTHOR

`chmod()` was developed by AT&T, the University of California, Berkeley, and HP.

`fchmod()` was developed by the University of California, Berkeley.

SEE ALSO

chmod(1), chown(2), creat(2), fcntl(2), read(2), lockf(2), mknod(2), open(2), getacl(2), setacl(2), write(2), cdf(4), acl(5).

STANDARDS CONFORMANCE

`chmod()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

`fchmod()`: AES

NAME

chown, fchown - change owner and group of a file

SYNOPSIS

```
#include <unistd.h>

int chown(const char *path, uid_t owner, gid_t group);

int fchown(int fildes, uid_t owner, gid_t group);
```

DESCRIPTION

`chown()` changes the user and group ownership of a file. *path* points to a path name naming a file. *fildes* is a file descriptor. `chown()` and `fchown()` set the owner ID and group ID of the file to the numeric values contained in *owner* and *group* respectively. A value of `UID_NO_CHANGE` or `GID_NO_CHANGE` can be specified in *owner* or *group* to leave unchanged the file's owner ID or group ID respectively. Note that *owner* and *group* should be less than `UID_MAX` (see *limits*(5)).

Only processes with effective user ID equal to the file owner or a user having appropriate privileges can change the ownership of a file. If privilege groups are supported, the owner of a file can change the ownership only if he is a member of a privilege group allowing CHOWN, as set up by the `setprivgrp` command (see *setprivgrp*(1M)). All users get CHOWN privileges by default.

The group ownership of a file can be changed to any group in the current process's access list or to the real or effective group ID of the current process. If privilege groups are supported and the user is permitted the CHOWN privilege, the file can be given to any group.

If `chown()` is invoked on a regular file by other than the super-user the set-user-ID and set-group-ID bits of the file mode are cleared. Whether `chown()` preserves or clears these bits on files of other types is implementation dependent.

Access Control Lists (ACLs)

A user can allow or deny specific individuals and groups access to a file by using the file's access control list (see *acl*(5)). When using `chown()` in conjunction with ACLs, if the new owner and/or group does not have an optional ACL entry corresponding to *u*`.%` and/or `%.`*g* in the file's access control list, the file's access permission bits remain unchanged. However, if the new owner and/or group is already designated by an optional ACL entry of *u*`.%` and/or `%.`*g*, `chown()` sets the file's permission bits (and the three basic ACL entries) to the permissions contained in that entry.

ERRORS

`chown()` fails and the owner and group of the file remain unchanged if one or more of the following is true:

[EBADF] *fildes* is not a valid file descriptor.

[ENOTDIR] A component of the path prefix is not a directory.

[ENOENT] The file named by *path* does not exist.

[EACCES] Search permission is denied on a component of the path prefix.

[EPERM] The effective user ID is not a user having appropriate privileges and one or more of the following conditions exist:

The effective user ID does not match the owner of the file.

When changing the owner of the file, the owner of the file is not a member of a privilege group allowing the CHOWN privilege.

When changing the group of the file, the owner of the file is not a member of a privilege group allowing the CHOWN privilege and the group number is not in the current process's access list.

[EROFS] The named file resides on a read-only file system.

[EFAULT] *path* points outside the allocated address space of the process. The reliable detection of this error will be implementation dependent.

[ENAMETOOLONG] A component of *path* exceeds `NAME_MAX` bytes while `_POSIX_NO_TRUNC` is in effect, or *path* exceeds `PATH_MAX` bytes.

[ELOOP] Too many symbolic links were encountered in translating *path*.

[EINVAL] Either *owner* or *group* is greater than or equal to UID_MAX.

DEPENDENCIES

Series 300, 400, and 700:

If the path given to `chown()` contains a symbolic link as the last element, this link is traversed and pathname resolution continues. `chown()` changes the owner and group of the symbolic link's target, rather than the owner and group of the link.

HP Clustered Environment:

`chown()` does not clear the set-user-ID bit of a directory because that bit indicates that the directory is hidden (see *cdf*(4)).

When `chown()` is called from a cluster client node, the privilege groups checked are the ones set up on the cluster server.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and `errno` is set to indicate the error.

WARNINGS

Access Control Lists

Access control list descriptions in this entry apply only to standard HP-UX operating systems. If HP-UX BLS software has been installed, access control lists are handled differently. Refer to HP-UX BLS documentation for information about access control lists in the HP-UX BLS environment.

AUTHOR

`chown()` was developed by AT&T.
`fchown()` was developed by the University of California, Berkeley.

SEE ALSO

chown(1), setprivgrp(1M), chmod(2), setacl(2), acl(5), limits(5), limits(5).

STANDARDS CONFORMANCE

`chown()`: AES [Series 300/400/700 only], SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

`fchown()`: AES

NAME

chroot - change root directory

SYNOPSIS

```
#include <unistd.h>

int chroot(const char *path);
```

DESCRIPTION

`chroot()` causes the named directory to become the root directory, the starting point for path searches for path names beginning with `/`. *path* points to a path name naming a directory. The user's working directory is unaffected by the `chroot()` system call.

The effective user ID of the process must be a user having appropriate privileges to change the root directory.

The `..` entry in the root directory is interpreted to mean the root directory itself. Thus, `..` cannot be used to access files outside the subtree rooted at the root directory.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

ERRORS

`chroot()` fails and the root directory remains unchanged if one or more of the following is true:

[ENOTDIR]	Any component of the path name is not a directory.
[ENOENT]	The named directory does not exist or a component of the *path* does not exist.
[EPERM]	The effective user ID is not a user who has appropriate privileges.
[EFAULT]	*path* points outside the allocated address space of the process. The reliable detection of this error is implementation dependent.
[ENAMETOOLONG]	The length of the specified path name exceeds **PATH_MAX** bytes, or the length of a component of the path name exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect.
[ELOOP]	Too many symbolic links were encountered in translating the path name.

SEE ALSO

chroot(1M), chdir(2).

STANDARDS CONFORMANCE

`chroot()`: AES, SVID2, XPG2, XPG3, XPG4

NAME

close - close a file descriptor

SYNOPSIS

```
#include <unistd.h>

int close(int fildes);
```

DESCRIPTION

`close()` closes the file descriptor indicated by *fildes*. *fildes* is a file descriptor obtained from a `creat()`, `open()`, `dup()`, `fcntl()`, or `pipe()` system call. All associated file segments which have been locked by this process with the `lockf()` function are released (i.e., unlocked).

RETURN VALUE

Upon successful completion, `close()` returns a value of 0; otherwise, it returns –1 and sets `errno` to indicate the error.

ERRORS

`close()` fails if the any of following conditions are encountered:

[EBADF] *fildes* is not a valid open file descriptor.

[EINTR] An attempt to close a slow device or connection was interrupted by a signal. The file descriptor still points to an open device or connection.

[ENOSPC] Not enough space on the file system. This error can occur when closing a file on an NFS file system. [When a `write()` system call is executed on a local file system and if a new buffer needs to be allocated to hold the data, the buffer is mapped onto the disk at that time. A full disk is detected at this time and `write()` returns an error. When the `write()` system call is executed on an NFS file system, the new buffer is allocated without communicating with the NFS server to see if there is space for the buffer (to improve NFS performance). It is only when the buffer is written to the server (at file close or the buffer is full) that the disk-full condition is detected.]

SEE ALSO

creat(2), dup(2), exec(2), fcntl(2), lockf(2), open(2), pipe(2).

STANDARDS CONFORMANCE

`close()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

cnodeid - get the cnode ID of the local machine

SYNOPSIS

```
#include <cluster.h>

cnode_t cnodeid(void);
```

DESCRIPTION

`cnodeid()` returns the cnode ID of the local machine.

SEE ALSO

cnodes(1), cnodes(2), getcontext(2), getccent(3C).

AUTHOR

`cnodeid` was developed by HP.

NAME

cnodes - get a list of active nodes in cluster

SYNOPSIS

```
#include <cluster.h>

int cnodes(cnode_t *buf);
```

DESCRIPTION

`cnodes()` determines the number of active cnodes in the cluster, including the cnode on which it is invoked. If *buf* is not a null pointer, it must point to an array of type `cnode_t` with at least `MAX_CNODE` elements. In this case, the values of the cnode IDs of the nodes currently in the cluster are stored in the array, terminated by the cnode ID `0`.

RETURN VALUE

Upon successful completion, `cnodes()` returns the current number of active cnodes. If the value `0` is returned, the machine is not a member of a cluster. In case of an error, a value of `-1` is returned and `errno` is set to indicate the error.

ERRORS

`cnodes()` may fail if:

[EFAULT] *buf* is not a null pointer and points to an illegal address. Reliable detection of this error is not guaranteed.

SEE ALSO

cnodes(1), cnodeid(2), getcontext(2), getccent(3C).

AUTHOR

cnodes was developed by HP.

NAME

connect - initiate a connection on a socket

SYNOPSIS

```
#include <sys/socket.h>
```

AF_INET only:

```
#include <netinet/in.h>
```

AF_UNIX only:

```
#include <sys/un.h>
```

AF_CCITT only:

```
#include <x25/x25addrstr.h>

int connect(int s, const void *addr, int addrlen);
```

DESCRIPTION

`connect()` initiates a connection on a socket.

The parameter *s* is a socket descriptor. *addr* is a pointer to a socket address structure containing the address of a remote socket to which a connection is to be established. *addrlen* is the size of this address structure. Since the size of the socket address structure varies among socket address families, the correct socket address structure should be used with each address family (for example, `struct sockaddr_in` for **AF_INET**, and `struct sockaddr_un` for **AF_UNIX**). Typically, the `sizeof()` function is used to pass this value (for example, `sizeof(struct sockaddr_in)`).

If the socket is of type **SOCK_DGRAM**, `connect()` specifies the peer address to which messages are to be sent, and the call returns immediately. Furthermore, this socket can only receive messages sent from this address.

If the socket is of type **SOCK_STREAM**, `connect()` attempts to contact the remote host in order to make a connection between the remote socket (peer) and the local socket specified by *s*. The call normally blocks until the connection completes. If non-blocking mode has been enabled using the **O_NONBLOCK** or **O_NDELAY** `fcntl()` flags or the **FIOSNBIO** `ioctl()` request and the connection cannot be completed immediately, `connect()` returns an error as described below. In these cases, `select()` can be used on this socket to determine when the connection has completed by selecting it for writing.

O_NONBLOCK and **O_NDELAY** are defined in <**sys/fcntl.h**> and explained in *fcntl*(2), *fcntl*(5), and *socket*(7). **FIOSNBIO** is defined in <**sys/ioctl.h**> and explained in *ioctl*(2), *ioctl*(5), and *socket*(7).

If *s* is a **SOCK_STREAM** socket that is bound to the same local address as another **SOCK_STREAM** socket, `connect()` returns EADDRINUSE if *addr* is the same as the peer address of that other socket. This situation can only happen if the **SO_REUSEADDR** option has been set on an **AF_INET** socket (see *getsockopt*(2)).

If the AF_INET socket does not already have a local name bound to it (see *bind*(2)), `connect()` also binds the socket to a local address chosen by the system.

Generally, stream sockets may successfully connect only once; datagram sockets may use `connect()` multiple times to change the peer address. For datagram sockets, a side effect of attempting to connect to some invalid address (see DIAGNOSTICS below) is that the peer address is no longer maintained by the system. An example of an invalid address for a datagram socket is *addrlen* set to 0 and *addr* set to any value.

AF_CCITT only:

Use the `x25addrstr` struct for the address structure. The caller must know the X.121 address of the DTE to which the connection is to be established, including any sub-addresses or protocol-IDs that may be needed. Refer to *af_ccitt*(7F) for a detailed description of the `x25addrstr` address structure. If address-matching by protocol-ID, specify the protocol-ID with the **X25_WR_USER_DATA** `ioctl()` call before issuing the `connect()` call. The **X25_WR_USER_DATA** `ioctl()` call is described in *socketx25*(7).

DEPENDENCIES

AF_CCITT:

The **SO_REUSEADDR** option to `setsockopt()` is not supported for sockets in the AF_CCITT address family.

RETURN VALUE

Upon successful completion, `connect()` returns 0; otherwise it returns -1 and sets `errno` to indicate the error.

DIAGNOSTICS

`connect()` fails if any of the following conditions are encountered:

[EBADF]	*s* is not a valid file descriptor.
[ENOTSOCK]	*s* is a file descriptor for a file, not a socket.
[EADDRNOTAVAIL]	The specified address is not available on this machine, or the socket is a TCP or UDP socket and the zero port number is specified. For datagram sockets, the peer address is no longer maintained by the system.
[EAFNOSUPPORT]	Addresses in the specified address family cannot be used with this socket. For datagram sockets, the peer address is no longer maintained by the system.
[EALREADY]	Non-blocking I/O is enabled using `O_NONBLOCK`, `O_NDELAY`, or `FIOSNBIO`, and a previous connection attempt has not yet completed.
[EISCONN]	The socket is already connected.
[EINVAL]	The socket has already been shut down, or has a `listen()` active on it; *addrlen* is a bad value; an attempt was made to `connect()` an AF_UNIX socket to an NFS-mounted (remote) name; the X.121 address length is zero, negative, or greater than fifteen digits. For datagram sockets, if *addrlen* is a bad value, the peer address is no longer maintained by the system.
[ETIMEDOUT]	Connection establishment timed out without establishing a connection. *backlog* may be full (see *listen*(2)).
[ECONNREFUSED]	The attempt to connect was forcefully rejected.
[ENETUNREACH]	The network is not reachable from this host. **For AF_CCITT only:** X.25 Level 2 is down. The X.25 link is not working: wires might be broken, or connections are loose on the interface hoods at the modem, or the modem failed, or noise interfered with the line for an extremely long period of time.
[EADDRINUSE]	The address is already in use. For datagram sockets, the peer address is no longer maintained by the system.
[EFAULT]	*addr* is not a valid pointer.
[EINPROGRESS]	Non-blocking I/O is enabled using `O_NONBLOCK`, `O_NDELAY`, or `FIOSNBIO`, and the connection cannot be completed immediately. This is not a failure. Make the `connect()` call again a few seconds later. Alternatively, wait for completion by calling `select()`, selecting for write.
[ENODEV]	The *x25ifname* field refers to a non-existent interface.
[ENOSPC]	All available virtual circuits are in use.
[ENETDOWN]	The X.25 interface specified in the *addr* struct was found or but was not in the initialized state. *x25ifname* field name is an interface which has been shut down or never initialized or suffered a power failure which erased its state information.
[ENOBUFS]	No buffer space is available. The `connect()` has failed.

[EINTR] The connect was interrupted by delivery of a signal before the connect sequence was complete. The building of the connection still takes place, even though the user is not blocked on the `connect()` call.

[EOPNOTSUPP] A `connect()` attempt was made on a socket type which does not support this call. Under X.25 an attempt was made to issue a `connect()` call on a `listen()` socket.

AUTHOR

`connect()` was developed by the University of California, Berkeley.

SEE ALSO

accept(2), select(2), socket(2), getsockname(2), socket(7), socketx25(7), af_ccitt(7F).

NAME

creat - create a new file or rewrite an existing one

SYNOPSIS

```
#include <fcntl.h>

int creat(const char *path, mode_t mode);
```

DESCRIPTION

`creat()` creates a new regular file or prepares to rewrite an existing file named by the path name pointed to by *path*.

If the file exists, its length is truncated to 0, and its mode and owner are unchanged. Otherwise, the file's owner ID is set to the effective user ID of the process. If the set-group-ID bit of the parent directory is set, the file's group ID is set to the group ID of the parent directory. Otherwise, the file's group ID is set to the process's effective group ID. The low-order 12 bits of the file mode are set to the value of *mode* modified as follows:

- All bits set in the process's file mode creation mask are cleared (see *umask*(2)).
- The "save text image after execution" bit of the mode is cleared (see *chmod*(2)).

Upon successful completion, the file descriptor is returned and the file is open for writing (only), even if the *mode* does not permit writing. The file offset is set to the beginning of the file. The file descriptor is set to remain open across `exec()` system calls (see *fcntl*(2)). No process can have more than **OPEN_MAX** files open simultaneously. This is discussed in *open*(2). A new file can be created with a mode that forbids writing.

Access Control Lists (ACLs)

On systems that support access control lists, three base ACL entries are created corresponding to the file access permission bits. An existing file's access control list is unchanged by `creat()` (see *setacl*(2), *chmod*(2), and *acl*(5)).

ERRORS

`creat()` fails if one or more of the following is true:

[EACCES] Search permission is denied on a component of the path prefix.

[EACCES] The file does not exist and the directory in which the file is to be created does not permit writing.

[EACCES] The file exists and write permission is denied.

[EAGAIN] The file exists, enforcement mode file and record locking is set and there are outstanding record locks on the file.

[EDQUOT] User's disk quota block or inode limit has been reached for this file system.

[EFAULT] *path* points outside the allocated address space of the process. The reliable detection of this error is implementation dependent.

[EISDIR] The named file is an existing directory.

[ELOOP] Too many symbolic links were encountered in translating the path name.

[EMFILE] More than the maximum number of file descriptors are currently open.

[ENAMETOOLONG] The length of the specified path name exceeds **PATH_MAX** bytes, or the length of a component of the path name exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect.

[ENFILE] The system file table is full.

[ENOENT] The named file does not exist (for example, *path* is null, or a component of *path* does not exist).

[ENOSPC] Not enough space on the file system.

[ENOTDIR] A component of the path prefix is not a directory.

[ENXIO] The named file is a character special or block special file, and the device associated with this special file does not exist.

[EROFS] The named file resides or would reside on a read-only file system.

[ETXTBSY] The file is a pure procedure (shared text) file that is being executed.

RETURN VALUE

Upon successful completion, a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

WARNINGS

Access Control Lists

Access control list descriptions in this entry apply only to standard HP-UX operating systems. If HP-UX BLS software has been installed, access control lists are handled differently. Refer to HP-UX BLS documentation for information about access control lists in the HP-UX BLS environment.

SEE ALSO

chmod(2), close(2), dup(2), fcntl(2), lockf(2), lseek(2), open(2), read(2), setacl(2), truncate(2), umask(2), write(2), acl(5).

STANDARDS CONFORMANCE

`creat()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

dup - duplicate an open file descriptor

SYNOPSIS

```
#include <unistd.h>

int dup(int fildes);
```

DESCRIPTION

fildes is a file descriptor obtained from a `creat()`, `open()`, `dup()`, `fcntl()`, or `pipe()` system call. `dup()` returns a new file descriptor having the following in common with the original:

- Same open file (or pipe).
- Same file pointer (i.e., both file descriptors share one file pointer).
- Same access mode (read, write or read/write).
- Same file status flags (see *fcntl*(2), `F_DUPFD`).

The new file descriptor is set to remain open across `exec()` system calls. See *fcntl*(2).

The file descriptor returned is the lowest one available.

RETURN VALUE

Upon successful completion, the file descriptor is returned as a non-negative integer. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

ERRORS

`dup()` fails if one or more of the following is true:

[EBADF]	*fildes* is not a valid open file descriptor.
[EMFILE]	Request violates the maximum number of open file descriptors.

AUTHOR

`dup()` was developed by AT&T and HP.

SEE ALSO

close(2), creat(2), dup2(2), exec(2), fcntl(2), open(2), pipe(2).

STANDARDS CONFORMANCE

`dup()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

dup2 - duplicate an open file descriptor to a specific slot

SYNOPSIS

```
#include <unistd.h>

int dup2(int fildes, int fildes2);
```

DESCRIPTION

fildes is a file descriptor obtained from a `creat()`, `open()`, `dup()`, `fcntl()`, or `pipe()` system call.

fildes2 is a non-negative integer less than the maximum value allowed for file descriptors.

`dup2()` causes *fildes2* to refer to the same file as *fildes*. If *fildes2* refers to an already open file, the open file is closed first.

The file descriptor returned by `dup2()` has the following in common with *fildes*:

- Same open file (or pipe).
- Same file pointer (that is, both file descriptors share one file pointer.)
- Same access mode (read, write or read/write).
- Same file status flags (see *fcntl*(2), `F_DUPFD`).

The new file descriptor is set to remain open across `exec()` system calls. See *fcntl*(2).

This routine is found in the C library. Programs using `dup2()` but not using other routines from the Berkeley importability library (such as the routines described in *bsdproc*(2)) should not give the `-lBSD` option to *ld*(1).

RETURN VALUE

Upon successful completion, `dup2()` returns the new file descriptor as a non-negative integer, *fildes2*. Otherwise, it returns -1 and sets `errno` to indicate the error.

ERRORS

`dup2()` fails if the following is true:

[EBADF] *fildes* is not a valid open file descriptor or *fildes2* is not in the range of legal file descriptors.

[EINTR] An attempt to close *fildes2* was interrupted by a signal. The file is still open.

SEE ALSO

close(2), creat(2), dup(2), exec(2), fcntl(2), open(2), pipe(2).

STANDARDS CONFORMANCE

`dup2()`: AES, SVID2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

errno - error indicator for function calls

SYNOPSIS

```
#include <errno.h>

extern int errno;
```

DESCRIPTION

Many functions in the HP-UX operating system indicate an error condition by returning an otherwise out-of-range value (usually `-1`). Most of these functions set the external variable `errno` to a non-zero code value that more specifically identifies the particular error condition that was encountered.

All errors detected and the corresponding error code values stored in `errno` are documented in the ERRORS section on manual pages for those functions that set it.

The value of `errno` is zero immediately after a successful call to any of the functions described by *exec*(2) and *ptrace*(2), but it is never set to zero by any other HP-UX function. Functions for which the use of `errno` is not described may nevertheless change its value to a non-zero value.

Since `errno` is *not* cleared on successful function calls, its value should be checked or used *only* when an error has been indicated and when the function's ERRORS section documents the error codes.

Applications should not attempt to take the address of `errno`, because it may be converted to a macro in a future release.

The following is a complete list of the error codes. The numeric values can be found in <`errno.h`> but they should not be used in an application program because they can vary from system to system.

E2BIG — Arg list too long. An argument and or environment list longer than maximum supported size is presented to a member of the `exec()` family. Other possibilities include: message size or number of semaphores exceeds system limit (`msgop`, `semop`), or too many privileged groups have been set up (`setprivgrp`).

EACCES — Permission denied. An attempt was made to access a file or IPC object in a way forbidden by the protection system.

EADDRINUSE — Address already in use. Only one usage of each address is normally permitted.

EADDRNOTAVAIL — Cannot assign requested address. Normally results from an attempt to create a socket with an address not on this machine.

EAFNOSUPPORT — Address family not supported by protocol family. An address incompatible with the requested protocol was used. For example, you should not necessarily expect to be able to use PUP Internet addresses with ARPA Internet protocols.

EAGAIN — No more processes. A `fork()` failed because the system's process table is full or the user is not allowed to create any more processes, or a `semop()` or `msgop()` call would have to block.

EALREADY — Operation already in progress. An operation was attempted on a non-blocking object which already had an operation in progress.

EBADF — Bad file number. Either a file descriptor refers to no open file, a read (respectively write) request is made to a file which is open only for writing (respectively reading), or the file descriptor is not in the legal range of file descriptors.

EBUSY — Device or resource busy. An attempt to mount a device that was already mounted or an attempt was made to dismount a device on which there is an active file (open file, current directory, mounted-on file, active text segment). It will also occur if an attempt is made to enable accounting when it is already enabled. The device or resource is currently unavailable, such as when a non-shareable device file is in use.

ECHILD — No child processes. A `wait()` was executed by a process that had no existing or unwaited-for child processes.

ECONNABORTED
: Software caused connection abort. A connection abort was caused internal to your host machine.

ECONNREFUSED
: Connection refused. No connection could be made because the target machine actively refused it. This usually results from trying to connect to a service that is inactive on the foreign host.

ECONNRESET
: Connection reset by peer. A connection was forcibly closed by a peer. This normally results from the peer executing a **shutdown()** call (see *shutdown*(2)).

EDEADLK
: Resource deadlock would occur. A process which has locked a system resource would have been put to sleep while attempting to access another process' locked resource.

EDESTADDRREQ
: Destination address required. A required address was omitted from an operation on a socket.

EDOM
: Math argument. The argument of a function in the math package (3M) is out of the domain of the function.

EEXIST
: File exists. An existing file was mentioned in an inappropriate context; e.g., **link()**.

EFAULT
: Bad address. The system encountered a hardware fault in attempting to use an argument of a system call; can also result from passing the wrong number of parameters to a system call. The reliable detection of this error is implementation dependent.

EFBIG
: File too large. The size of a file exceeded the maximum file size (for the file system) or **ULIMIT** was exceeded (see *ulimit*(2)), or a bad semaphore number in a **semop()** call (see **semop**(2)).

EHOSTDOWN
: Host is down. A socket operation encountered a dead host. Networking activity on the local host has not been initiated.

EHOSTUNREACH
: No route to host. A socket operation was attempted to an unreachable host.

EIDRM
: Identifier Removed. This error is returned to processes that resume execution due to the removal of an identifier from the file system's name space (see *msgctl*(2), *semctl*(2), and *shmctl*(2)).

EILSEQ
: Illegal byte sequence. A wide character code has been detected that does not correspond to a valid character, or a byte sequence does not form a valid wide character code.

EINPROGRESS
: Operation now in progress. An operation that takes a long time to complete was attempted on a non-blocking object (see *ioctl*(2) and *fcntl*(2)).

EINTR
: Interrupted system call. An asynchronous signal (such as interrupt or quit), which the user has elected to catch, occurred during a system call. If execution is resumed after processing the signal, it will appear as if the interrupted system call returned this error condition unless the system call is restarted (see *sigvector*(2)).

EINVAL
: Invalid argument. Some invalid argument (such as unmounting a device that is not currently mounted, mentioning an undefined signal in **signal()** or **kill()**, or reading or writing a file for which **lseek()** has generated a negative pointer). Also set by the math functions described in the (3M) entries of this manual.

EIO
: I/O error - some physical I/O error. This error may in some cases occur on a call following the one to which it actually applies.

EISCONN
: Socket is already connected. A **connect()** request was made on an already connected socket, or, a **sendto()** or **sendmsg()** request on a connected socket specified a destination other than the connected party.

EISDIR
: Is a directory. An attempt to open a directory for writing.

ELOOP
: Too many levels of symbolic links. A path name search involved more than **MAXSYMLINKS** symbolic links. **MAXSYMLINKS** is defined in **<sys/param.h>**.

EMFILE Too many open files. No process may have more than a system-defined number of file descriptors open at a time.

EMLINK Too many links. An attempt to make more than the maximum number of links to a file.

EMSGSIZE Message too long. The socket requires that the message be sent atomically, and the size of the message to be sent made this impossible.

ENAMETOOLONG
File name too long. A path specified exceeds the maximum path length for the system. The maximum path length is specified by **PATH_MAX** and is defined in **<limits.h>**. PATH_MAX is guaranteed to be at least 1023 bytes. This error is also generated if the length of a path name component exceeds **NAME_MAX** and the **_POSIX_NO_TRUNC** option is in effect for the specified path. Currently, **_POSIX_NO_TRUNC** is in effect only for HFS file systems configured to allow path name components up to 255 bytes long (see *convertfs*(1M)) and therefore only path names referring to such file systems can generate the error for this case. The values of **NAME_MAX**, **PATH_MAX**, and **_POSIX_NO_TRUNC** for a particular path name can be queried by using the **pathconf()** system call (see *pathconf*(2)).

ENETDOWN Network is down. A socket operation encountered a dead network.

ENETRESET Network dropped connection on reset. The host you were connected to crashed and rebooted.

ENETUNREACH
Network is unreachable. A socket operation was attempted to an unreachable network.

ENFILE File table overflow. The system's table of open files is full, and temporarily no more **open()**s can be accepted.

ENOBUFS No buffer space available. An operation on a socket was not performed because the system lacked sufficient buffer space.

ENODEV No such device. An attempt was made to apply an inappropriate system call to a device (such as read a write-only device).

ENOENT No such file or directory. This error occurs when a file name is specified and the file should exist but does not, or when one of the directories in a path name does not exist. It also occurs with **msgget()**, **semget()**, and **shmget()** when *key* does not refer to any object and the **IPC_CREAT** flag is not set.

ENOEXEC Exec format error. A request is made to execute a file which, although it has the appropriate permissions, does not start with a valid magic number (see *a.out*(4)), or the file is too small to have a valid executable file header.

ENOLCK System lock table is full. Too many files have file locks on them, or there are too many record locks on files, or there are too many instances of a reading or writing process sleeping until an enforcement mode lock clears. This error may also indicate system problems in handling a lock request on a remote NFS file. This error is also currently returned for all attempts to perform locking operations on a remote NFS file that has its locking enforcement mode bit set, since the stateless nature of NFS prevents maintaining the necessary lock information.

ENOMEM Not enough space. During a system call such as **exec()**, **brk()**, **fork()**, or **sbrk()**, a program asks for more space than the system is able to supply. This may not be a temporary condition; the maximum space size is a system parameter. The error can also occur if the arrangement of text, data, and stack segments requires too many segmentation registers, or if there is not enough swap space during a **fork()**.

ENOMSG No message of desired type. An attempt was made to receive a message of a type that does not exist on the specified message queue; see *msgop*(2).

ENOPROTOOPT Protocol not available. A bad option was specified in a **getsockopt()** or **setsockopt()** call (see *getsockopt*(2)).

ENOSPC No space left on device. During a `write()` to an ordinary file, there is no free space left on the device; or no space in system table during `msgget()`, `semget()`, or `semop()` while `SEM_UNDO` flag is set.

ENOSYM Symbol does not exist in executable. The dynamic loader was unable to resolve a symbolic reference in a shared library during a call to one of the dynamic loader interface routines (see *shl_load*(3X). The program may be in an inconsistent state and should be terminated immediately.

ENOSYS Function is not available. The requested function or operation is not implemented or not configured in the system.

ENOTBLK Block device required. A non-block file was mentioned where a block device was required, such as in `mount()`.

ENOTCONN Socket is not connected. A request to send or receive data was disallowed because the socket was not connected.

ENOTDIR Not a directory. A non-directory was specified where a directory is required, such as in a path prefix or as an argument to `chdir()`.

ENOTEMPTY Directory not empty. An attempt was made to remove a non-empty directory.

ENOTSOCK Socket operation on non-socket. An operation was attempted on something that is not a socket.

ENOTTY Not a typewriter. The (`ioctl()`) command is inappropriate to the selected device type.

ENXIO No such device or address. I/O on a special file refers to a subdevice that does not exist, or is beyond the limits of the device. It can also occur when, for example, a tape drive is not on line or no disk pack is loaded on a drive.

EOPNOTSUPP Operation not supported. The requested operation on a socket or NFS file is either invalid or unsupported. For example, this might occur when an attempt to `accept()` a connection on a datagram socket fails.

EPERM Not owner. Typically, this error indicates an attempt to modify a file in some way forbidden except to its owner or the super-user, such as to change its mode. It is also returned for attempts by ordinary users to do things for which they need, but lack, a special privilege.

EPFNOSUPPORT
Protocol family not supported. The protocol family has not been configured into the system or no implementation for it exists. The socket is not connected.

EPIPE Broken pipe. Data has been written to a pipe for which the other (reading) end has been closed. This most often occurs when the reading process exits before the writing process. This condition also generates the signal `SIGPIPE`; the error is returned if the signal is ignored.

EPROTONOSUPPORT
Protocol not supported. The protocol has not been configured into the system or no implementation for it exists.

EPROTOTYPE Protocol wrong type for socket. A protocol was specified that does not support the semantics of the socket type requested. For example, ARPA Internet UDP protocol cannot be used with type `SOCK_STREAM`.

ERANGE Result too large. The value of a function in the math package (3M) is not representable within machine precision, or a `semop()` call would cause either a semaphore value or a semaphore adjust value to exceed it system-imposed maximum.

EROFS Read-only file system. An attempt to modify a file or directory was made on a device mounted read-only.

ESHUTDOWN Cannot send after socket shutdown. A request to send data was disallowed because the socket had already been shut down with a previous `shutdown()` call.

ESOCKTNOSUPPORT
Socket type not supported. The support for the socket type has not been configured into the

system or no implementation for it exists.

ESPIPE Illegal seek. An `lseek()` was issued to a pipe.

ESRCH No such process. No process can be found corresponding to that specified by *pid* in `kill()`, `rtprio()`, or `ptrace()`, or the process is not accessible.

ETIMEDOUT Connection timed out. A `connect()` request failed because the connected party did not properly respond after a period of time (timeout period varies, depending on the communication protocol).

ETXTBSY Text file busy. An attempt to execute an executable file which is currently open for writing (or reading). Also, an attempt to open for writing an otherwise writable file which is currently open for execution.

EWOULDBLOCK
Operation would block. An operation which would cause a process to block was attempted on an object in non-blocking mode (see *ioctl*(2) and *fcntl*(2)).

EXDEV Cross-device link. A link to a file on another device was attempted.

DEPENDENCIES

The following NFS errors are also defined:

EREFUSED The same error as ECONNREFUSED. The external variable `errno` is defined as ECONNREFUSED for NFS compatibility.

EREMOTE Too many levels of remote in path. An attempt was made to remotely mount an NFS file system into a path which already has a remotely mounted NFS file system component.

ESTALE Stale NFS file handle. A client referenced an open file, but the file was previously deleted.

Series 700/800:
In the definition of error ENOMEM, the term "segmentation registers" is invalid.

STANDARDS CONFORMANCE

`errno`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

execl, execv, execle, execve, execlp, execvp - execute a file

SYNOPSIS

```
#include <unistd.h>

extern char **environ;

int execl(
    const char *path,
    const char *arg0, ...
    /* const char *arg1,
       ...,
       const char *argn,
       (char *)0 */
);

int execv(const char *path, char * const argv[]);

int execle(
    const char *path,
    const char *arg0, ...
    /* const char *arg1,
       ...,
       const char *argn,
       (char *)0,
       char * const envp[] */
);

int execve(const char *file, char * const argv[], char * const envp[]);

int execlp(
    const char *file,
    const char *arg0, ...
    /* const char *arg1,
       ... ,
       const char *argn,
       (char *)0 */
);

int execvp(const char *file, char * const argv[]);
```

DESCRIPTION

exec(), in all its forms, loads a program from an ordinary, executable file onto the current process, replacing the current program. The *path* or *file* argument refers to either an executable object file or a file of data for an interpreter. In this case, the file of data is also called a script file.

An executable object file consists of a header (see *a.out*(4)), text segment, and data segment. The data segment contains an initialized portion and an uninitialized portion (bss). For **execlp**() and **execvp**() the shell (**/bin/sh**) can be loaded to interpret a script instead. A successful call to **exec**() does not return because the new program overwrites the calling program.

When a C program is executed, it is called as follows:

```
main (argc, argv, envp)
int argc;
char **argv, **envp;
```

where *argc* is the argument count and *argv* is the address of an array of character pointers to the arguments themselves. As indicated, *argc* usually has a value of at least one, and the first member of the array points to a string containing the name of the file. (Exit conditions from *main* are discussed in *exit*(2).)

path points to a path name that identifies the executable file containing the new program.

file (in **execlp**() or **execvp**()) points to a file name identifying the executable file containing the new program. The path prefix for this file is obtained by searching the directories passed as the environment line **PATH** = (see *environ*(5)). The environment is supplied by the shell (see *sh*(1)). If *file* does not have an

executable magic number (*magic*(4)), it is passed to **/bin/sh** as a shell script.

arg0, *arg1*, ..., *argn* are pointers to null-terminated character strings. These strings constitute the argument list available to the new program. By convention, at least *arg0* must be present and point to a string identical to *path* or *path*'s last component.

argv is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new program. By convention, *argv* must have at least one member, and must point to a string that is identical to *path* or *path*'s last component. *argv* is terminated by a null pointer.

envp is an array of character pointers to null-terminated strings. These strings constitute the environment in which the new program runs. *envp* is terminated by a null pointer. For **exec()** and **execv()**, the C run-time start-off routine places a pointer to the environment of the calling program in the global cell:

```
extern char **environ;
```

and it is used to pass the environment of the calling program to the new program.

Open file descriptors remain open, except for those whose close-on-exec flag is set (see *fcntl*(2)). The file offset, access mode, and status flags of open file descriptors are unchanged.

Note that normal executable files are open only briefly when they start execution. Other executable file types can be kept open for a long time, or even indefinitely under some circumstances.

The processing of signals by the process is unchanged by **exec()**, except that signals caught by the process are set to their default value (see *signal*(2)).

If the set-user-ID mode bit of the executable file pointed to by *path* or *file* is set (see *chmod*(2)), **exec()** sets the effective-user-ID of the new process to the user ID of the executable file. Similarly, if the set-group-ID mode bit of the executable file is set, the effective-group-ID of the process is set to the group ID of the executable file. The real-user-ID and real-group-ID of the process are unchanged. Note that the set-user(group)-ID function does not apply to scripts; thus, if **execlp()** or **execvp()** executes a script, the set-user(group)-ID bits are ignored, even if they are set.

The saved-user-ID and saved-group-ID of the process are always set to the effective-user-ID and effective-group-ID, respectively, of the process at the end of the *exec*, whether or not set-user(group)- ID is in effect.

The shared memory segments attached to the calling program are not attached to the new program (see *shmop*(2)).

Text and data segment memory locks are not passed on to the new program (see *plock*(2)).

Profiling is disabled for the new process; see *profil*(2).

The process also retains the following attributes:

- current working directory
- file creation mode mask (see *umask*(2))
- file locks (see *fcntl*(2)), except for files closed-on-exec
- file size limit (see *ulimit*(2))
- interval timers (see *getitimer*(2))
- nice value (see *nice*(2))
- nice value (see parent process ID
- pending signals
- process ID
- process group ID
- real user ID
- real group ID
- real-time priority (see *rtprio*(2))
- root directory (see *chroot*(2))
- *semadj* values (see *semop*(2))
- session membership
- signal mask (see *sigvector*(2))
- supplementary group IDs
- time left until an alarm clock signal (see *alarm*(2))
- trace flag (see *ptrace*(2) PT_SETTRC request)

- **tms_utime**, **tms_stime**, **tms_cutime**, and **tms_cstime** (see *times*(2))

The initial line of a script file must begin with **#!** as the first two bytes, followed by zero or more spaces, followed by *interpreter* or *interpreter argument*. One or more spaces or tabs must separate *interpreter* and *argument*. The first line should end with either a new-line or null character.

#! *interpreter*
#! *interpreter argument*

When the script file is executed, the system executes the specified *interpreter* as an executable object file. Even in the case of **execlp()** or **execvp()**, no path searching is done of the interpreter name.

The *argument* is anything that follows the *interpreter* and tabs or spaces. If an *argument* is given, it is passed to the *interpreter* as **argv[1]**, and the name of the script file is passed as **argv[2]**. Otherwise, the name of the script file is passed as **argv[1]**. The **argv[0]** is passed as specified in the **exec()** call, unless either **argv** or **argv[0]** is null as specified, in which case a pointer to a null string is passed as **argv[0]**. All other arguments specified in the **exec()** call are passed following the name of the script file (that is, beginning at **argv[3]** if there is an argument; otherwise at **argv[2]**).

If the initial line of the script file exceeds a system-defined maximum number of characters, **exec()** fails. The minimum value for this limit is 32.

Set-user-ID and set-group-ID bits are honored for the script but not for the interpreter.

RETURN VALUE

If **exec()** returns to the calling program, an error has occurred; the return value is **-1** and **errno** is set to indicate the error.

ERRORS

exec() fails and returns to the calling program if one or more of the following is true:

[E2BIG] The number of bytes in the new program's argument list is greater than the system-imposed limit. This limit is at least 5120 bytes on HP-UX systems.

[EACCES] Read permission is denied for the executable file or interpreter, and trace flag (see *ptrace*(2) request **PT_SETTRC**) of the process is set.

[EACCES] Search permission is denied for a directory listed in the executable file's or the interpreter's path prefix.

[EACCES] The executable file or the interpreter is not an ordinary file.

[EACCES] The file described by *path* or *file* is not executable. The super-user cannot execute a file unless at least one access permission bit or entry in its access control list has an execute bit set.

[EFAULT] *path*, *argv*, or *envp* point to an illegal address. The reliable detection of this error is implementation dependent.

[EFAULT] The executable file is shorter than indicated by the size values in its header, or is otherwise inconsistent. The reliable detection of this error is implementation dependent.

[EINVAL] The executable file is incompatible with the architecture on which the **exec()** has been performed, and is presumed to be for a different architecture. It is not guaranteed that every architecture's executable files will be recognized.

[ELOOP] Too many symbolic links are encountered in translating the path name.

[ENAMETOOLONG] The executable file's path name or the interpreter's path name exceeds **PATH_MAX** bytes, or the length of a component of the path name exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect.

[ENOENT] *path* is null.

[ENOENT] One or more components of the executable file's path name or the interpreter's path name does not exist.

[ENOEXEC] The **exec()** is not an **execlp()** or **execvp()**, and the executable file has the appropriate access permission, but there is neither a valid magic number nor the

characters `#!` as the first two bytes of its initial line.

[ENOEXEC] The number of bytes in the initial line of a script file exceeds the system's maximum.

[ENOMEM] The new process requires more memory than is available or allowed by the system-imposed maximum.

[ENOTDIR] A component of the executable file's path prefix or the interpreter's path prefix is not a directory.

[ETXTBSY] The executable file is currently open for writing.

WARNINGS

Access Control Lists

Access control list descriptions in this entry apply only to standard HP-UX operating systems. If HP-UX BLS software has been installed, access control lists are handled differently. Refer to HP-UX BLS documentation for information about access control lists in the HP-UX BLS environment.

DEPENDENCIES

Series 700/800

Unsharable executable files (EXEC_MAGIC magic number produced via the **-N** option of *ld*(1)) are not supported.

SEE ALSO

sh(1), alarm(2), exit(2), fork(2), nice(2), ptrace(2), semop(2), signal(2), times(2), ulimit(2), umask(2), a.out(4), acl(5), environ(5), signal(5).

STANDARDS CONFORMANCE

environ: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

execl(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

execle(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

execlp(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

execv(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

execve(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

execvp(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

exit, _exit - terminate process

SYNOPSIS

```
#include <stdlib.h>

void exit(int status);

#include <unistd.h>

void _exit(int status);
```

DESCRIPTION

`exit()` terminates the calling process and passes *status* to the system for inspection, see *wait*(2). Returning from *main* in a C program has the same effect as `exit();` the *status* value is the function value returned by *main* (this value is undefined if *main* does not take care to return a value or to call `exit()` explicitly).

`exit()` cannot return to its caller. The result of an `exit()` call during exit processing is undefined.

The functions `exit()` and `_exit()`, are equivalent, except that `exit()` calls functions registered by `atexit()` and flushes standard I/O buffers, while `_exit()` does not. Both `exit()` and `_exit()` terminate the calling process with the following consequences:

Functions registered by `atexit()` (see *atexit*(2)) are called in reverse order of registration.

All file descriptors open in the calling process are closed.

All files created by `tmpfile()` are removed (see *tmpfile*(3S)).

If the parent process of the calling process is executing a `wait()`, `wait3()`, or `waitpid()`, it is notified of the calling process's termination, and the low-order eight bits; i.e., bits 0377 of *status* are made available to it (see *wait*(2)).

If the parent process of the calling process is not executing a `wait()`, `wait3()`, or `waitpid()`, and does not have `SIGCLD` set to `SIG_IGN`, the calling process is transformed into a **zombie process. A zombie process** is a process that only occupies a slot in the process table. It has no other space allocated either in user or kernel space. Time accounting information is recorded for use by `times()` (see *times*(2)).

The parent process ID is set to 1 for all of the calling process's existing child processes and zombie processes. This means the initialization process (proc1) inherits each of these processes.

Each attached shared memory segment is detached and the value of `shm_nattach` in the data structure associated with its shared memory identifier is decremented by 1 (see *shmop*(2)).

For each semaphore for which the calling process has set a semadj value (see *semop*(2)), that semadj value is added to the semval of the specified semaphore.

If the process has a process, text, or data lock, an `unlock()` is performed, see *plock*(2).

An accounting record is written on the accounting file if the system's accounting routine is enabled (see *acct*(2)).

A `SIGCHLD` signal is sent to the parent process.

If the calling process is a controlling process, the `SIGHUP` signal is sent to each process in the foreground process group of the controlling terminal belonging to the calling process. The controlling terminal associated with the session is disassociated from the session, allowing it to be acquired by a new controlling process.

If the exit of the calling process causes a process group to become orphaned, and if any member of the newly-orphaned process group is stopped, all processes in the newly-orphaned process group are sent `SIGHUP` and `SIGCONT` signals.

If the current process has any child processes that are being traced, they are sent a `SIGKILL` signal.

AUTHOR

`exit()` was developed by HP, AT&T, and the University of California, Berkeley.

SEE ALSO

Exit conditions (**$?**) in sh(1),
acct(2), plock(2), semop(2), shmop(2), times(2), vfork(2), wait(2), signal(5).

STANDARDS CONFORMANCE

exit(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

_exit(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

fcntl - file control

SYNOPSIS

```
#include <fcntl.h>

int fcntl(int fildes, int cmd, ... /* arg */);
```

DESCRIPTION

`fcntl()` provides for control over open files. *fildes* is an open file descriptor.

The following are possible values for the *cmd* argument:

F_DUPFD Return a new file descriptor having the following characteristics:

- Lowest numbered available file descriptor greater than or equal to *arg*.**val**.
- Same open file (or pipe) as the original file.
- Same file pointer as the original file (that is, both file descriptors share one file pointer).
- Same access mode (read, write or read/write).
- Same file status flags (that is, both file descriptors share the same file status flags).
- The close-on-exec flag associated with the new file descriptor is set to remain open across *exec*(2) system calls.

F_GETFD Get the close-on-exec flag associated with the file descriptor *fildes*. If the low-order bit is 0 the file will remain open across *exec*(2), otherwise the file will be closed upon execution of *exec*(2).

F_SETFD Set the close-on-exec flag associated with *fildes* to the low-order bit of *arg*.**val** (see **F_GETFD**).

F_GETFL Get file status flags and access modes; see *fcntl*(5).

F_SETFL Set file status flags to *arg*.*val*. Only certain flags can be set; see *fcntl*(5). It is not possible to set both **O_NDELAY** and **O_NONBLOCK**.

F_GETLK Get the first lock that blocks the lock described by the variable of type `struct flock` pointed to by *arg*. The information retrieved overwrites the information passed to `fcntl()` in the `flock` structure. If no lock is found that would prevent this lock from being created, the structure is passed back unchanged, except that the lock type is set to **F_UNLCK**.

F_SETLK Set or clear a file segment lock according to the variable of type `struct flock` pointed to by *arg*.*lockdes* (see *fcntl*(5)). The *cmd* **F_SETLK** is used to establish read (**F_RDLCK**) and write (**F_WRLCK**) locks, as well as to remove either type of lock (**F_UNLCK**). If a read or write lock cannot be set, `fcntl()` returns immediately with an error value of −1.

F_SETLKW This *cmd* is the same as **F_SETLK** except that if a read or write lock is blocked by other locks, the process will sleep until the segment is free to be locked.

A read lock prevents any other process from write-locking the protected area. More than one read lock can exist for a given segment of a file at a given time. The file descriptor on which a read lock is being placed must have been opened with read access.

A write lock prevents any other process from read-locking or write-locking the protected area. Only one write lock may exist for a given segment of a file at a given time. The file descriptor on which a write lock is being placed must have been opened with write access.

The structure `flock` describes the type (`l_type`), starting offset (`l_whence`), relative offset (`l_start`), size (`l_len`), and process ID (`l_pid`) of the segment of the file to be affected. The process ID field is only used with the **F_GETLK** *cmd* to return the value of a block in lock. Locks can start and extend beyond the current end of a file, but cannot be negative relative to the beginning of the file. A lock can be set to always extend to the end of file by setting `l_len` to zero (0). If such a

lock also has `l_start` set to zero (0), the whole file will be locked. Changing or unlocking a segment from the middle of a larger locked segment leaves two smaller segments for either end. Locking a segment already locked by the calling process causes the old lock type to be removed and the new lock type to take effect. All locks associated with a file for a given process are removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process in a *fork*(2) system call.

When enforcement-mode file and record locking is activated on a file (see *chmod*(2)), future `read()` and `write()` system calls on the file are affected by the record locks in effect.

NETWORKING FEATURES

NFS The advisory record-locking capabilities of *fcntl(2)* are implemented throughout the network by the "network lock daemon" (see *lockd(1M)*). If the file server crashes and is rebooted, the lock daemon attempts to recover all locks associated with the crashed server. If a lock cannot be reclaimed, the process that held the lock is issued a `SIGLOST` signal.

Record locking, as implemented for NFS files, is only advisory.

RETURN VALUE

Upon successful completion, the value returned depends on *cmd* as follows:

`F_DUPFD`	A new file descriptor.
`F_GETFD`	Value of close-on-exec flag (only the low-order bit is defined).
`F_SETFD`	Value other than –1.
`F_GETFL`	Value of file status flags and access modes.
`F_SETFL`	Value other than –1.
`F_GETLK`	Value other than –1.
`F_SETLK`	Value other than –1.
`F_SETLKW`	Value other than –1.

Otherwise, a value of –1 is returned and `errno` is set to indicate the error.

ERRORS

`fcntl()` fails if any of the following conditions occur:

[EBADF]	*fildes* is not a valid open file descriptor, or was not opened for reading when setting a read lock or for writing when setting a write lock.
[EMFILE]	*cmd* is `F_DUPFD` and the maximum number of file descriptors is currently open.
[EMFILE]	*cmd* is `F_SETLK` or `F_SETLKW`, the type of lock is a read or write lock, and no more file-locking headers are available (too many files have segments locked).
[EMFILE]	*cmd* is `F_DUPFD` and *arg* . *val* is greater than or equal to the maximum number of file descriptors.
[EMFILE]	*cmd* is `F_DUPFD` and *arg* . *val* is negative.
[EINVAL]	*cmd* is `F_GETLK`, `F_SETLK`, or `F_SETLKW`, and *arg* . *lockdes* or the data it points to is not valid, or *fildes* refers to a file that does not support locking.
[EINVAL]	*cmd* is not a valid command.
[EINVAL]	*cmd* is `F_SETFL` and both `O_NONBLOCK` and `O_NDELAY` are specified.
[EINTR]	*cmd* is `F_SETLKW` and the call was interrupted by a signal.
[EACCES]	*cmd* is `F_SETLK`, the type of lock (`l_type`) is a read lock (`F_RDLCK`) or write lock (`F_WRLCK`) and the segment of a file to be locked is already write-locked by another process, or the type is a write lock (`F_WRLCK`) and the segment of a file to be locked is already read- or write-locked by another process.
[ENOLCK]	*cmd* is `F_SETLK` or `F_SETLKW`, the type of lock is a read or write lock, and no more file-locking headers are available (too many files have segments locked), or no more record locks are available (too many file segments locked).

[ENOLCK] *cmd* is **F_SETLK** or **F_SETLKW**, the type of lock (**l_type**) is a read lock (**F_RDLCK**) or write lock (**F_WRLCK**) and the file is an NFS file with access bits set for enforcement mode.

[ENOLCK] *cmd* is **F_GETLK**, **F_SETLK**, or **F_SETLKW**, the file is an NFS file, and a system error occurred on the remote node.

[EDEADLK] *cmd* is **F_SETLKW**, when the lock is blocked by a lock from another process and sleeping (waiting) for that lock to become free. This causes a deadlock situation.

[EAGAIN] *cmd* is **F_SETLK** or **F_SETLKW**, and the file is mapped in to virtual memory via the **mmap()** system call (see *mmap*(2)).

[EFAULT] *cmd* is either **F_GETLK**, **F_SETLK**, or **F_SETLKW**, and *arg* points to an illegal address.

AUTHOR

fcntl() was developed by HP, AT&T and the University of California, Berkeley.

APPLICATION USAGE

Because in the future the external variable **errno** will be set to EAGAIN rather than EACCES when a section of a file is already locked by another process, portable application programs should expect and test for either value, for example:

```
flk->l_type = F_RDLCK;
   if (fcntl(fd, F_SETLK, flk) == -1)
      if ((errno == EACCES) || (errno == EAGAIN))
            /*
             * section locked by another process,
             * check for either EAGAIN or EACCES
             * due to different implementations
             */
       else if ...
            /*
             * check for other errors
             */
```

SEE ALSO

lockd(1M), statd(1M), chmod(2), close(2), exec(2), lockf(2), open(2), read(2), write(2), fcntl(5).

FUTURE DIRECTIONS

The error condition which currently sets **errno** to EACCES will instead set **errno** to EAGAIN (see also APPLICATION USAGE above).

STANDARDS CONFORMANCE

fcntl(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

fork - create a new process

SYNOPSIS

```
#include <unistd.h>

pid_t fork(void);
```

DESCRIPTION

`fork()` causes the creation of a new process. The new process (child process) is an exact copy of the calling process (parent process). This means that the child process inherits the following attributes from the parent process:

- Real, effective, and saved user ID.
- Real, effective, and saved group ID.
- List of supplementary group IDs (see *getgroups*(2)).
- Process group. ID
- Environment.
- File descriptors.
- Close-on-exec flags (see *exec*(2)).
- Signal handling settings (`SIG_DFL`, `SIG_IGN`, *address*).
- Signal mask (see *sigvector*(2)).
- Profiling on/off status (see *profil*(2)).
- Command name in the accounting record (see *acct*(4)).
- Nice value (see *nice*(2)).
- All attached shared memory segments (see *shmop*(2)).
- Current working directory
- Root directory (see *chroot*(2)).
- File mode creation mask (see *umask*(2)).
- File size limit (see *ulimit*(2)).
- Real-time priority (see *rtprio*(2)).

Each of the child's file descriptors shares a common open file description with the corresponding file descriptor of the parent. This implies that changes to the file offset, file access mode, and file status flags of file descriptors in the parent also affect those in the child, and vice-versa.

The child process differs from the parent process in the following ways:

The child process has a unique process ID. The child process ID also does not match any active process group ID.

The child process has a different parent process ID (which is the process ID of the parent process).

The set of signals pending for the child process is initialized to the empty set.

The trace flag (see *ptrace*(2) `PT_SETTRC` request) is cleared in the child process.

The `AFORK` flag in the `ac_flags` component of the accounting record is set in the child process.

Process locks, text locks, and data locks are not inherited by the child (see *plock*(2)).

All `semadj` values are cleared (see *semop*(2)).

The child process's values of `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_cstime` are set to zero (see *times*(2)).

The time left until an alarm clock signal is reset to 0 (clearing any pending alarm), and all interval timers are set to 0 (disabled).

The *vfork*(2) system call can be used to fork processes more quickly than `fork()`, but has some restrictions. See *vfork*(2) for details.

If a parent and child process both have a file opened and the parent or child closes the file, the file is still open for the other process.

RETURN VALUE

Upon successful completion, `fork()` returns a value of `0` to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of -1 is returned to the parent process, no

child process is created, and `errno` is set to indicate the error.

The parent and child processes resume execution immediately after the `fork()` call; they are distinguished by the value returned by *fork*.

ERRORS

`fork()` fails and no child process is created if one or more of the following is true:

[EAGAIN] The system-imposed limit on the total number of processes under execution would be exceeded.

[EAGAIN] The system-imposed limit on the total number of processes under execution by a single user would be exceeded.

[ENOMEM] There is insufficient swap space and/or physical memory available in which to create the new process.

WARNINGS

Standard I/O streams (see *stdio*(3S)) are duplicated in the child. Therefore, if *fork* is called after a buffered I/O operation without first closing or flushing the associated standard I/O stream (see *fclose*(3S)), the buffered input or output might be duplicated.

AUTHOR

`fork()` was developed by AT&T, the University of California, Berkeley, and HP.

SEE ALSO

acct(2), chroot(2), exec(2), exit(2), fcntl(2), getgroups(2), lockf(2), nice(2), plock(2), profil(2), ptrace(2), rtprio(2), semop(2), setuid(2), setpgrp(2), shmop(2), times(2), ulimit(2), umask(2), vfork(2), wait(2), fclose(3S), stdio(3S), acct(4), signal(5).

STANDARDS CONFORMANCE

`fork()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

fsctl - file system control

SYNOPSIS

```
#include <sys/fsctl.h>

int fsctl(
    int fildes,
    int command,
    void *outbuf,
    size_t outlen
);
```

DESCRIPTION

`fsctl()` provides access to file-system-specific information. *fildes* is an open file descriptor for a file in the file system of interest. The possible values for *command* depend on the type of file system. Currently, defined *command*s exist only for the CDFS file system (see `sys/cdfsdir.h`).

outbuf is a pointer to the data area in which data is returned from the file system. *outlen* gives the length of the data area pointed to by *outbuf*.

The CDFS *command*s are:

`CDFS_DIR_REC` Returns the directory record for the file or directory indicated by *fildes*. The record is returned in a structure of type *cddir*, defined in `<sys/cdfsdir.h>`.

`CDFS_XAR` Returns the extended attribute record, if any, for the file or directory indicated by *fildes*. Because the size of an extended attribute record varies, be sure *outbuf* points to a data area of sufficient size. To find the necessary size, do the following:

1. Use *statfs*(2). to get the logical block size of the CDFS volume.
2. Use an `fsctl()` call with the `CDFS_DIR_REC` command to get the extended attribute record size (in blocks) for the file or directory of interest. The `mincdd_xar_len` field in the returned structure contains the size of the extended attribute record in logical blocks. (If this field is zero, the file or directory has no extended attribute record.)
3. Multiply `mincdd_xar_len` by the logical block size obtained in step 1 to get the total space needed.
4. Once you get the extended attribute record, cast *outbuf* into a pointer to a structure of type `cdxar_iso` (defined in `<sys/cdfsdir.h>`). This enables you to access those fields that are common to all extended attribute records. (See EXAMPLES below for an example of this process.)

   If the extended attribute record contains additional system use or application use data, that data will have to be accessed manually.

`CDFS_AFID` Returns the abstract file identifier for the primary volume whose root directory is specified by *fildes*, terminated with a NULL character. Note that the constant `CDMAXNAMLEN` defined in `<sys/cdfsdir.h>` gives the maximum length a file identifier can have. Thus, `CDMAXNAMLEN` + 1 can be used for *outlen* and the size of *outbuf*.

`CDFS_BFID` Returns the bibliographic file identifier for the primary volume whose root directory is specified by *fildes*, terminated with a NULL character. `CDMAXNAMLEN` + 1 can be used for the value of *outlen* and the size of *outbuf*.

`CDFS_CFID` Returns the copyright file identifier for the primary volume whose root directory is specified by *fildes*, terminated with a NULL character. `CDMAXNAMLEN` + 1 can be used for the value of *outlen* and the size of *outbuf*.

`CDFS_VOL_ID` Returns the volume ID for the primary volume specified by *fildes*, terminated with a NULL character. The maximum size of the volume ID is 32 bytes, so a length of 33 can be used for *outlen* and the size of *utbuf*.

CDFS_VOL_SET_ID
Returns the volume set ID for the primary volume specified by *fildes*, terminated with a NULL character. The maximum size of the volume set ID is 128 bytes, so a length of 129 can be used for *outlen* and the size of *outbuf*.

EXAMPLES

The following code fragment gets the extended attribute record for a file on a CDFS volume. The filename is passed in as the first argument to the routine. Note that error checking is omitted for brevity.

```
#include <sys/types.h>
#include <sys/vfs.h>
#include <fcntl.h>
#include <sys/cdfsdir.h>
main(argc, argv)
int argc;
char *argv[];
{
   int fildes, size = 0;
   char *malloc(), *outbuf;
   struct statfs buf;
   struct cddir cdrec;
   struct cdxar_iso *xar;
            .
            .
            .
   statfs(argv[1], &buf);   /* get logical block size */

   fildes = open(argv[1], O_RDONLY);  /* open file arg */

   /* get directory record for file arg */
   fsctl(fildes, CDFS_DIR_REC, &cdrec, sizeof(cdrec));

   size = buf.f_bsize * cdrec.cdd_min.mincdd_xar_len;   /* compute size */

   if(size) {   /* if size != 0 then there is an xar */
   outbuf = malloc(size);   /* malloc sufficient memory */

   fsctl(fildes, CDFS_XAR, outbuf, size);  /* get xar */

   xar = (struct cdxar_iso *)outbuf;  /* cast outbuf to access fields */
            .
            .
            .
   }
      .
      .
      .
}
```

RETURN VALUE

`fsctl()` returns the number of bytes read if successful. If an error occurs, -1 is returned and `errno` is set to indicate the error.

ERRORS

`fsctl()` fails if any of the following conditions are encountered:

[EBADF] *fildes* is not a valid open file descriptor.

[EFAULT] *outbuf* points to an invalid address.

[ENOENT] The requested information does not exist.

[EINVAL] *command* is not a valid command.

[EINVAL] *outlen* is negative, or *fildes* does not refer to a CDFS file system.

SEE ALSO

statfs(2), cdfs(4), cdfsdir(4), cdnode(4), cdrom(4).

NAME

fsync - synchronize a file's in-core state with its state on disk

SYNOPSIS

```
#include <unistd.h>

int fsync(int fildes);
```

DESCRIPTION

`fsync()` causes all modified data and attributes of *fildes* to be moved to a permanent storage device. This normally results in all in-core modified copies of buffers for the associated file to be written to a disk. `fsync()` applies to ordinary files, and applies to block special devices on systems which permit I/O to block special devices.

`fsync()` should be used by programs that require a file to be in a known state; such as when building a simple transaction facility.

RETURN VALUE

`fsync()` returns 0 on success or -1 if an error error occurs, and sets `errno` to indicate the error.

ERRORS

fsync fails if any of the following conditions are encountered:

[EBADF]	*fildes* is not a valid descriptor.
[EINVAL]	*fildes* refers to a file type to which `fsync()` does not apply.

WARNINGS

The current implementation of this function is inefficient for large files.

AUTHOR

`fsync()` was developed by the the University of California, Berkeley and HP.

SEE ALSO

fcntl(2), fcntl(5), open(2), select(2), sync(2), sync(1M).

STANDARDS CONFORMANCE

`fsync()`: AES, XPG3, XPG4

NAME

ftime - get date and time more precisely

SYNOPSIS

```
#include <sys/timeb.h>

int ftime(struct timeb *tp);
```

REMARKS

This facility is provided for backwards compatibility with Version 7 systems. Either `time()` or `gettimeofday()` should be used in new programs.

DESCRIPTION

`ftime()` fills in a structure pointed to by its argument, as defined by `<sys/timeb.h>`:

```
/*
 * Structure returned by ftime system call
 */
struct timeb {
    time_t time;
    unsigned short millitm;
    short  timezone;
    short  dstflag;
};
```

The structure contains the time in seconds since 00:00:00 UTC (Coordinated Universal Time), January 1, 1970, up to 1000 milliseconds of more-precise interval, the local timezone (measured in minutes of time westward from UTC), and a flag that, if nonzero, indicates that Daylight Saving time applies locally during the appropriate part of the year. Consult *gettimeofday*(2) for more details on the meaning of the timezone field.

This function can be accessed by giving the `-lV7` option to the `ld` command (see *ld*(1)).

`ftime()` can fail for exactly the same reasons as *gettimeofday*(2).

SEE ALSO

date(1), gettimeofday(2), stime(2), time(2), ctime(3C).

WARNINGS

The millisecond value usually has a granularity greater than one due to the resolution of the system clock. Depending on any granularity (particularly a granularity of one) renders code non-portable.

NAME

getaccess - get a user's effective access rights to a file

SYNOPSIS

```
#include <sys/getaccess.h>

int getaccess(
    const char *path,
    uid_t uid,
    int ngroups,
    const gid_t *gidset,
    void *label,
    void *privs
);
```

DESCRIPTION

getaccess() identifies the access rights (read, write, execute/search) a specific user ID has to an existing file. *path* points to a path name of a file. If the call succeeds, it returns a value of zero or greater, representing the specified user's effective access rights (modes) to the file. The rights are expressed as the logical OR of bits (**R_OK**, **W_OK**, and **X_OK**) whose values are defined in the header **<unistd.h>**. A return of zero means that access is denied.

The *uid* parameter is a user ID. Special values, defined in **<sys/getaccess.h>**, represent the calling process's effective, real, or saved user ID:

UID_EUID	Effective user ID.
UID_RUID	Real user ID.
UID_SUID	Saved user ID.

ngroups is the number of group IDs in *gidset*, not to exceed **NGROUPS_MAX** + 1 (**NGROUPS_MAX** is defined in **<limits.h>**). If the *ngroups* parameter is positive, the *gidset* parameter is an array of group ID values to use in the check. If *ngroups* is a recognized negative value, *gidset* is ignored. Special negative values of *ngroups*, defined in **<sys/getaccess.h>**, represent various combinations of the process's effective, real, or saved user ID and its supplementary groups list:

NGROUPS_EGID	Use process's effective group ID only.
NGROUPS_RGID	Use process's real group ID only.
NGROUPS_SGID	Use process's saved group ID only.
NGROUPS_SUPP	Use process's supplementary groups only.
NGROUPS_EGID_SUPP	Use process's effective group ID plus supplementary groups.
NGROUPS_RGID_SUPP	Use process's real group ID plus supplementary groups.
NGROUPS_SGID_SUPP	Use process's saved group ID plus supplementary groups.

The *label* and *privs* parameters are placeholders for future extensions. For now, the values of these parameters must be **(void *) 0**.

The access check rules for access control lists are described in *acl*(5). In addition, the **W_OK** bit is cleared for files on read-only file systems or shared-text programs being executed. Note that as in *access*(2), the **X_OK** bit is not turned off for shared-text programs open for writing because there is no easy way to know that a file open for writing is a shared-text program.

If the caller's user ID is 0, or if it is **UID_EUID**, **UID_RUID**, or **UID_SUID** (see **<sys/getaccess.h>**) and the process's respective user ID is 0, **R_OK** and **W_OK** are always set except when **W_OK** is cleared for files on read-only file systems or shared-text programs being executed. **X_OK** is set if and only if the file is not a regular file or the execute bit is set in any of the file's ACL entries.

getaccess() checks each directory component of *path* by first using the caller's effective user ID, effective group ID, and supplementary groups list, regardless of the user ID specified. An error occurs, distinct from "no access allowed," if the caller cannot search the path to the file. (In this case it is inappropriate for the caller to learn anything about the file.)

Comparison of *access*(2) **and** *getaccess*(2)

The following table compares various attributes of **access()** and **getaccess()**.

access()	getaccess()
checks all ACL entries	same
uses real uid, real gid, and supplementary groups list	uses specified uid and groups list; macros available for typical values
checks specific mode value, returns succeed or fail	returns all mode bits, each on or off
checks path to file using caller's effective IDs	same
W_OK false if shared-text file currently being executed	same
W_OK false if file on read-only file system	same
X_OK not modified for file currently open for writing	same
R_OK and W_OK always true for superuser (except as above)	same
X_OK always true for superuser	X_OK true for super-user if file is not a regular file *or* execute is set in any ACL entry

RETURN VALUE

Upon successful completion, **getaccess()** returns a non-negative value representing the access rights of the specified user to the specified file. If an error occurs, a value of **-1** is returned and **errno** is set to indicate the error.

ERRORS

getaccess() fails if any of the following conditions are encountered:

[EACCES] A component of the *path* prefix denies search permission to the caller.

[EFAULT] *path* or *gidset* points outside the allocated address space of the process. The reliable detection of this error is implementation dependent.

[EINVAL] *ngroups* is invalid; *ngroups* is either zero, an unrecognized negative value, or a value larger than **NGROUPS + 1**.

[EINVAL] *gidset* contains an invalid group ID value.

[EINVAL] The value of *label* or *privs* is not a null pointer.

[ELOOP] Too many symbolic links were encountered in translating the *path* name.

[ENAMETOOLONG] The length of the specified path name exceeds **PATH_MAX** bytes, or the length of a component of the path name exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect.

[ENOENT] The named file does not exist (for example, *path* is null or a component of *path* does not exist).

[ENOTDIR] A component of the *path* prefix is not a directory.

[EOPNOTSUPP] **getaccess()** is not supported on some types of remote files.

EXAMPLES

The following call determines the caller's effective access rights to file "test," and succeeds if the user has read access:

```
#include <unistd.h>
#include <sys/getaccess.h>

int mode;
mode = getaccess ("test", UID_EUID, NGROUPS_EGID_SUPP,
(int *) 0, (void *) 0, (void *) 0);

if ((mode >= 0) && (mode & R_OK)) ...
```

Here is one way to test access rights to file **/tmp/hold** for user ID **23**, group ID **109**:

```
int gid = 109;
int mode;

mode = getaccess ("/tmp/hold", 23, 1, & gid,
(void *) 0, (void *) 0);
```

Should the need arise, the following code builds a *gidset* that includes the process's effective group ID:

```
#include <limits.h>

int gidset [NGROUPS_MAX + 1];
int ngroups;

gidset [0] = getegid();
ngroups = 1 + getgroups (NGROUPS_MAX, & gidset [1]);
```

AUTHOR

`getaccess()` was developed by HP.

SEE ALSO

access(2), chmod(2), getacl(2), setacl(2), stat(2), acl(5), unistd(5).

NAME

getacl, fgetacl - get access control list (ACL) information

SYNOPSIS

```
#include <sys/acl.h>

int getacl(
    const char *path,
    int nentries,
    struct acl_entry *acl
);

int fgetacl(int fildes, int nentries, struct acl_entry *acl);
```

DESCRIPTION

`getacl()` returns a complete listing of all ACL entries (*uid.gid*, *mode*) in an existing file's access control list. *path* points to a path name of a file.

Similarly, `fgetacl()` returns a complete listing of all ACL entries for an open file known by the file descriptor *fildes*.

nentries is the number of entries being reported on, and is never more than the constant **NACLENTRIES** defined in `<sys/acl.h>`. If *nentries* is non-zero, it must be at least as large as the number of entries in the file's ACL, including base entries (see *setacl*(2)). `getacl()` returns the number of entries in the file's ACL, as well as the ACL entries themselves in the array of structures *acl* declared by the calling program.

If *nentries* is zero, `getacl()` returns the number of entries in the file's ACL, including base ACL entries, and *acl* is ignored.

Entries are reported in groups of decreasing order of specificity (see *setacl*(2)), then sorted in each group by user ID and group ID. The content of array entries beyond the number of defined entries for the file is undefined.

RETURN VALUE

Upon successful completion, `getacl()` and `fgetacl()` return a non-negative value. If an error occurs, a value of -1 is returned, and `errno` is set to indicate the error.

ERRORS

`getacl()` or `fgetacl()` fail to modify the *acl* array if any of the following is true:

[ENOTDIR] A component of the *path* prefix is not a directory.

[ENOENT] The named file does not exist (for example, *path* is null or a component of *path* does not exist).

[EBADF] *fildes* is not a valid file descriptor.

[EACCES] A component of the *path* prefix denies search permission.

[EFAULT] *path* or a portion of *acl* to be written points outside the allocated address space of the process.

[EINVAL] *nentries* is non-zero and less than the number of entries in the file's ACL, or it is greater than **NACLENTRIES**.

[EOPNOTSUPP] `getacl()` is not supported on remote files by some networking services.

[ENFILE] The system file table is full.

[ENAMETOOLONG] The length of *path* exceeds **PATH_MAX** bytes, or the length of a component of *path* exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect.

[ELOOP] Too many symbolic links were encountered in translating the *path* name.

EXAMPLES

The following call returns the number of entries in the ACL on file `/users/bill/mcfile`.

```
#include <sys/acl.h>
```

```
entries = getacl ("/users/bill/mcfile", 0, (struct acl_entry *) 0);
```

The following call returns in *acl* all entries in the ACL on the file opened with file descriptor 5.

```
#include <sys/acl.h>

int nentries;
struct acl_entry acl [NACLENTRIES];

entries = fgetacl (5, NACLENTRIES, acl);
```

DEPENDENCIES

NFS `getacl()` and `fgetacl()` are not supported on remote files.

AUTHOR

`getacl()` and `fgetacl()` were developed by HP.

SEE ALSO

access(2), chmod(2), getaccess(2), setacl(2), stat(2), unistd(5).

NAME

getaudid - get the audit ID (aid) for the current process

SYNOPSIS

```
#include <sys/audit.h>

int getaudid(void);
```

DESCRIPTION

`getaudid()` returns the audit ID (*aid*) for the current process. This call is restricted to the super-user.

RETURN VALUE

Upon successful completion, the audit ID is returned; otherwise, a `-1` is returned.

ERRORS

`getaudid()` fails if the following is true:

[EPERM] The caller is not super-user.

AUTHOR

`getaudid()` was developed by HP.

SEE ALSO

setaudid(2).

NAME

getaudproc - get the audit process flag for the calling process

SYNOPSIS

```
#include <sys/audit.h>

int getaudproc(void);
```

DESCRIPTION

`getaudproc()` returns the audit process flag for the calling process. The audit process flag (*u_audproc*) determines whether the process run by a given user should be audited. The process is audited if the returned flag is 1. If the returned flag is 0, the process is not audited. This call is restricted to the super-user.

RETURN VALUE

Upon successful completion, the audit process flag is returned; otherwise, a `-1` is returned and `errno` is set to indicate the error.

ERRORS

`getaudproc()` fails if the following is true:

[EPERM] The caller is not the super-user.

AUTHOR

`getaudproc()` was developed by HP.

SEE ALSO

setaudproc(2).

NAME

getcontext - return process context for context-dependent file search

SYNOPSIS

```
#include <unistd.h>

int getcontext(char *contextbuf, size_t length);
```

DESCRIPTION

`getcontext()` reads the per-process context (see *context*(5)) into the buffer pointed to by *contextbuf*. The context is returned as a null-terminated string containing a blank-separated list of names. The function value returned by `getcontext()` is the length of this string, including the null terminator. If this string, including the null terminator, is less than *length* bytes, a truncated, null-terminated string of *length* bytes is returned. In particular, if *length* is zero, only the function value is returned.

RETURN VALUE

Upon successful completion, the length of the context string, including the null terminator, is returned. Otherwise, a value of `-1` is returned and `errno` is set to indicate the error.

ERRORS

`getcontext()` may fail if the following is true:

[EFAULT] *contextbuf* points to an illegal address. Reliable detection of this error is not guaranteed.

EXAMPLES

In the following example `getcontext()` is called once with a *length* parameter of zero to determine the size of a buffer to allocate for the context.

```
int length;
char *contextbuf;

length = getcontext ((char *)0, 0);
contextbuf = malloc (length);
(void) getcontext (contextbuf, length);
```

AUTHOR

`getcontext()` was developed by HP.

SEE ALSO

getcontext(1), cnodeid(2), cnodes(2), cdf(4), context(5).

NAME

getdirentries - get entries from a directory in a filesystem-independent format

SYNOPSIS

```
#include <ndir.h>

int getdirentries(
    int fildes,
    struct direct *buf,
    size_t nbytes,
    off_t *basep
);
```

DESCRIPTION

`getdirentries()` places directory entries from the directory referenced by the file descriptor *fildes* into the buffer pointed to by *buf*, in a filesystem-independent format. Up to *nbytes* of data are transferred. *nbytes* must be greater than or equal to the block size associated with the file; see *stat*(2). Smaller block sizes can cause errors on certain file systems.

The data in the buffer consists of a series of `direct` structures, each containing the following entries:

```
unsigned long       d_fileno;
unsigned short      d_reclen;
unsigned short      d_namlen;
char                d_name[MAXNAMLEN + 1];
```

The `d_fileno` entry is a number unique for each distinct file in the file system. Files linked by hard links (see *link*(2)) have the same `d_fileno`. The `d_reclen` entry identifies the length, in bytes, of the directory record. The `d_name` entry contains a null-terminated file name. The `d_namlen` entry specifies the length of the file name. Thus the actual size of `d_name` can vary from 2 to `MAXNAMLEN` + 1. Note that the `direct` structures in the buffer are not necessarily tightly packed. The `d_reclen` entry must be used as an offset from the beginning of a `direct` structure to the next structure, if any.

The return value of the system call is the actual number of bytes transferred. The current position pointer associated with *fildes* is set to point to the next block of entries. The pointer is not necessarily incremented by the number of bytes returned by `getdirentries()`. If the value returned is zero, the end of the directory has been reached.

The current position pointer is set and retrieved by `lseek()` (see *lseek*(2). `getdirentries()` writes the position of the block read into the location pointed to by *basep*. The current position pointer can be set safely only to a value previously returned by `lseek()`, to a value previously returned in the location pointed to by *basep*, or to zero. Any other manipulation of the position pointer causes undefined results.

RETURN VALUE

If successful, the number of bytes actually transferred is returned. Otherwise, `-1` is returned and `errno` is set to indicate the error.

ERRORS

`getdirentries()` fails if any of the following conditions are encountered:

[EBADF]	*fildes* is not a valid file descriptor open for reading.
[EFAULT]	Either *buf* or *basep* points outside the allocated address space.
[EINTR]	A read from a slow device was interrupted by the delivery of a signal before any data arrived.
[EIO]	An I/O error occurred while reading from or writing to the file system.

AUTHOR

`getdirentries()` was developed by Sun Microsystems, Inc.

SEE ALSO

open(2), lseek(2).

NAME

getdomainname, setdomainname - get/set name of current Network Information Service domain

SYNOPSIS

```
int getdomainname(char *name, int namelen);
int setdomainname(char *name, int namelen);
```

DESCRIPTION

getdomainname() returns the name of the Network Information Service (NIS) domain for the current processor, as previously set by **setdomainname()**. The parameter *namelen* specifies the size of the *name* array. The returned value is null-terminated unless the area pointed to by *name* is not large enough to hold the domain name plus the null byte. In this case, only the *namelen* number of bytes is returned.

setdomainname() sets the domain of the host machine to *name*, which has a length of *namelen*. This call is restricted to the super-user and is normally used only when the system is booted.

These Network Information Service domains enable two distinct networks with common host names to merge. Each network is distinguished by having a different domain name. Currently, only the Network Information Service uses these domains.

RETURN VALUE

If the call succeeds, a value of 0 is returned. If the call fails, a value of -1 is returned and **errno** is set to indicate the error.

ERRORS

getdomainname() and **setdomainname()** fail if any of the following conditions are encountered:

[EFAULT]	*name* points outside the accessible address space.
[EPERM]	The caller is not super-user. This error only applies to **setdomainname()**.

WARNINGS

The length of the *name* array should be at least 65; NIS domain names can be up to 64 characters long.

NIS servers use the NIS domain name as the name of a subdirectory of **/usr/etc/yp**. Since the NIS domain name can be as long as 64 characters, the domain name set with **setdomainname()** can exceed the maximum file name length allowed on the local file system. If that length is exceeded, the name of the subdirectory is the truncated NIS domain name.

AUTHOR

getdomainname was developed by Sun Microsystems, Inc.

SEE ALSO

domainname(1), ypserv(1M), ypfiles(4).

NAME

getevent - get events and system calls that are currently being audited

SYNOPSIS

```
#include <sys/audit.h>

int getevent(
    struct aud_type *a_syscall,
    struct aud_event_tbl *a_event
);
```

DESCRIPTION

getevent() gets the events and system calls being audited. The events are returned in a table pointed to by *a_event*. The system calls are returned in a table pointed to by *a_syscall*. This call is restricted to the super-user.

RETURN VALUE

Upon successful completion, a value of 0 is returned; otherwise, a -1 is returned and **errno** is set to indicate the error.

ERRORS

getevent() fails if the following is true:

[EPERM] The caller is not super-user.

AUTHOR

getevent() was developed by HP.

SEE ALSO

setevent(2), audevent(1M).

NAME

getfh - return file handle for file on remote node

SYNOPSIS

```
#include <time.h>
#include <rpc/rpc.h>
#include <errno.h>
#include <nfs/nfs.h>

int getfh(char *path, fhandle_t *fhp);
```

DESCRIPTION

getfh() returns a file handle in the struct pointed to by *fhp* for the file pointed to by *path*. This information is used to perform an NFS mount for a remote node. **getfh()** is executed on the remote node; results are passed back to the program doing the NFS mount. The caller should never examine the file handle contents. The file handle only identifies a file to the node that produced the file handle. (The term "file handle" refers to an NFS concept.)

Only the super-user can invoke **getfh()**.

RETURN VALUE

Upon successful completion, **getfh()** returns 0; otherwise it returns −1 and sets **errno** to indicate the error.

ERRORS

getfh() fails if any of the following conditions are encountered:

[EPERM]	The effective user ID is not super-user.
[ENOENT]	File or directory specified by *path* does not exist.
[EINVAL]	Invalid argument, or the file or directory has not been exported by **exportfs** (see *exportfs*(1M)).
[EREMOTE]	The file or directory specified by *path* is a remote file or directory.

WARNINGS

This call should be used only by HP-supplied commands and is not recommended for use by non-HP-supplied programs.

AUTHOR

Sun Microsystems, Inc.

SEE ALSO

exportfs(1M), mount(1M), vfsmount(2).

NAME

getgroups - get group access list

SYNOPSIS

```
#include <unistd.h>

int getgroups(int ngroups, gid_t gidset[]);
```

DESCRIPTION

`getgroups()` gets the current group access list of the user process and stores it in the array *gidset*. The parameter *ngroups* indicates the number of entries which may be placed in *gidset*. No more than `NGROUPS`, as defined in `<sys/param.h>`, is ever returned.

As a special case, if the *ngroups* argument is zero, `getgroups()` returns the number of group entries for the process. In this case, the array pointed to by the *gidset* argument is not modified.

EXAMPLES

The following call to *getgroups*(2) retrieves the group access list of the calling process and stores the group ids in array mygidset:

```
int ngroups = NGROUPS;
gid_t mygidset[NGROUPS];
int ngrps;

ngrps = getgroups (ngroups, mygidset);
```

RETURN VALUE

If successful, `getgroups()` returns a non-negative value indicating the number of elements returned in *gidset*. If an error occurs, a value of -1 is returned and `errno` is set to indicate the type of error.

ERRORS

`getgroups()` fails if any of the following conditions are encountered:

[EFAULT] *gidset* specifies an invalid address. The reliable detection of this error is implementation dependent.

[EINVAL] The argument *ngroups* is not zero and is less than the number of groups in the current group access list of the process.

AUTHOR

`getgroups()` was developed by HP and the University of California, Berkeley

SEE ALSO

setgroups(2), initgroups(3C)

STANDARDS CONFORMANCE

`getgroups()`: AES, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

gethostname - get name of current host

SYNOPSIS

```
#include <unistd.h>

int gethostname(char *hostname, size_t size);
```

DESCRIPTION

`gethostname()` returns in the array to which *hostname* points, the standard host name for the current processor as set by `sethostname()` (see *sethostname*(2)). *size* specifies the length of the *hostname* array. *hostname* is null-terminated unless insufficient space is provided.

RETURN VALUE

`gethostname()` returns 0 if successful. Otherwise, it returns -1 and sets `errno` to indicate the error.

ERRORS

`gethostname()` can fail if the following is true:

[EFAULT] *hostname* points to an illegal address. The reliable detection of this error is implementation dependent.

AUTHOR

`gethostname()` was developed by the University of California, Berkeley.

SEE ALSO

hostname(1), uname(1), sethostname(2), uname(2).

NAME

getitimer, setitimer - get/set value of interval timer

SYNOPSIS

```
#include <time.h>

int getitimer(int which, struct itimerval *value);

int setitimer(
    int which,
    const struct itimerval *value,
    struct itimerval *ovalue
);
```

DESCRIPTION

The system provides each process with three interval timers, defined in <time.h>. `getitimer()` returns the current value for the timer specified in *which*, whereas `setitimer()` call sets the value of a timer (optionally returning the previous value of the timer).

A timer value is defined by the *itimerval* structure:

```
struct itimerval {
   struct timeval   it_interval;    /* timer interval */
   struct timeval   it_value;       /* current value */
};
```

If *it_value* is non-zero, it indicates the time to the next timer expiration. If *it_interval* is non-zero, it specifies a value to be used in reloading *it_value* when the timer expires. Setting *it_value* to 0 disables a timer. Setting *it_interval* to 0 causes a timer to be disabled after its next expiration (assuming *it_value* is non-zero).

Time values smaller than the resolution of the system clock are rounded up to this resolution. The machine-dependent clock resolution is 1/HZ seconds, where the constant HZ is defined in <sys/param.h>. Time values larger than an implementation-specific maximum value are rounded down to this maximum. The maximum values for the three interval timers are specified by the constants MAX_ALARM, MAX_VTALARM, and MAX_PROF defined in <sys/param.h>. On all implementations, these values are guaranteed to be at least 31 days (in seconds).

The *which* parameter specifies which timer to use. The possible values are ITIMER_REAL, ITIMER_VIRTUAL, and ITIMER_PROF.

The ITIMER_REAL timer decrements in real time. A SIGALRM signal is delivered when this timer expires.

The ITIMER_VIRTUAL timer decrements in process virtual time. It runs only when the process is executing. A SIGVTALRM signal is delivered when it expires.

The ITIMER_PROF timer decrements both in process virtual time and when the system is running on behalf of the process. It is designed to be used by interpreters in statistically profiling the execution of interpreted programs. Each time the ITIMER_PROF timer expires, the SIGPROF signal is delivered. Since this signal can interrupt in-progress system calls, programs using this timer must be prepared to restart interrupted system calls.

Interval timers are not inherited by a child process across a `fork()`, but are inherited across an `exec()`.

Three macros for manipulating time values are defined in <time.h>:

`timerclear`	Set a time value to zero.
`timerisset`	Test if a time value is non-zero.
`timercmp`	Compare two time values. (Beware that >= and <= do not work with the `timercmp` macro.)

The timer used with ITIMER_REAL is also used by `alarm()` (see *alarm*(2)). Thus successive calls to `alarm()`, `getitimer()`, and `setitimer()` set and return the state of a single timer. In addition, a call to `alarm()` sets the timer interval to zero.

RETURN VALUE

If the calls succeed, a value of 0 is returned. If an error occurs, -1 is returned, and `errno` is set to indicate the error.

ERRORS

`getitimer()` or `setitimer()` fail if any of the following conditions are encountered:

[EFAULT] The *value* structure specified a bad address. Reliable detection of this error is implementation dependent.

[EINVAL] A *value* structure specified a microsecond value less that zero or greater than or equal to one million.

[EINVAL] *which* does not specify one of the three possible timers.

EXAMPLES

The following call to `setitimer()` sets the real-time interval timer to expire initially after 10 seconds and every 0.5 seconds thereafter:

```
struct itimerval rttimer;
struct itimerval old_rttimer;

rttimer.it_value.tv_sec     = 10;
rttimer.it_value.tv_usec    = 0;
rttimer.it_interval.tv_sec  = 0;
rttimer.it_interval.tv_usec = 500000;

setitimer (ITIMER_REAL, &rttimer, &old_rttimer);
```

AUTHOR

`getitimer()` was developed by the University of California, Berkeley.

SEE ALSO

alarm(2), exec(2), gettimeofday(2), signal(5).

NAME

getpeername - get address of connected peer

SYNOPSIS

```
#include <sys/socket.h>
```

AF_CCITT only:

```
#include <x25/x25addrstr.h>
```

```
int getpeername(int s, void *addr, int *addrlen);
```

DESCRIPTION

getpeername( ) returns the address of the peer socket connected to the socket indicated by *s*, where *s* is a socket descriptor. *addr* points to a socket address structure in which this address is returned. *addrlen* points to an object of type **int**, which should be initialized to indicate the size of the address structure. On return, it contains the actual size of the address returned (in bytes). If *addr* does not point to enough space to contain the whole address of the peer, only the first *addrlen* bytes of the address are returned.

AF_CCITT only:

The *addr* struct contains the X.25 addressing information of the *remote* peer socket connected to socket *s*. However, the **x25ifname[]** field of the *addr* struct contains the name of the *local* X.25 interface through which the call arrived.

RETURN VALUE

Upon successful completion, **getpeername( )** returns 0; otherwise it returns -1 and sets **errno** to indicate the error.

ERRORS

getpeername( ) fails if any of the following conditions are encountered:

[EBADF]	The argument *s* is not a valid file descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOTCONN]	The socket is not connected.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[EFAULT]	The *addr* or *addrlen* parameters are not valid pointers.
[EINVAL]	The socket has been shut down.
[EOPNOTSUPP]	Operation not supported for AF_UNIX sockets.

AUTHOR

getpeername( ) was developed by the University of California, Berkeley.

SEE ALSO

bind(2), socket(2), getsockname(2), inet(7F), af_ccitt(7F).

NAME

getpid, getpgrp, getppid, getpgrp2 - get process, process group, and parent process ID

SYNOPSIS

```
#include <unistd.h>

pid_t getpid(void);

pid_t getpgrp(void);

pid_t getppid(void);

pid_t getpgrp2(pid_t pid);
```

DESCRIPTION

The following functions return the information indicated:

`getpid()`	Process ID of the calling process.
`getpgrp()`	Process group ID of the calling process.
`getppid()`	Parent process ID of the calling process.
`getpgrp2()`	Process group ID of the specified process. If *pid* is zero, the call applies to the current process. For this to be allowed, the current process and the referenced process must be in the same session.

ERRORS

getpgrp2 fails if any of the following conditions are encountered:

[EPERM]	The current process and the specified process are not in the same session.
[ESRCH]	No process can be found corresponding to that specified by *pid*.

AUTHOR

`getpid()`, `getppid()`, `getpgrp()`, and `getpgrp2()` were developed by HP, AT&T, and the University of California, Berkeley.

SEE ALSO

exec(2), fork(2), setpgrp(2), setpgid(2), signal(5).

STANDARDS CONFORMANCE

`getpid()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

`getpgrp()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

`getppid()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

getpriority, setpriority - get and set process priorities

SYNOPSIS

```
#include <sys/resource.h>

int getpriority(int which, int who);

int setpriority(int which, int who, int priority);
```

DESCRIPTION

`getpriority()` returns the priority of the indicated processes.

`setpriority()` sets the priority of the indicated processes to *priority*.

The processes are indicated by *which* and *who*, where *which* can have one of the following values:

`PRIO_PROCESS` Get or set the priority of the specified process where *who* is the process ID. A *who* of 0 implies the process ID of the calling process.

`PRIO_PGRP` Get or set the priority of the specified process group where *who* is the process-group ID, indicating all processes belonging to that process-group. A *who* of 0 implies the process-group ID of the calling process.

`PRIO_USER` Get or set the priority of the specified user where *who* is the user ID, indicating all processes owned by that user. A *who* of 0 implies the user ID of the calling process.

If more than one process is indicated, the priority returned by `getpriority()` is the smallest valued priority of all the indicated processes, and `setpriority()` sets the priority of all indicated processes.

priority is a value between -20 and 20, where smaller values indicate better priorities. The default priority for a processes is 0, and negative priorities require appropriate privileges.

RETURN VALUE

On success, `getpriority()` returns an integer in the range from –20 to 20, and `setpriority()` returns 0. Otherwise, both return –1 and set `errno` to indicate the error. See WARNINGS below.

ERRORS

`getpriority()` and `setpriority()` fail if any of the following conditions are encountered:

[ESRCH] Processes indicated by *which* and *who* cannot be found.

[EINVAL] *which* is not one of the choices listed above.

[EACCES] The calling process does not have access rights to change one or more of the indicated processes. All processes for which access is allowed are still affected.

[EPERM] The calling process attempted to change the priority of a process to a negative value without having appropriate privileges.

WARNINGS

Note that `getpriority()` can return –1 when it successfully finds a priority of –1, and when it fails. To determine whether a failure occurred, set `errno` to 0 before calling `getpriority()` then examine `errno` after the call returns.

AUTHOR

`setpriority()` and `getpriority()` were developed by the University of California, Berkeley.

SEE ALSO

nice(1), renice(1), nice(2).

NAME

getprivgrp, setprivgrp - get and set special attributes for group

SYNOPSIS

```
#include <sys/privgrp.h>

int getprivgrp(struct privgrp_map *grplist);

int setprivgrp(gid_t grpid, const int *mask);
```

DESCRIPTION

setprivgrp() associates a kernel capability with a group ID. This allows subsetting of super-user-like privileges for members of a particular group or groups. **setprivgrp()** takes two arguments: the integer group id and a mask of permissions. The mask is created by treating the access types defined in **<sys/privgrp.h>** as bit numbers (using 1 for the least significant bit). Thus, privilege number 5 would be represented by the bit 1<<(5-1) or 16. More generally, privilege **p** is represented by:

mask[(**p**-1) / BITS_PER_INT)] & (1 << ((**p**-1) % BITS_PER_INT)).

As it is possible to have more than **word size** distinct privileges, mask is a pointer to an integer array of size **PRIV_MASKSIZ**.

setprivgrp() privileges include those specified in the file **<sys/privgrp.h>**. A process can access the system call protected by a specific privileged group if it belongs to or has an effective group ID of a group having access to the system call. All processes are considered to belong to the pseudo-group **PRIV_GLOBAL**.

Specifying a *grpid* of **PRIV_NONE** causes privileges to be revoked on all privileged groups having any of the privileges specified in *mask*. Specifying a *grpid* of **PRIV_GLOBAL** causes privileges to be granted to all processes.

The constant **PRIV_MAXGRPS** in **<sys/privgrp.h>** defines the system limit on the number of groups that can be assigned privileges. One of these is always the psuedo-group **PRIV_GLOBAL** , allowing for **PRIV_MAXGRPS** - 1 actual groups.

getprivgrp() returns a table of the privileged group assignments into a user supplied structure. *grplist* points to an array of structures of type **privgrp_map** associating a groupid with a privilege mask. Privilege masks are formed by ORing together elements from the access types specified in **<sys/privgrp.h>**. The array may have gaps in it distinguished as having a **priv_groupno** field of **PRIV_NONE**. The group number **PRIV_GLOBAL** gives the global privilege mask. Only information about groups which are in the user's group access list, or about his real or effective group id, is returned to an ordinary user. The complete set is returned to the privileged user.

EXAMPLES

The following example prints out **PRIV_GLOBAL** and the group IDs of the privilege groups to which the user belongs:

```
#include <sys/types.h>

struct privgrp_map pgrplist[PRIV_MAXGRPS];
int i;
gid_t pgid;

getprivgrp (pgrplist);
for (i=0; i<PRIV_MAXGRPS; i++) {
   if ((pgid = pgrplist[i].priv_groupno) != PRIV_NONE) {
      if (pgid == PRIV_GLOBAL)
          printf ("(PRIV_GLOBAL) ");
       printf ("privilege group id = %d\n", pgid);
   }
}
```

NOTES

Only users with the #idfef B1 **setprocident** privilege

NAME

getrlimit, setrlimit - control consumption of system resources

SYNOPSIS

```
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlp);

int setrlimit(int resource, const struct rlimit *rlp);
```

DESCRIPTION

setrlimit() sets a limit on consumption of system resources by the current process and each process it creates. **getrlimit()** is used to obtain the value of the current limit.

Each call to either **getrlimit()** or **setrlimit()** identifies a specific resource to be operated upon as well as a resource limit. A resource limit is a pair of values: one specifying the current (soft) limit, the other a maximum (hard) limit. Soft limits can be changed by a process to any value that is less than or equal to the hard limit. A process can irreversibly lower its hard limit to any value that is greater than or equal to the soft limit. Only users with appropriate privileges can raise a hard limit. Both hard and soft limits can be changed in a single call to **setrlimit()**, subject to the constraints described above.

The *resource* parameter selects the system resource limits to be set or retrieved. The possible values for *resource* are defined in <**sys/resource.h**>. Currently, only the following values are supported:

RLIMIT_NOFILE the maximum number of files a process can have open. The soft limit for this resource is the same as the value returned by **sysconf(_SC_OPEN_MAX)**.

RLIMIT_OPEN_MAX defined to be the same as **RLIMIT_NOFILE**.

The *rlp* argument points to an object of type struct rlimit, which is defined in <**sys/resource.h**>, and includes the following members:

```
int rlim_cur      Current (soft) limit
int rlim_max      Hard limit
```

For **getrlimit()**, the system stores the two limits on the specified resource in the structure to which *rlp* points.

For **setrlimit()**, the system reads new values for the two limits on the specified resource from the structure to which *rlp* points.

RETURN VALUE

Upon successful completion, **getrlimit()** and **setrlimit()** return a value of 0. Otherwise, a value of -1 is returned, the limits on the *resource* and the *rlp* structure are unchanged, and **errno** is set to indicate the error.

ERRORS

getrlimit() and **setrlimit()** fail if:

[EFAULT] The address specified for *rlp* is invalid. Reliable detection of this error is implementation dependent.

[EINVAL] The number specified for *resource* is invalid.

setrlimit fails if:

[EPERM] The *rlp* argument specified a hard or soft limit higher than the current hard limit value, and the caller does not have appropriate privileges.

[EINVAL] A user with appropriate privileges has attempted to raise *rlp->rlim_cur* or *rlp->rlim_max* to a value greater than the system is capable of supporting.

[EINVAL] The value of *rlp->rlim_cur* is less than the number of file descriptors the process already has allocated.

[EINVAL] The value of *rlp->rlim_max* is less than the current soft limit.

AUTHOR

`getrlimit()` and `setrlimit()` were developed by HP, AT&T, and the University of California, Berkeley.

SEE ALSO

sysconf(2).

NAME

getsockname - get socket address

SYNOPSIS

```
#include <sys/socket.h>
```

AF_CCITT only:

```
#include <x25/x25addrstr.h>
```

```
int getsockname(int s, void *addr, int *addrlen);
```

DESCRIPTION

`getsockname()` returns the address of the socket indicated by *s*, where *s* is a socket descriptor. *addr* points to a socket address structure in which this address is returned. *addrlen* points to an `int` which should be initialized to indicate the size of the address structure. On return it contains the actual size of the address returned (in bytes). If *addr* does not point to enough space to contain the whole address of the socket, only the first *addrlen* bytes of the address are returned.

AF_CCITT only:

The `x25_host[]` field of the *addr* struct returns the X.25 addressing information of the local socket *s*. The `x25ifname[]` field of the *addr* struct contains the name of the local X.25 interface through which the call arrived.

RETURN VALUE

Upon successful completion, `getsockname()` returns 0; otherwise, it returns -1 and sets `errno` to indicate the error.

ERRORS

`getsockname()` fails if any of the following conditions are encountered:

[EBADF]	*s* is not a valid descriptor.
[ENOTSOCK]	*s* is a file, not a socket.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[EFAULT]	The *addr* or *addrlen* parameters are not valid pointers.
[EINVAL]	The socket has been shut down.
[EOPNOTSUPP]	Operation not supported for AF_UNIX sockets.

AUTHOR

`getsockname()` was developed by the University of California, Berkeley.

SEE ALSO

bind(2), socket(2), getpeername(2), inet(7F), af_ccitt(7F).

NAME

getsockopt, setsockopt - get and set options on sockets

SYNOPSIS

```
#include <sys/socket.h>

int getsockopt(
    int s,
    int level,
    int optname,
    void *optval,
    int *optlen);

int setsockopt(
    int s,
    int level,
    int optname,
    const void *optval,
    int optlen);
```

DESCRIPTION

getsockopt() and **setsockopt()** manipulate options associated with a socket. The socket is identified by the socket descriptor *s*. Options can exist at multiple protocol levels, and they are always present at the uppermost "socket" level (see *socket*(2)).

When manipulating socket options, the level at which the option resides (*level*) and the name of the option (*optname*) must be specified. To manipulate options at the "socket" level, *level* is specified as **SOL_SOCKET**.

There are two kinds of options: boolean and non-boolean. Boolean options are either set or not set and also can use *optval* and *optlen* (see below) to pass information. Non-boolean options always use *optval* and *optlen* to pass information.

To determine whether boolean option *optname* is set, the return value of **getsockopt()** must be examined. If the option is set, **getsockopt()** returns without error. If the boolean option is not set, **getsockopt()** returns –1 and **errno** is set to indicate the error.

For **setsockopt()**, the parameters *optval* and *optlen* are used to pass option information from the system to the calling process. *optval* is the address of a location in memory that contains the option information to be passed to the system. *optlen* is an integer that specifies the size in bytes of the option information.

For **getsockopt()**, *optval* and *optlen* are used to pass option information from the system to the calling process. *optval* is the address of a location in memory that contains the option information to be passed to the calling process, or (char *) NULL if the option information is not of interest and not to be passed to the calling process. *optlen* is an address of an integer initially used to specify the maximum number of bytes of option information to be passed. If *optval* is not (char *) NULL, *optlen* is set on return to the actual number of bytes of option information passed. If the **getsockopt()** call fails, no option information is passed.

optname and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file **<sys/socket.h>** contains definitions for "socket" level options (see *socket*(2)). Options at other protocol levels vary in format and name. Consult the appropriate entries in Section 7P, such as *tcp*(7P).

The "socket" level options defined in the include file **<sys/socket.h>** are explained below:

SO_DEBUG	(boolean option) no functionality; included only for compatibility.
SO_DONTROUTE	(boolean option; SOCK_STREAM sockets only) causes outgoing messages to bypass standard routing facilities and to be routed by the network portion of the Internet address.
SO_ERROR	returns the current contents of the variable *so_error* for this socket and then clears the variable (*so_error* is defined in **<sys/socketvar.h>**. The contents match those found in **errno**.

SO_REUSEADDR	(boolean option; AF_INET sockets only) allows local address reuse.
SO_KEEPALIVE	(boolean option; SOCK_STREAM and AF_INET <<<<<<< getsockopt.2 sockets only) keeps otherwise idle connections active. If a connection has been idle for two hours, transmissions are forced every 75 seconds until a response is received or 10 minutes expires, whichever occurs first. If 10 minutes expires with no response, the connection is dropped.
SO_LINGER	(boolean option; SOCK_STREAM and AF_INET sockets only) lingers on close if data is present. For **SO_LINGER**, *optval* points to a **struct linger**, defined in **/usr/include/sys/socket.h**. The linger structure contains an integer boolean flag to toggle behavior on/off and an integer linger value.
SO_BROADCAST	(boolean option; SOCK_DGRAM and AF_INET sockets only) toggles permission to transmit broadcast messages.
SO_RCVBUF	(non-boolean option) For stream sockets it changes the buffer size of a socket's receive socket buffer. For datagram sockets it changes the maximum size message a socket can receive. A stream socket's buffer size can be increased at any time but decreased only prior to establishing a connection. For datagram sockets, the inbound maximum message size can be increased or decreased at any time. The default and maximum values for **SO_RCVBUF** are protocol-specific. Refer to the appropriate entries in Sections 7F and 7P.
SO_SNDBUF	(non-boolean option) For stream sockets, it changes the buffer size of a socket's send socket buffer. For datagram sockets it changes the maximum size message that can be sent. A stream socket's buffer size can be increased at any time but decreased only prior to establishing a connection. For datagram sockets, the maximum outbound message size can be increased or decreased at any time. The default and maximum values for **SO_SNDBUF** are protocol-specific. Refer to the appropriate entries in Sections 7F and 7P.
SO_USELOOPBACK	(boolean option) no functionality; included only for compatibility.

None of the boolean options are supported for SOCK_DGRAM sockets.

If **SO_DONTROUTE** is set, the system does not use the network routing tables when determining which interface to use to send an outbound message. Instead, the system sends the message out through the interface that has a configured address matching the address to which the message is intended to be sent. If **SO_DONTROUTE** is not set, the system uses the network routing tables.

SO_REUSEADDR indicates the rules used in validating addresses supplied in a **bind()** call should allow reuse of local addresses. This allows multiple SOCK_STREAM sockets to be bound to the same local address, as long as all existing sockets at the desired address are in a connected state before the **bind()** is done on the new socket. The **SO_REUSEADDR** option has no effect on SOCK_DGRAM sockets.

The **SO_KEEPALIVE** option defaults to off. If **SO_KEEPALIVE** is set on and the connection has been idle for two hours, TCP sends a packet to the remote socket to acknowledge that it is still alive. If the remote socket does not respond within 75 seconds, TCP sends another packet. If TCP sends a total of 8 packets without response from the remote socket (i.e., 10 minutes have passed), TCP drops the connection. The next socket call (e.g., **recv()**) returns an error, and **errno** is set to ETIMEDOUT.

SO_LINGER controls the actions taken when unsent messages are queued on a SOCK_STREAM socket and a *close*(2) is performed. If **SO_LINGER** is toggled on with a non-zero linger interval, the system blocks the process on the **close()** attempt until it is able to transmit the data or until it decides it is unable to deliver the information. If **SO_LINGER** is toggled on with a linger interval of zero, the connection is immediately terminated on the **close()** of the socket, and any unsent data queued on the connection is lost. If **SO_LINGER** is toggled off (default upon socket creation) and a **close()** is issued, the call returns immediately. The system still gracefully brings down the connection by transmitting any queued data, if possible. **SO_LINGER** can be toggled on/off at any time during the life of an established connection. Toggling **SO_LINGER** does not affect the action of **shutdown()**.

The **SO_BROADCAST** option requests permission to send Internet broadcast datagrams on the socket.

For stream sockets, **SO_RCVBUF** and **SO_SNDBUF** can be used with **getsockopt()** to find the current sizes (in number of bytes) of the socket's receive and send buffers, respectively. If supported by the protocol, **SO_RCVBUF** and **SO_SNDBUF** can also be used with **setsockopt()** to set the sizes (in number of bytes) of the socket's receive and send buffers, respectively. The sizes are passed as integer values using *optval* and *optlen*. You can increase a socket's buffer size at any time, but you can decrease it only prior to establishing a connection. The default and maximum buffer sizes are protocol-specific. See the appropriate entries in Sections 7F and 7P for more information.

For datagram sockets, **SO_RCVBUF** and **SO_SNDBUF** can be used with **getsockopt()** to find the current maximum datagram size (in number of bytes) in the inbound and outbound direction, respectively. **SO_RCVBUF** and **SO_SNDBUF** can also be used with **setsockopt()** to set the maximum datagram size. The default and maximum datagram sizes are protocol-specific. See the appropriate entries in Sections 7F and 7P for more information.

AF_CCITT

SO_SNDBUF and **SO_RCVBUF** are the only options supported for sockets of the AF_CCITT address family.

RETURN VALUE

Upon successful completion, **getsockopt()** and **setsockopt()** return 0; otherwise, they return -1 and set **errno** to indicate the error.

DIAGNOSTICS

getsockopt() and **setsockopt()** fail if any of the following conditions are encountered:

[EBADF]	The argument *s* is not a valid descriptor.
[EOPNOTSUPP]	The option is not supported by the protocol in use by the socket.
[ENOBUFS]	No buffer space is available.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOPROTOOPT]	In **getsockopt()**, the requested option is currently not set.
[EINVAL]	The option is unknown at the socket level or the socket has been shut down.
[EFAULT]	The *optval* or, in the case of **getsockopt()**, *optlen* parameters are not valid pointers.

AUTHOR

getsockopt() was developed by the University of California, Berkeley.

SEE ALSO

socket(2), getprotoent(3N), af_ccitt(7F), tcp(7P), udp(7P), unix(7P).

NAME

gettimeofday, settimeofday - get/set date and time

SYNOPSIS

```
#include <time.h>

int gettimeofday(
    struct timeval *tp,
    struct timezone *tzp
);

int settimeofday(
    const struct timeval *tp,
    const struct timezone *tzp
);
```

DESCRIPTION

`gettimeofday()` returns and `settimeofday()` sets the system's notion of the current Coordinated Universal Time (UTC) and the system's notion of the current time zone. Time is expressed in seconds and microseconds since midnight January 1, 1970.

The structures pointed to by *tp* and *tzp* are defined in `<time.h>` as:

```
struct timeval {
   unsigned long   tv_sec;     /* seconds since Jan. 1, 1970 */
   long            tv_usec;    /* and microseconds */
};

struct timezone {
   int     tz_minuteswest;     /* of UTC */
   int     tz_dsttime;         /* type of DST correction to apply */
};
```

The `timezone` structure indicates the local time zone (measured in minutes of time westward from UTC), and a flag that, if nonzero, indicates that Daylight Savings Time applies locally during the appropriate part of the year. Programs should use this timezone information only in the absence of the TZ environment variable.

Only users with appropriate privileges can set the time of day.

EXAMPLES

The following example calls `gettimeofday()` twice. It then computes the lapsed time between the calls in seconds and microseconds and stores the result in a timeval structure:

```
struct timeval          first,
                        second,
                        lapsed;
struct timezone tzp;

   gettimeofday (&first, &tzp);

/* lapsed time */

   gettimeofday (&second, &tzp);

if (first.tv_usec > second.tv_usec) {

   second.tv_usec += 1000000;
   second.tv_sec--;

}
lapsed.tv_usec = second.tv_usec - first.tv_usec;
lapsed.tv_sec = second.tv_sec - first.tv_sec;
```

RETURN VALUE

`gettimeofday()` and `settimeofday()` return 0 on success; otherwise, if an error occurs, they return -1 and set `errno` to indicate the error.

ERRORS

`gettimeofday()` and `settimeofday()` fail if any of the following conditions are encountered:

[EFAULT] An argument address referenced invalid memory. The reliable detection of this error will be implementation dependent.

[EPERM] A user lacking appropriate privileges attempted to set the time.

WARNINGS

The microsecond value usually has a granularity much greater than one due to the resolution of the system clock. Relying on any granularity (particularly of one) will render code non-portable.

DEPENDENCIES

Series 300/400

`gettimeofday()` has a granularity of 4 microseconds.

Clustered Systems

In an HP Clustered Environment, setting the time of day sets the date and timezone on all systems in the cluster.

AUTHOR

`gettimeofday()` was developed by the University of California, Berkeley.

SEE ALSO

date(1), stime(2), time(2), ctime(3C), privilege(5).

NAME

getuid, geteuid, getgid, getegid - get real user, effective user, real group, and effective group IDs

SYNOPSIS

```
#include <unistd.h>
uid_t getuid(void);
uid_t geteuid(void);
gid_t getgid(void);
gid_t getegid(void);
```

DESCRIPTION

The following functions return the information indicated:

`getuid()`	Real-user-ID of the calling process.
`geteuid()`	Effective-user-ID of the calling process.
`getgid()`	Real-group-ID of the calling process.
`getegid()`	Effective-group-ID of the calling process.

No means is available for ascertaining the saved-user-ID or saved-group-ID of a process.

SEE ALSO

setuid(2).

STANDARDS CONFORMANCE

`getuid()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

`getegid()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

`geteuid()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

`getgid()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

ioctl - control device

SYNOPSIS

```
#include <sys/ioctl.h>

int ioctl(int fildes, int request, ... /* arg */);
```

DESCRIPTION

`ioctl()` performs a variety of functions on character special files (devices). The write-ups of various devices in Section (7) discuss how `ioctl()` applies to them. The type of *arg* is dependent on the specific `ioctl()` call, as described in Section (7).

request is made up of several fields which encode the size and direction of the argument (referenced by *arg*), as well as the desired command. An enumeration of the request fields are:

`IOC_IN`	Argument is read by the driver (meaning that the argument is copied from the application to the driver).
`IOC_OUT`	Argument is written by the driver (meaning that the argument is copied from the driver to the application). Ignored if an error occurs.
`IOCSIZE_MASK`	Number of bytes in the passed argument. A nonzero size indicates that *arg* is a pointer to the passed argument. A zero size indicates that *arg* is the passed argument (if the driver wants to use it), and is not treated as a pointer.
`IOCCMD_MASK`	The request command itself.

When both `IOC_IN` and `IOC_OUT` are zero, it can be assumed that *request* is not encoded for size and direction, for compatibility purposes. Requests that do not require any data to be passed and requests that use *arg* as a value (as opposed to a pointer), have the `IOC_IN` bit set to one and the `IOCSIZE_MASK` field set to zero.

The following macros are used to create the request argument. *x* and *y* are concatenated (`(x<<8) | y`) to form `IOCCMD` and shifted into the proper location according to `IOCCMD_MASK`. `t` is the type (e.g. `struct hpib_cmd`) of the actual argument that the request references, and its size is taken and shifted into the appropriate place according to `IOCSIZE_MASK.`

`_IOR(x,y,t)`	Sets `IOC_OUT` and initializes the values at `IOCCMD_MASK` and `IOCSIZE_MASK` accordingly.
`_IOW(x,y,t)`	Sets `IOC_IN` and initializes the values at `IOCCMD_MASK` and `IOCSIZE_MASK` accordingly.
`_IOWR(x,y,t)`	Sets both `IOC_IN` and `IOC_OUT` and initializes the values at `IOCCMD_MASK` and `IOCSIZE_MASK.`

Note: any data structure referenced by *arg* must *not* contain any pointers.

RETURN VALUE

If an error has occurred, a value of -1 is returned and `errno` is set to indicate the error.

`ioctl()` fails if one or more of the following are true: IOC_OUT is ignored if an error occurs.

[EBADF]	*fildes* is not a valid open file descriptor.
[ENOTTY]	The request is not appropriate to the selected device.
[EINVAL]	*request* or *arg* is not valid.
[EINTR]	A signal was caught during the `ioctl()` system call.
[EPERM]	Typically this error indicates that an ioctl request was attempted that is forbidden in some way to the calling process.

WARNINGS

Check all references to *signal*(5) for appropriateness on systems that support *sigvector*(2). *sigvector*(2) can affect the behavior described on this page.

AUTHOR

`ioctl()` was developed by AT&T and HP.

SEE ALSO
ioctl(5), termio(7).

STANDARDS CONFORMANCE
`ioctl()`: SVID2, XPG2

NAME

ipcconnect - initiate a connection to another process

SYNOPSIS

```
#include <sys/ns_ipc.h>

void ipcconnect(
    ns_int_t calldesc,
    ns_int_t destdesc,
    ns_int_t *flags,
    short opt[],
    ns_int_t *vcdesc,
    ns_int_t *result);
```

DESCRIPTION

`ipcconnect()` is used to initiate a virtual circuit on which data can be sent and received. When `ipcconnect()` returns, a connection is not yet established; a successful return only indicates that a connection request was sent without error. Actively establishing a virtual circuit with NetIPC calls is a two-step process:

- `ipcconnect()` is called to request a connection, then
- *ipcrecv*(3N) is called to find out if a connection initiated with `ipcconnect()` was successfully established.

The *opt* parameter can be used to specify the number of bytes you expect to send and receive on the connection. The default for both sending and receiving is 100 bytes. This information is passed to the underlying protocol. When TCP is the underlying protocol, it limits the number of bytes that can be queued on a socket to the specified value.

PARAMETERS

calldesc (input parameter)
`NS_NULL_DESC` should be specified. A valid call socket descriptor can be specified to ensure backward compatibility.

destdesc (input parameter)
A destination descriptor obtained by calling `ipclookup()` or `ipcdest()` (see *ipclookup*(3N) and *ipcdest*(3N)).

flags (input parameter)
Either 0 or a pointer to 0. All other values are reserved for future use.

opt (input parameter)
Options for this call. If no options are used, this parameter can be null. Otherwise, see below.

vcdesc (output parameter)
A pointer to a virtual circuit number that can be used in subsequent NetIPC calls to reference the connection.

result (output parameter)
See ERRORS below.

OPTION PARAMETER

`NSO_MAX_SEND_SIZE` (*optioncode* = 3) (*datalength* = 2) A two-byte integer specifying the maximum number of bytes that can be sent with a single `ipcsend()` call on this connection (see *ipcsend*(3N)). **Range:** 1 to 32 000 bytes. **Default:** 100 bytes.

`NSO_MAX_RECV_SIZE` (*optioncode* = 4) (*datalength* = 2) A two-byte integer specifying the maximum number of bytes that can be received with a single `ipcrecv()` call on this connection (see *ipcrecv*(3N)). **Range:** 1 to 32,000 bytes. **Default:** 100 bytes.

RETURN VALUE

None. Errors are returned in the *result* parameter.

ERRORS

[NSR_ADDR_NOT_AVAIL]	The protocol address specified by the destination descriptor is 0, which is illegal for connection establishment, OR there is no available interface to the destination network.
[NSR_BOUNDS_VIO]	A length or offset value in the option parameter is illegal *or* one of the pointer arguments is invalid.
[NSR_DESC]	The *calldesc* argument is not **NSR_NULL_DESC** or a valid socket descriptor, *or* the *destdesc* argument is not a valid destination descriptor.
[NSR_DEST_UNREACHABLE]	The network or host specified by the destination descriptor is unreachable from this host at this time.
[NSR_DUP_OPTION]	A particular option is defined more than once in the *opt* parameter.
[NSR_FLAGS]	An unsupported flag is set in the *flags* parameter.
[NSR_KIND_AND_PROTOCOL]	The requested protocol is not supported in the default domain.
[NSR_MSGSIZE]	The value specified in **NSO_MAX_SEND_SIZE** or **NSO_MAX_RECV_SIZE** is invalid.
[NSR_NO_DESC_AVAILABLE]	The process exceeded the system-defined number of file and socket descriptors that can be open at a time (see *getrlimit*(2)).
[NSR_NO_ERROR]	The call was successful.
[NSR_NO_FILE_AVAILABLE]	The system cannot allocate a file structure at this time.
[NSR_NO_MEMORY]	Sufficient system memory is not available to execute this call at this time.
[NSR_NOT_ALLOWED]	An unsupported flag is set in the *flags* parameter.
[NSR_NOT_CALL_SOCKET]	The *calldesc* argument is not an **NS_CALL** socket.
[NSR_OPT_OPTION]	An option in the *opt* parameter is unknown or unsupported.
[NSR_OPT_SYNTAX]	A length or offset value in the *opt* parameter is invalid.
[NSR_PROTOCOL]	The requested protocol is not supported.
[NSR_SIGNAL_INDICATION]	The call aborted due to a signal.

AUTHOR

`ipcconnect()` was developed by HP.

SEE ALSO

getrlimit(2), ipccontrol(2), ipccreate(2), ipcdest(2), ipcgetnodename(2), ipclookup(2), ipcname(2), ipcnamerase(2), ipcrecv(2), ipcrecvcn(2), ipcselect(2), ipcsend(2), ipcsetnodename(2), ipcshutdown(2), addopt(3N), initopt(3N), ipcerrmsg(3N), optoverhead(3N), readopt(3N).

NAME

ipccontrol - perform special operations on a NetIPC socket

SYNOPSIS

```
#include <sys/ns_ipc.h>

void ipccontrol(
    ns_int_t descriptor,
    ns_int_t request,
    const void *wrtdata,
    ns_int_t wlen,
    void *readdata,
    ns_int_t *rlen,
    ns_int_t *flags,
    ns_int_t *result);
```

DESCRIPTION

`ipccontrol()` is used to manipulate NetIPC sockets. The type of request is specified in the *request* parameter. Some parameters are optional and not used in all requests. If *wrtdata* is not used, *wlen* must be zero. If *readdata* is unused, *rlen* must be zero.

All processes that own descriptors for a particular socket are affected by `ipccontrol()` operations performed on that socket. For example, one process can change a socket's read or write threshold, synchronous timeout interval, or synchronous/asynchronous mode while another process is reading, writing, or selecting on that socket. Exactly when the process that is sharing the socket will be affected by these operations cannot be reliably predicted. Reads, writes, and selects in progress may complete after using either the previous, new, or a combination of the previous and new values.

Parameters

descriptor (input parameter)
The descriptor that refers to the socket to be manipulated.

request (input parameter)
Request code. Defines which operation is to be performed. See below.

wrtdata (input parameter)
A data buffer used to pass timeout and threshold information.

wlen (input parameter)
Length in bytes of the *wrtdata* data buffer.

readdata (output parameter)
A data buffer used to contain any data returned by the call.

rlen (input/output parameter)
The length in bytes of the *readdata* data buffer. On output, this parameter contains the total number of bytes returned to the process.

flags (input parameter)
Reserved for future use. This parameter should be 0 or a pointer to 0.

result (output parameter)
The error code returned. See ERRORS below for more information.

Request Parameter

`NSC_NBIO_ENABLE` (request code 1)
Place socket referenced by *descriptor* in asynchronous mode.

`NSC_NBIO_DISABLE` (request code 2)
Place socket referenced by *descriptor* in synchronous mode.

`NSC_TIMEOUT_RESET` (request code 3)
Change the referenced socket's synchronous timeout. The default timeout value is 60 seconds. The timeout value is specified in tenths of seconds (for example, a value of 1200 indicates 120 seconds). The new timeout value is treated as a 16-bit signed integer, and must be placed in the first two bytes of the *wrtdata* parameter. The timeout value must be in the range of zero

to 32 767. Negative values have no meaning and will result in an error. A value of zero sets the timeout to infinity. The timeout is not reset if the referenced socket is switched to asynchronous mode then back to synchronous mode.

NSC_TIMEOUT_GET (request code 4)
Return the synchronous timeout value for the socket referenced in the *descriptor* parameter. The timeout value is treated as a 16-bit signed integer, and is returned in the *readdata* parameter.

NSC_RECV_THRESH_RESET (request code 1000)
Change the read threshold of the VC socket referenced in *descriptor* parameter. Read thresholds are one byte by default. The *descriptor* parameter must reference a VC socket descriptor. The new read threshold value must be placed in the first two bytes of the *wrtdata* parameter.

NSC_SEND_THRESH_RESET (request code 1001)
Change the write threshold of the VC socket referenced in the *descriptor* parameter. Write thresholds are one byte by default. The *descriptor* parameter must reference a VC socket descriptor. The new write threshold value must be placed in the first two bytes of the *wrtdata* parameter.

NSC_RECV_THRESH_GET (request code 1002)
Return the current write threshold for the VC socket referenced in the *descriptor* parameter. The *descriptor* parameter must reference a VC socket descriptor. The write threshold is treated as a 16-bit signed integer, and is returned in the *readdata* parameter.

NSC_SEND_THRESH_GET (request code 1003)
Return the current read threshold for the VC socket referenced in the *descriptor* parameter. The *descriptor* parameter must reference a VC socket descriptor. The read threshold is treated as a 16-bit signed integer, and is returned in the *readdata* parameter.

NSC_GET_NODE_NAME (request code 9008)
Obsolescent. Use *getnodename*(2) instead.

RETURN VALUE

None. Errors are returned in the *result* parameter.

ERRORS

[NSR_BOUNDS_VIO]	One of the pointer arguments is invalid.
[NSR_DESC]	The argument *descriptor* is not a valid NetIPC socket descriptor.
[NSR_DLEN]	The specified *wlen* or *rlen* parameter is invalid.
[NSR_NO_ERROR]	The call was successful.
[NSR_REQUEST]	The request was unknown.
[NSR_TIMEOUT_VALUE]	An illegal timeout value was specified.
[NSR_THRESH_VALUE]	An illegal threshold value was specified.

AUTHOR

`ipccontrol()` was developed by HP.

SEE ALSO

ipcconnect(2), ipccreate(2), ipcdest(2), ipcgetnodename(2), ipclookup(2), ipcname(2), ipcnamerase(2), ipcrecv(2), ipcrecvcn(2), ipcselect(2), ipcsend(2), ipcsetnodename(2), ipcshutdown(2), addopt(3N), initopt(3N), ipcerrmsg(3N), optoverhead(3N), readopt(3N).

NAME

ipccreate - create a call socket

SYNOPSIS

```
#include<sys/ns_ipc.h>

void ipccreate(
    ns_int_t socketkind,
    ns_int_t protocol,
    ns_int_t *flags,
    short opt[],
    ns_int_t *calldesc,
    ns_int_t *result);
```

DESCRIPTION

`ipccreate` is used to create a call socket for use with subsequent NetIPC calls to establish a virtual circuit connection between two processes.

A process can have a system-defined maximum number of descriptors open at a time (see *getrlimit*(2)). `ipccreate()` returns an error if a process attempts to exceed this limit. This limit includes file descriptors, as well as socket descriptors and destination descriptors. These descriptors may reference sockets and/or files inherited by or otherwise opened by the process.

The `NSO_PROTOCOL_ADDRESS` option (code 128) can be used to create a call socket with a specific protocol address. The peer process, which must have *a priori* knowledge of this protocol address, can call `ipcdest()` with this address to obtain a destination descriptor that will enable `ipcconnect()` to connect to this call socket.

PARAMETERS

socketkind	(input parameter) Must be `NS_CALL`. Other values are reserved for future use.
protocol	(input parameter) Indicates the protocol module that the calling process wants to access. Must be `NSP_TCP` or zero. Other values are reserved for future use.
flags	(input parameter) Must be 0 or a pointer to 0. Other values are reserved for future use.
opt	(input parameter) See below.
calldesc	(output parameter) Call socket descriptor. Refers to the newly-created call socket.
result	(output parameter) See diagnostics section below for more information.

Opt Parameter

See `initopt` and `addopt` for more information on NetIPC option buffers.

`NSO_MAX_CONN_REQ_BACK`
(*optioncode* = 6) (*datalength* = 2)
A two-byte integer specifying the maximum number of unreceived connection requests that can be queued to a call socket. If this value is not specified, the default maximum is used. **Default:** One request. **Range:** 1 to 20. (Note that a queue limit of one may be too few if many processes attempt to initiate connections to the call socket simultaneously. If this occurs, some connection requests will be automatically rejected.)

`NSO_PROTOCOL_ADDRESS`
(*optioncode* = 128) (*datalength* = 2)
A two-byte integer specifying a protocol-specific address to be used by the newly-created call socket. If this option is not specified, or if zero is specified, Net.SM IPC dynamically allocates an address. You must have super-user capability to request protocol addresses less than 1024. **Recommended Range:** 30 767 through 32 767. If the protocol is TCP then this option specifies the TCP port.

RETURN VALUE

None. Errors are returned in the *result* parameter.

ERRORS

[NSR_BOUNDS_VIO]	One of the pointer arguments is invalid.

[NSR_DUP_ADDRESS] The protocol address specified in the **NSO_PROTOCOL_ADDRESS** option is in use.

[NSR_DUP_OPTION] A particular option is defined more than once in the *opt* parameter.

[NSR_FLAGS] The *flags* parameter was not 0 or a pointer to 0.

[NSR_KIND_AND_PROTOCOL] The requested protocol is not supported in the default domain.

[NSR_MAX_CONNECTQ] The **NSO_MAX_CONN_REQ_BACK** option must be greater than 0 and less than 20.

[NSR_NO_DESC_AVAILABLE] The process exceeded the system-defined number of file and socket descriptors that can be open at a time (see *getrlimit*(2)).

[NSR_NO_ERROR] The call was successful.

[NSR_NO_FILE_AVAILABLE] The system cannot allocate a file structure at this time.

[NSR_NO_MEMORY] Sufficient system memory is not available to execute this call at this time.

[NSR_NOT_ALLOWED] The protocol address specified via the **NSO_PROTOCOL_ADDRESS** option was less than 1024 and the program did not have super-user capability.

[NSR_OPT_OPTION] An option specified in the *opt* parameter is unknown or unsupported.

[NSR_OPT_SYNTAX] A length or offset value in the *opt* parameter is invalid.

[NSR_PROTOCOL] The combination of the *protocol* parameter and *socketkind* parameter could not be satisfied. At least one is incorrect.

AUTHOR

ipccreate() was developed by HP.

SEE ALSO

getrlimit(2), ipcconnect(2), ipccontrol(2), ipcdest(2), ipcgetnodename(2), ipclookup(2), ipcname(2), ipcnamerase(2), ipcrecv(2), ipcrecvcn(2), ipcselect(2), ipcsend(2), ipcsetnodename(2), ipcshutdown(2), addopt(3N), initopt(3N), ipcerrmsg(3N), optoverhead(3N), readopt(3N).

NAME

ipcdest - create a NetIPC destination descriptor

SYNOPSIS

```
#include <sys/ns_ipc.h>

void ipcdest(
    ns_int_t socketkind,
    const char *nodename,
    ns_int_t nodelen,
    ns_int_t protocol,
    short *protoaddr,
    ns_int_t protolen,
    ns_int_t *flags,
    short opt[],
    ns_int_t *destdesc,
    ns_int_t *result);
```

DESCRIPTION

`ipcdest()` creates a destination descriptor which the calling process can use to establish a connection to another process.

`ipcdest()` can be used to obtain a destination descriptor for a call socket with a particular protocol address. To create a call socket with a particular address, use `ipccreate()` with the NSO_PROTOCOL_ADDRESS option (see *ipccreate*(3N)).

`ipcdest()` does not verify that the remote endpoint described by the input parameters exists. This evaluation is delayed until the destination descriptor is used in a subsequent `ipcconnect()` call.

Parameters

socketkind (input parameter) Defines the type of socket. Must be **NS_CALL** or 3 to specify a call socket. Other values are reserved for future use.

nodename (input parameter) The ASCII-coded name that identifies the node where the call socket with *protoaddr* resides. **Default:** The organization, organization and domain, or all parts of the node name can be omitted. When organization or organization and domain are omitted, they default to the local organization and/or domain. If the *nodelen* parameter is set to zero, this parameter is ignored and the node name defaults to the local node.

nodelen (input parameter) The length in bytes of the *nodename* parameter. Zero indicates that the *nodename* parameter is ignored, and the node name defaults to the local node. A fully-qualified node can be up to 50 bytes long.

protocol (input parameter) Defines the Transport Layer protocol to be used. Must be **NSP_TCP** or 4 to indicate the Transmission Control Protocol (TCP). Other values are reserved for future use.

protoaddr (input parameter) A data buffer that contains a TCP protocol address.

protolen (input parameter) The length in bytes of the protocol address. TCP protocol addresses are two bytes long.

flags (input parameter) This parameter is reserved for future use. All bits must be clear (not set).

opt (input parameter) No options are defined for this call. You must set this parameter to zero (0) or pass the constant (C programs only) **NSO_NULL**.

destdesc (output parameter) Destination descriptor. Can be used in a subsequent `ipcconnect` call to establish a connection to the call socket with *protoaddr*.

result (output parameter) See ERRORS below.

RETURN VALUE

None. Errors are returned in the *result* parameter.

ERRORS

[NSR_NO_ERROR]	The call was successful.
[NSR_BOUNDS_VIO]	A parameter address is invalid.
[NSR_NOT_CALL_SOCKET]	The *socketkind* parameter is not NS_CALL.
[NSR_FLAGS]	The value in the *flags* parameter is invalid.
[NSR_OPT_OPTION]	An option specified in the *opt* parameter is unknown or unsupported.
[NSR_PROTOCOL]	The protocol of the specified socket is not supported by the local system.
[NSR_KIND_AND_PROTOCOL]	The socketkind and protocol parameters are not compatible.
[NSR_ADDR_OPT]	The value in the *protolen* parameter is invalid.
[NSR_NLEN]	The value in the *nodelen* parameter is invalid.
[NSR_NODE_NAME_SYNTAX]	The node specified in the *nodename* parameter is invalid.
[NSR_NO_NODE]	The specified node is unknown to the local host.
[NSR_NO_MEMORY]	Sufficient system memory is not available to execute this call at this time.
[NSR_PATH_REPORT]	The path report could not be interpreted.
[NSR_DEST_UNREACHABLE]	The path report contained no usable paths.
[NSR_NO_FILE_AVAIL]	No file table entries are available at this time.
[NSR_NO_DESC_AVAIL]	The process exceeded the system-defined number of file and socket descriptors that can be open at a time (see *getrlimit*(2)).

AUTHOR

`ipcdest()` was developed by HP.

SEE ALSO

getrlimit(2), ipcconnect(2), ipccontrol(2), ipccreate(2), ipcgetnodename(2), ipclookup(2), ipcname(2), ipcnamerase(2), ipcrecv(2), ipcrecvcn(2), ipcselect(2), ipcsend(2), ipcsetnodename(2), ipcshutdown(2), addopt(3N), initopt(3N), ipcerrmsg(3N), optoverhead(3N), readopt(3N).

NAME

ipcgetnodename - obtain NetIPC node name of current host

SYNOPSIS

```
#include <sys/ns_ipc.h>

void ipcgetnodename(
    char *nodename,
    ns_int_t *size,
    ns_int_t *result);
```

DESCRIPTION

`ipcgetnodename()` returns the NetIPC node name for the current processor as set by `setnodename()` in the array to which *nodename* points (see *setnodename*(2)).

Parameters

nodename (input parameter) A pointer to a character array in which the ASCII-coded NetIPC node name is to be returned.

size (input/output parameter) The length in bytes of the *nodename* array on input and the length of the returned NetIPC node name on output.

result (output parameter) See ERRORS below.

RETURN VALUE

None. Errors are returned in the *result* parameter.

ERRORS

[NSR_NO_ERROR]	The call was successful.
[NSR_NLEN]	The value of the *size* parameter is not large enough for the NetIPC node name.
[NSR_BOUNDS_VIO]	Output parameter address is invalid.

AUTHOR

ipcgetnodename was developed by HP.

SEE ALSO

ipcconnect(2), ipccontrol(2), ipccreate(2), ipcdest(2), ipclookup(2), ipcname(2), ipcnamerase(2), ipcrecv(2), ipcrecvcn(2), ipcselect(2), ipcsend(2), ipcsetnodename(2), ipcshutdown(2), addopt(3N), initopt(3N), ipcerrmsg(3N), optoverhead(3N), readopt(3N).

NAME

ipclookup - obtain a NetIPC destination descriptor

SYNOPSIS

```
#include <sys/ns_ipc.h>

void ipclookup(
    const char *socketname,
    ns_int_t nlen,
    const char *nodename,
    ns_int_t nodelen,
    ns_int_t *flags,
    ns_int_t *destdesc,
    ns_int_t *protocol,
    ns_int_t *socketkind,
    ns_int_t *result);
```

DESCRIPTION

ipclookup() is used to obtain a destination descriptor for a named call socket. When supplied with valid socket and node names, **ipclookup()** looks up the call socket in the socket registry at the node specified in the *nodename* parameter and returns a destination descriptor that can be used by subsequent NetIPC calls to locate the call socket. A destination descriptor is required by the **ipcconnect()** call to provide the information necessary to direct a connection request to the proper node and call socket and thus initiate a connection.

When a process attempts to look up a socket name in the appropriate socket registry, the name must be there or an NSR_NAME_NOT_FOUND error is returned to the calling process. When two processes are running concurrently, it may be difficult to ensure that a socket name is placed in the socket registry prior to being "looked up" by another process. This problem is referred to as a race condition because the two processes are "racing" to see which one accesses the socket registry first.

In order to avoid a race situation, the process that calls **ipclookup()** can test for a NSR_NAME_NOT_FOUND error in the call's *result* parameter. If this error is returned, the process can try again by entering a loop and repeating the **ipclookup()** call for a specified number of times. The process should also call **sleep()** to suspend execution for an interval (see *sleep*(3C), then repeat the **ipclookup()** call.

Parameters

socketname (input parameter) The name of the call socket to be "looked up". Uppercase and lowercase characters are treated as equivalent.

nlen (input parameter) The length of the *socketname* parameter in characters. Maximum length is 16 characters.

nodename (input parameter) The ASCII-coded name that that identifies the node where the socket specified in the *socketname* parameter resides. Default: organization, organization and domain, or all parts of the node name can be omitted. When organization or organization and domain are omitted, they default to the local organization and/or domain. If the entire parameter is omitted, the node name defaults to the local node.

nodelen (input parameter) The length in bytes of the *nodename* parameter. If zero is specified, NetIPC searches the local node's socket registry (see *nodename* parameter above for more information).

flags (input parameter) This parameter is reserved for future use. All bits must be clear (not set).

destdesc (output parameter) Destination descriptor. Refers to the descriptor that indicates the location of the named call socket. Can be used in subsequent NetIPC calls.

protocol (output parameter) This parameter is reserved for future use. Zero (0) is always returned in this parameter.

socketkind (output parameter) Identifies the socket's type. Can be used in an **ipccreate()** call to create a socket of the appropriate type.

result (output parameter) See ERRORS below.

RETURN VALUE

None. Errors are returned in the *result* parameter.

ERRORS

[NSR_NO_ERROR]	The call was successful.
[NSR_BOUNDS_VIO]	A parameter address is invalid.
[NSR_FLAGS]	The value in the *flags* parameter in invalid.
[NSR_PROTOCOL]	The protocol of the socket specified by *socketname* is not supported by the local system.
[NSR_NLEN]	The value in the *nodelen* parameter is not valid.
[NSR_NODE_NAME_SYNTAX]	The string pointed to by *nodename* is invalid.
[NSR_NO_NODE]	*nodename* is unknown to the local host.
[NSR_NO_MEMORY]	Sufficient system memory is not available to execute this call at this time.
[NSR_PATH_REPORT]	The path report could not be interpreted.
[NSR_NAME_NOT_FOUND]	The specified *socketname* was not found in the socket registry.
[NSR_CANT_CONTACT_SERVER]	The `ipclookup()` request could not be sent to the remote socket registry server.
[NSR_NO_REG_RESPONSE]	No response was received from the remote socket registry server.
[NSR_VERSION]	The reply from the remote socket registry indicates a version error occurred.
[NSR_BAD_REG_MSG]	A corrupt reply message was received from the remote socket registry server.
[NSR_NO_FILE_AVAIL]	No file table entries are available.
[NSR_NO_DESC_AVAIL]	The process exceeded the system-defined number of file and socket descriptors that can be open at a time (see *getrlimit*(2)).

AUTHOR

`ipclookup()` was developed by HP.

SEE ALSO

getrlimit(2), ipcconnect(2), ipccontrol(2), ipccreate(2), ipcdest(2), ipcgetnodename(2), ipcname(2), ipcnamerase(2), ipcrecv(2), ipcrecvcn(2), ipcselect(2), ipcsend(2), ipcsetnodename(2), ipcshutdown(2), addopt(3N), initopt(3N), ipcerrmsg(3N), optoverhead(3N), readopt(3N), sleep(3C).

NAME

ipcname - associate a name with a call socket or destination descriptor

SYNOPSIS

```
#include <sys/ns_ipc.h>

void ipcname(
    ns_int_t descriptor,
    const char *socketname,
    ns_int_t nlen,
    ns_int_t *result);
```

DESCRIPTION

`ipcname()` associates a name with a call socket and adds this information to the local node's socket registry. The name a process associates with a call socket must be known to its peer process so that the peer process can look up the name with an `ipclookup()` call. This can be accomplished by hard-coding the name into both processes or by passing the name from one process to another.

The name associated with a call socket can be user-defined or randomly generated by NetIPC, and must be unique to your node (i.e., it cannot be simultaneously associated with two descriptors). For example, if a call to `ipcname()` assigns the name `Liz` to a call socket, a subsequent call with `Liz` results in an error. To ensure that the name being assigned to a call socket is unique, use the random name generating feature of `ipcname()` (see the *nlen* parameter below for more information). A call socket can be listed under multiple names.

`ipcname()` always enters its listings into the local node's socket registry. `ipclookup()`, by contrast, can look up socket names at both the local node and at a remote node. Since "long distance" look-ups take longer than local look-ups, it may be helpful to use `ipcname()` to name a destination descriptor associated with a remotely named call socket. When a process names a destination descriptor, the name of the destination descriptor is placed in the local socket registry (the socket registry at the node where the calling process resides). This allows other processes to look up the name in the local socket registry rather than calling `ipclookup()` to look up the name in a socket registry at a remote node where the call socket resides.

Using `ipcname()` to name a destination descriptor is less reliable than looking up the socket name at the remote node because destination descriptors can become outdated. As a precaution, locally stored destination descriptors should be refreshed periodically.

`ipcname()` cannot be used to name VC sockets.

PARAMETERS

descriptor	(input parameter) The descriptor that references the call socket to be named. Can be a call socket descriptor or a destination descriptor.
socketname	(input/output parameter) The ASCII-coded name to be associated with the descriptor. Uppercase and lowercase characters are treated as equivalent. NetIPC can also return a randomly-generated name in this parameter (see the *nlen* parameter).
nlen	(input parameter) The length in characters of the *socketname* parameter. Maximum length is 16 characters. If zero is specified, NetIPC returns a random, eight-byte name in the *socketname* parameter. The eight-byte length is not returned in the *nlen* parameter.
result	(output parameter) See ERRORS below.

RETURN VALUE

None. Errors are returned in the *result* parameter.

ERRORS

[NSR_NO_ERROR]	The call was successful.
[NSR_CANT_NAME_VC]	The *descriptor* parameter corresponds to a VC socket and naming of VC sockets is not allowed.
[NSR_DESC]	The *descriptor* parameter does not correspond to a NetIPC socket.

[NSR_NLEN]	The value specified in the *nlen* parameter is invalid.
[NSR_DUP_NAME]	The specified *socketname* already exists in the local socket registry.
[NSR_NO_MEMORY]	Sufficient system memory is not available to execute this call at this time.
[NSR_BOUNDS_VIO]	The output parameter address is invalid.

AUTHOR

`ipcname()` was developed by HP.

SEE ALSO

ipcconnect(2), ipccontrol(2), ipccreate(2), ipcdest(2), ipcgetnodename(2), ipclookup(2), ipcnamerase(2), ipcrecv(2), ipcrecvcn(2), ipcselect(2), ipcsend(2), ipcsetnodename(2), ipcshutdown(2), addopt(3N), initopt(3N), ipcerrmsg(3N), optoverhead(3N), readopt(3N).

NAME

ipcnamerase - delete a name associated with a NetIPC call socket or destination descriptor

SYNOPSIS

```
#include <sys/ns_ipc.h>

void ipcnamerase(
    const char *socketname,
    ns_int_t nlen,
    ns_int_t *result);
```

DESCRIPTION

`ipcnamerase()` can be called to remove listings from the local node's socket registry. Only the owner of a call socket or destination descriptor can remove the socket's name from the local socket registry.

If a call socket descriptor or destination descriptor is destroyed by `ipcshutdown()` or if its last owner terminates, any listings for it that exist at the local socket registry are automatically purged.

If multiple processes have descriptors for the same socket, the first `ipcnamerase()` call succeeds; subsequent calls fail.

Parameters

socketname	(input parameter) The ASCII-coded name that was previously associated with a call socket descriptor or destination descriptor via `ipcname()`. Uppercase and lowercase characters are treated as equivalent.
nlen	(input parameter) The length in bytes of the specified name. Maximum length is 16 bytes.
result	(output parameter) See ERRORS below.

RETURN VALUE

None. Errors are returned in the *result* parameter.

ERRORS

[NSR_NO_ERROR]	The call was successful.
[NSR_NLEN]	The value specified in the nlen parameter is invalid.
[NSR_NAME_NOT_FOUND]	The name specified by *socketname* does not exist in local socket registry.
[NSR_NO_OWNERSHIP]	The caller is not the owner of the named socket.

AUTHOR

`ipcnamerase()` was developed by HP.

SEE ALSO

ipcconnect(2), ipccontrol(2), ipccreate(2), ipcdest(2), ipcgetnodename(2), ipclookup(2), ipcname(2), ipcrecv(2), ipcrecvcn(2), ipcselect(2), ipcsend(2), ipcsetnodename(2), ipcshutdown(2), addopt(3N), initopt(3N), ipcerrmsg(3N), optoverhead(3N), readopt(3N).

NAME

ipcrecv - establish an NetIPC virtual circuit connection *or* receive data on an established connection

SYNOPSIS

```
#include <sys/ns_ipc.h>

void ipcrecv(
    ns_int_t vcdesc,
    void *data,
    ns_int_t *dlen,
    ns_int_t *flags,
    short opt[],
    ns_int_t *result);
```

DESCRIPTION

`ipcrecv()` serves two purposes:

- Establish a virtual circuit connection that was initiated with `ipcconnect()` (see *ipcconnect*(2)),
- Receive data on a previously established virtual circuit connection.

After a program calls `ipcconnect()`, it must call `ipcrecv()` to complete the connection. When `ipcrecv()` is called to finish establishing a connection, no data is returned in the *data* parameter and the *dlen* parameter is ignored. An exception `ipcselect()` (see *ipcselect*(2)) can be performed to determine whether connections are pending on a call socket.

When `ipcrecv()` is called to receive data queued on a established connection, several different alternatives are available:

- Normal reading: Data is moved from the connection queue into the user's buffer.
- Preview reading: This alternative is specified by setting the NSF_PREVIEW bit (bit 30) of the *flags* parameter. When this bit is set, data is copied into the process's buffer, but still remains in the connection queue. Consequently, the next `ipcrecv()` call reads the same data.
- Vectored or "scattered" reading: The calling process can pass a data vector argument that describes one or more locations. Received data is then placed into these locations. This alternative can be used with both the normal and the preview read described above, and is specified by setting the `NSF_VECTORED` bit (bit 31) of the *flags* parameter.

For vectored reads an iovec structure contains the data vector. An iovec structure can be defined as:

```
struct iovec {
     char        *iov_base;
     unsigned    iov_len;
};
```

and the normal type for the data argument can be replaced by:

```
struct iovec *data;
```

Each iovec entry specifies the base address and length of an area in memory where data should be placed. `ipcrecv()` always fills one area completely before proceeding to the next area.

`ipcrecv()` behavior varies, depending on whether the socket referenced is in synchronous or asynchronous mode. A socket is in synchronous mode by default. It can be placed in asynchronous mode by calling `ipccontrol()` (see *ipccontrol*(2)). By default, calls that block reach their timeout limit in 60 seconds. The length of the timeout period can be changed by calling `ipccontrol()`. Refer to *ipccontrol*(2) for more information.

If the socket referenced by `ipcrecv()` is in synchronous mode and no data is queued on the connection, the call blocks until data arrives or the socket timer expires.

If the socket referenced by `ipcrecv()` is in asynchronous mode and no data is queued on the connection, NSR_WOULD_BLOCK is returned in the *result* parameter.

Parameters

vcdesc (input parameter) "virtual circuit" socket descriptor. Refers to a socket that:

- Is the endpoint of a virtual circuit connection that has not yet been established, or
- Is the endpoint of an established virtual circuit on which data will be received.

data (output parameter) A pointer to a data buffer for holding the received data, or a pointer to an array of data vectors describing the locations where the data is to be placed.

dlen (input/output parameter) If *data* is a data buffer, *dlen* is the maximum number of bytes that can be received. If *data* is a data vector, *dlen* refers to the length of the data vector in bytes. As a return parameter, *dlen* indicates how many bytes were actually received. If **ipcrecv()** is used to establish a connection (not to receive data), *dlen* is meaningless on input and a value of zero (0) is returned on output.

flags (input/output parameter) See below.

opt (input parameter) A pointer to a NetIPC options buffer. See below.

result (output parameter) The error code returned. Refer to ERRORS below for more information.

Flags Parameter

Flags are only valid on an established connection.

NSF_DATA_WAIT (bit 20) (input parameter) This flag exists for backward compatibility. Existing programs that use this flag may suffer performance degradation due to network congestion avoidance algorithms in the networking protocol code. This flag should be removed from those programs.

NSF_MORE_DATA (bit 26) (output parameter) This bit is always set for backwards compatibility.

NSF_PREVIEW (bit 30) (input parameter) When this bit is set, data queued on the connection may be previewed. Data is placed in the *data* parameter but not removed from the connection queue. Since the data remains in the queue, the next **ipcrecv()** call reads the same data.

NSF_VECTORED (bit 31) (input parameter) When set, this bit indicates that *data* is a data vector and not a data buffer.

Opt Parameter

Options are only valid when **ipcrecv()** is issued against an established connection.

NSO_DATA_OFFSET (*optioncode* = 8) (*datalength* = 2) A two-byte integer that defines a byte offset from the beginning of a data buffer where NetIPC is to begin placing data. This option is valid only if *data* is a data buffer and not a data vector.

RETURN VALUE

None. Errors are returned in the *result* parameter.

ERRORS

[NSR_BOUNDS_VIO] A length or offset value in the *opt* parameter is illegal, *or* one of the pointer arguments is invalid.

[NSR_DESC] The *vcdesc* argument is not a valid socket descriptor

[NSR_DLEN] The specified *dlen* parameter is invalid.

[NSR_DUP_OPTION] A particular option is defined more than once in the *opt* parameter.

[NSR_MESSAGE_SIZE] The value in the *dlen* exceeds the maximum limit for this socket. The default maximum is 100 bytes. You can use **ipccontrol()** to alter this value.

[NSR_NO_ERROR]	The call was successful.
[NSR_NOT_CONNECTION]	The *vcdesc* parameter did not reference a VC socket.
[NSR_OPT_OPTION]	An option specified in the *opt* parameter is unknown or unsupported.
[NSR_OPT_SYNTAX]	A length or offset value in the *opt* parameter is invalid.
[NSR_REMOTE_ABORT]	The connection was aborted due to an action by the peer.
[NSR_REMOTE_RELEASED]	The connection was released due to action by the peer.
[NSR_SIGNAL_INDICATION]	The call aborted due to a signal received.
[NSR_SOCKET_TIMEOUT]	The socket timer expired: • Before the connection completed (first call to `ipcrecv()` and the socket is in synchronous mode), • Before any data arrived (connection established, socket in synchronous mode, **NSF_DATA_WAIT** flag not set), *or* • Before the requested amount of data arrived (connection established, socket in synchronous mode, **NSR_DATA_WAIT** flag set).
[NSR_TOO_MANY_VECTS]	The number of data vectors exceeds the maximum limit of 16.
[NSR_VECT_COUNT]	A negative data length was specified in the iovec.
[NSR_WOULD_BLOCK]	The connection is still pending; the data present is less than requested, the socket in asynchronous mode, and the **NSF_DATA_WAIT** flag is set; or no data is present, and the socket is in asynchronous mode with the **NSF_DATA_WAIT** flag *not* set.

AUTHOR

`ipcrecv()` was developed by HP.

SEE ALSO

ipcconnect(2), ipccontrol(2), ipccreate(2), ipcdest(2), ipcgetnodename(2), ipclookup(2), ipcname(2), ipcnamerase(2), ipcrecvcn(2), ipcselect(2), ipcsend(2), ipcsetnodename(2), ipcshutdown(2), addopt(3N), initopt(3N), ipcerrmsg(3N), optoverhead(3N), readopt(3N).

NAME

ipcrecvcn - receive a connection on a call socket

SYNOPSIS

```
#include <sys/ns_ipc.h>

void ipcrecvcn(
    ns_int_t calldesc,
    ns_int_t *vcdesc,
    ns_int_t *flags,
    short opt[],
    ns_int_t *result);
```

DESCRIPTION

Before calling `ipcrecvcn()`, `ipccreate()` must be called to create a new call socket. When `ipcrecvcn()` is invoked against a call socket that has queued connection requests, it returns a virtual circuit (VC) socket descriptor to the calling process. The VC socket descriptor can be used with subsequent NetIPC calls to send and receive data.

When a socket is created, it is placed in synchronous mode by default. A socket can be placed in asynchronous mode by calling `ipccontrol()`. When the call socket is in synchronous mode, `ipcrecvcn()` blocks until a connection request arrives or the synchronous socket timer expires. The timeout value can be altered by calling `ipccontrol()`. When the call socket is in asynchronous mode, `ipcrecvcn()` returns **NSR_WOULD_BLOCK** if no connection requests are queued for the call socket.

An exception `ipcselect()` can be performed on the referenced call socket to determine if connections are pending on a call socket.

Parameters

calldesc	(input parameter) Socket descriptor. Refers to a call socket owned by the calling process.
vcdesc	(output parameter) VC socket descriptor. Refers to a VC socket that is the end-point of an established virtual circuit connection.
flags	(input parameter) Must be 0. Other values are reserved for future use.
opt	(input parameter) See below.
result	(output parameter) The error code returned. Refer to ERRORS below for more information.

Opt Parameter

NSO_MAX_SEND_SIZE	(*optioncode* = 3) (*datalength* = 2) A signed two-byte integer that specifies the maximum number of bytes you expect to send with a single `ipcsend()` (see *ipcsend*(2)) call on the VC socket. **Range:** 1 to 32 000 bytes. **Default:** 100 bytes.
NSO_MAX_RECV_SIZE	(*optioncode* = 4) (*datalength* = 2) A signed two-byte integer that specifies the maximum number of bytes you expect to receive with a single `ipcrecv()` (see *ipcrecv*(2)) call on this connection. **Range:** 1 to 32 000 bytes. **Default:** 100 bytes.

RETURN VALUE

None. Errors are returned in the *result* parameter.

ERRORS

[NSR_DESC]	*calldesc* is not a valid socket descriptor.
[NSR_BOUNDS_VIO]	A length or offset value in the *opt* parameter is invalid.
[NSR_DUP_OPTION]	A particular option is defined more than once in the *opt* parameter.
[NSR_MSGSIZE]	The value specified in **NSO_MAX_SEND_SIZE** or **NSO_MAX_RECV_SIZE** is invalid.
[NSR_NO_ERROR]	The call was successful.
[NSR_NOT_CALL_SOCKET]	*calldesc* is not a call socket.

[NSR_OPT_OPTION]	The option in *opt* parameter is unknown or unsupported.
[NSR_OPT_SYNTAX]	A length or offset value in the *opt* parameter is invalid.
[NSR_SIGNAL_INDICATION]	A signal was received before a connection request arrived.
[NSR_SOCKET_TIMEOUT]	The socket timer expired before a connection request arrived.
[NSR_WOULD_BLOCK]	The socket is in asynchronous mode and no connection requests are pending.

AUTHOR

`ipcrecvcn()` was developed by HP.

SEE ALSO

ipcconnect(2), ipccontrol(2), ipccreate(2), ipcdest(2), ipcgetnodename(2), ipclookup(2), ipcname(2), ipcnamerase(2), ipcrecv(2), ipcselect(2), ipcsend(2), ipcsetnodename(2), ipcshutdown(2), addopt(3N), initopt(3N), ipcerrmsg(3N), optoverhead(3N), readopt(3N).

NAME

ipcselect - determine status of NetIPC socket

SYNOPSIS

```
#include <sys/ns_ipc.h>

void ipcselect(
    ns_int_t *sdbound,
    int readmap[],
    int writemap[],
    int exceptionmap[],
    ns_int_t timeout,
    ns_int_t *result);
```

DESCRIPTION

`ipcselect()` enables a process to detect and/or wait for the occurrence of any of several events across any of several sockets. A process should call `ipcselect()` with map elements set for descriptors that it owns. If a process attempts to perform a select on a closed or invalid descriptor, an error is returned. Performing a select on a destination descriptor has no meaning and should be avoided.

`ipcselect()` reports three types of information:

- Whether any of the referenced sockets are readable. A VC socket is considered readable if it can immediately satisfy an `ipcrecv()` (see *ipcrecv*(2)) request for a number of bytes greater than or equal to its read threshold. The read threshold is one byte by default and can be modified by calling `ipccontrol()` (see *ipccontrol*(2)). Read selecting on a call socket has no meaning and should be avoided.
- Whether any of the referenced sockets are writeable. A VC socket is considered writeable if it can immediately accommodate an `ipcsend()` (see *ipcsend*(2)) request that involves a number of bytes greater than or equal to the socket's write threshold. The write threshold is one byte by default and can be modified by calling `ipccontrol()`. Write selecting on a call socket has no meaning and should be avoided.
- Whether any of the referenced sockets are exceptional. A VC socket is exceptional if it is not connected. A call socket is exceptional if it has a connection queued on it (i.e., if a subsequent call to `ipcrecvcn()` can succeed).

When a socket is shared (i.e., more than one process has a descriptor for the same socket), an `ipcsend()` call may return an NSR_WOULD_BLOCK error even if a previous `ipcselect()` call indicated that the socket was writeable. For example, this would occur if another process (with a descriptor for the same socket) called `ipcsend()` after the original process called `ipcselect()` and before it called `ipcsend()`.

The following are examples of read selecting, write selecting, and exception selecting using `ipcselect()`.

Detecting Connection Requests

By setting bits in the *exceptionmap* parameter, a process can determine whether incoming connection requests are queued to certain call sockets. For example: Process *A* must determine whether certain call sockets have received connection requests. To do this, Process *A* calls `ipcselect()` with the *exceptionmap* map elements set to correspond to these sockets. Assuming that the timeout interval is long enough (set by *timeout* parameter), `ipcselect()` completes after at least one connection has been established and has been queued on one of the sockets specified in *exceptionmap*. When the call completes, only those elements remain set that correspond to sockets which have queued connections; the other elements will have been cleared.

Performing a Read Select

By setting elements in the *readmap* parameter, a process can determine whether certain VC sockets are readable. For example: Process *A* must determine which of its VC sockets have data queued to them. To do this, Process *A* performs a read select on those sockets by setting elements in the *readmap* parameter to correspond with the desired VC sockets. Upon completion of the call, only the elements that represent readable sockets remain set; the other elements will have been cleared. Process *A* can call `ipcselect()` with a zero-length timeout to determine the status of a socket immediately, or with a non-zero timeout if it is willing to wait for data to arrive.

Performing a Write Select

By setting bits in the *writemap* parameter, a process can determine whether certain VC sockets are writeable. For example: Process *A* must determine which of its VC sockets can accommodate a new `ipcsend()` request, and which of its call sockets can accommodate a new `ipcconnect()` request (see *ipcsend*(2) and *ipcconnect*(2)). To do this, it can perform a write select on these sockets by setting elements in the *writemap* parameter to correspond with the desired VC and call sockets. Upon completion of the call, only the elements that represent writeable sockets will remain set; the other elements will have been cleared. Process *A* can call `ipcselect()` with a zero-length timeout to determine the status of a socket immediately, or with a non-zero timeout if it is willing to wait before sending data on the connection.

`Exception Selecting`

By setting bits in the *exceptionmap* parameter, a process can determine whether certain connections have been aborted. VC sockets that reference aborted connections always exception select as "true" (their elements are set when the call completes). Exception selecting on VC sockets can also be useful when the connection associated with the socket is not fully established. For example: Process *B* has successfully created a VC socket via a call to `ipcconnect()`, but cannot know whether the connection associated with the socket is established until it calls `ipcrecv()`. If Process *B* calls `ipcrecv()` before the connection is established or before it becomes known that the connection cannot be established, it will block if the VC socket is in synchronous mode, or return a NSR_WOULD_BLOCK error if the VC socket is in asynchronous mode. Process *B* can avoid blocking in the synchronous case, or polling in the asynchronous case, by performing an exception select on the new VC socket. The socket selects as true if the connection has become "established" but `ipcrecv()` has not yet been called *or* if the attempt to connect has failed.

Parameters

sdbound (input/output parameter) Specifies the upper ordinal bound on the range of descriptors specified in the *readmap*, *writemap*, and *exceptionmap* parameters. An `ipcselect()` call is most efficient if *sdbound* is set to the ordinal value of the highest-numbered socket descriptor specified in the *map* parameters. As an output parameter, *sdbound* contains the upper ordinal boundary of all of the descriptors that met the select criteria. The maximum number of file and socket descriptors that a process can open at a time is a system-defined number (see *getrlimit*(2)).

readmap (input/output parameter) A bit map indexed with NetIPC socket descriptors. On input, this parameter specifies the socket descriptors to be examined for readability. If zero is passed, no sockets are examined. On output, *readmap* describes all readable sockets. Readability is described above.

writemap (input/output parameter) A bit map indexed with NetIPC socket descriptors. On input, this parameter specifies the socket descriptors to be examined for writeability. If zero is passed, no sockets will be examined. On output, *writemap* describes all writeable sockets. Writeablity is described above.

exceptionmap (input/output parameter) A bit map indexed with NetIPC sockets descriptors. On input, this parameter specifies the socket descriptors to be examined for exceptions. If zero is passed, no sockets will be examined. On output, *exceptionmap* describes all exceptional sockets. Exception conditions are described above.

timeout (input parameter) The number of tenths of seconds to wait. If no sockets are selectable, `ipcselect()` blocks for this amount of time. Valid values are zero, -1, or any positive integer. If *timeout* is set to zero, the call will not block. If *timeout* is set to -1, the call blocks until some event occurs. NOTE: If *timeout* is set to -1 and no bits are set in any of the bit maps, `ipcselect()` blocks indefinitely.

result (output parameter) The error code returned. Refer to ERRORS below for more information.

EXAMPLES

In the C programming language, the *readmap*, *writemap*, and *exceptionmap* parameters can be declared as *int* arrays. The size of the *map* arrays must be large enough to accommodate *sdbound*+1 bits. Thus, each *map* array must contain at least the following number of elements (where `BITS_PER_INT` is the number of bits in an `int` variable):

```
(sdbound + BITS_PER_INT) / BITS_PER_INT
```

The bits can be set to correspond to specific call or VC socket descriptors in the appropriate *map* parameter. The following example can be used to set a bit in the array. (The socket descriptor is represented by the variable *sd*, and the number of bits in an `int` variable is 32.)

```
readmap[sd/32] |= (unsigned)0x80000000 >> (sd % 32);
```

The next example can be used after an `ipcselect()` call completes to check whether or not a certain bit is set:

```
readmap[sd/32] & ((unsigned)0x80000000 >> (sd % 32))
```

RETURN VALUE

None. Errors are returned in the *result* parameter.

ERRORS

[NSR_BOUNDS_VIO]	One of the pointer arguments is invalid.
[NSR_DESC]	A socket descriptor specified in a bitmap is not valid.
[NSR_NO_ERROR]	No error occured.
[NSR_SIGNAL_INDICATION]	A signal caused the call to abort.
[NSR_SOCKET_TIMEOUT]	The timer expired before an exception was detected.
[NSR_TIMEOUT_VALUE]	The value specified in the *timeout* parameter is invalid.

AUTHOR

`ipcselect()` was developed by HP.

SEE ALSO

getrlimit(2), ipcconnect(2), ipccontrol(2), ipccreate(2), ipcdest(2), ipcgetnodename(2), ipclookup(2), ipcname(2), ipcnamerase(2), ipcrecv(2), ipcrecvcn(2), ipcsend(2), ipcsetnodename(2), ipcshutdown(2), addopt(3N), initopt(3N), ipcerrmsg(3N), optoverhead(3N), readopt(3N).

NAME

ipcsend - send data on a NetIPC socket

SYNOPSIS

```
#include <sys/ns_ipc.h>

void ipcsend(
    ns_int_t vcdesc,
    const void *data,
    ns_int_t dlen,
    ns_int_t *flags,
    short opt[],
    ns_int_t *result);
```

DESCRIPTION

`ipcsend()` is used to send data on an established connection. The data can be sent as a single contiguous buffer or as a scattered data vector. If the data is vectored, NetIPC gathers all the referenced data before sending it.

For vectored writes an iovec structure contains the data vector. An iovec structure can be defined as:

```
struct iovec {
     char       *iov_base;
     unsigned   iov_len;
};
```

and the normal type for the data argument can be replaced by:

```
struct iovec *data;
```

Each iovec entry specifies the base address and length of an area in memory where data should be accessed. `ipcsend()` always fills-in one area completely before proceeding to the next area.

`ipcsend()` behaves differently, depending on whether the referenced socket is in synchronous or asynchronous mode. These differences are as follows:

Synchronous I/O.
Send requests issued against sockets in synchronous mode may block. `ipcsend()` blocks if it cannot immediately obtain the buffer space needed to accommodate the data. The call resumes after the required buffer space becomes available or after the socket timer expires. Timeouts are 60 seconds by default, and can be altered by calling `ipccontrol()`.

Asynchronous I/O.
Send requests issued against sockets in asynchronous mode never block. If the buffer space required to accommodate the data is not immediately available, a **NSR_WOULD_BLOCK** error (code 56) is returned. After receiving this error, the process can try the call again later or determine when the socket is writeable by calling `ipcselect()`.

PARAMETERS

vcdesc	(input parameter) Socket descriptor. Refers to the virtual circuit (VC) socket endpoint of the connection through which the data will be sent. A VC socket descriptor is obtained by calling `ipcconnect()` or `ipcrecvcn()`.
data	(input parameter) A buffer to hold the data being sent, or a data vector that describes where the data to be sent is located.
dlen	(input parameter) If *data* is a data buffer, *dlen* is the length in bytes of the data in the buffer. If *data* is a data vector, *dlen* is the length in bytes of the data vector.
flags	(input parameter) See below.
opt	(input parameter) An array of options and associated information. See below.
result	(output parameter) The error code returned. Refer to ERRORS below for more information.

FLAGS PARAMETER

NSF_MORE_DATA (bit 26) (input parameter) When this bit is set, the underlying network protocol can temporarily delay sending data for efficiency reasons.

NSF_VECTORED (bit 31) (input parameter) When this bit is set, the data parameter refers to a data vector and not to a data buffer.

OPT PARAMETER

NSO_DATA_OFFSET

(*optioncode* = 8) (*datalength* = 2) A two-byte integer that indicates a byte offset from the beginning of the data buffer where the data to be sent actually begins. Only valid if the data parameter is a data buffer.

RETURN VALUE

None. Errors are returned in the *result* parameter.

ERRORS

[NSR_BOUNDS_VIO]	An address parameter is invalid.
[NSR_DESC]	The *vcdesc* parameter is not a valid descriptor.
[NSR_DLEN]	The value specified in the *dlen* parameter is invalid.
[NSR_DUP_OPTION]	The *opt* array contains duplicate information.
[NSR_FLAGS]	An illegal flag was specified.
[NSR_MSGSIZE]	An illegal data length was specified. By default, data transfer is limited to a 100 byte maximum. You can alter this limit by calling `ipccontrol()`.
[NSR_NOT_CONNECTION]	The *vcdesc* parameter is not a valid VC socket.
[NSR_OPT_OPTION]	An option in the *opt* parameter in unknown or invalid.
[NSR_OPT_SYNTAX]	A length or offset value in the *opt* parameter is invalid.
[NSR_SIGNAL_INDICATION]	The call aborted due to a signal.
[NSR_SOCKET_TIMEOUT]	The socket timer expired before the data could be transfered. By default, the socket timer is 60 seconds. This value can be altered by calling `ipccontrol()`.
[NSR_TOO_MANY_VECTS]	The maximum number of data vectors was exceeded. The limit is 16.
[NSR_VECT_COUNT]	An incorrect data length was specified for vectored data.
[NSR_WOULD_BLOCK]	The requested data cannot be sent at this time.

AUTHOR

`ipcsend()` was developed by HP.

SEE ALSO

ipcconnect(2), ipccontrol(2), ipccreate(2), ipcdest(2), ipcgetnodename(2), ipclookup(2), ipcname(2), ipcnamerase(2), ipcrecv(2), ipcrecvcn(2), ipcselect(2), ipcsetnodename(2), ipcshutdown(2), addopt(3N), initopt(3N), ipcerrmsg(3N), optoverhead(3N), readopt(3N).

NAME

ipcsetnodename - set NetIPC node name of host CPU

SYNOPSIS

```
#include <sys/ns_ipc.h>

void ipcsetnodename(
    const char *nodename,
    ns_int_t namelen,
    ns_int_t *result);
```

DESCRIPTION

`ipcsetnodename()` sets the NetIPC node name of the host processor to *nodename*, which has a length of *namelen* characters.

Super-user capability is required to use this call.

Parameters

nodename (input parameter) The ASCII-coded name that is to be assigned to this host.

namelen (output parameter) The length in bytes of the *nodename* parameter.

result (output parameter) See ERRORS below.

RETURN VALUE

None. Errors are returned in the *result* parameter.

AUTHOR

ipcsetnodname was developed by HP.

ERRORS

[NSR_NO_ERROR]	The call was successful.
[NSR_NOT_ALLOWED]	The caller does not have super-user capability.
[NSR_BOUNDS_VIO]	The input parameter address is invalid.
[NSR_NLEN]	The value of the *namelen* parameter is invalid.
[NSR_NODE_NAME_SYNTAX]	The syntax of the *nodename* parameter is illegal.

AUTHOR

`ipcsetnodename()` was developed by HP.

SEE ALSO

ipcconnect(2), ipccontrol(2), ipccreate(2), ipcdest(2), ipcgetnodename(2), ipclookup(2), ipcname(2), ipcnamerase(2), ipcrecv(2), ipcrecvcn(2), ipcselect(2), ipcsend(2), ipcshutdown(2), addopt(3N), initopt(3N), ipcerrmsg(3N), optoverhead(3N), readopt(3N).

NAME

ipcshutdown - release a NetIPC descriptor

SYNOPSIS

```
#include <sys/ns_ipc.h>

void ipcshutdown(
    ns_int_t descriptor,
    ns_int_t *flags,
    short opt[],
    ns_int_t *result);
```

DESCRIPTION

`ipcshutdown()` is used to release a descriptor. The referenced descriptor can be a call socket descriptor, virtual circuit (VC) socket descriptor, or destination descriptor. Once a descriptor has been realeased, the descriptor can no longer be used by the calling process. Since the descriptor may be shared between processes, it is destroyed only if the calling process is the last process referencing it.

When a call socket, VC socket, or destination descriptor is destroyed, all resources are released and the descriptor name(s) in the local socket registry are removed. Shutting down a VC socket does not affect any call sockets, and shutting down a call socket does not affect any VC sockets created using the call socket.

All of the data in transit on a VC socket, including any data that has already been queued on the destination VC socket, may be destroyed when the connection is shut down unless the **NSF_GRACEFUL_RELEASE** flag is set. If a process sends important data to its peer process just prior to shutting that process down, it is recommended that the calling process receive a confirmation from the peer process before calling `ipcshutdown()` or exiting, or use the **NSF_GRACEFUL_RELEASE** flag to ensure that the data was received.

PARAMETERS

descriptor	(input parameter) The descriptor to be released. Can be a call socket descriptor, VC socket descriptor, or destination descriptor.
flags	(input parameter) Must be 0 or **NSF_GRACEFUL_RELEASE**. See below.
opt	(input parameter) No options are defined for this call. Can be 0 or a pointer to an empty NetIPC option buffer.
result	(output parameter) The error code returned. Refer to ERRORS below for more information.

Flags Parameter

NSF_GRACEFUL_RELEASE
If this flag is set, the underlying network protocol can continue to transmit data after the calling process exits.

RETURN VALUE

None. Errors are returned to the *result* parameter.

ERRORS

[NSR_DESC]	The *descriptor* parameter is not a valid VC socket descriptor, call socket descriptor, or destination descriptor.
[NSR_FLAGS]	The *flags* parameter is illegal or unsupported.
[NSR_NO_ERROR]	The call was successful.
[NSR_OPT_OPTION]	An unsupported option was specified.
[NSR_OPT_SYNTAX]	A length or offset within the *opt* parameter is invalid or unsupported.

AUTHOR

`ipcshutdown()` was developed by HP.

SEE ALSO

ipcconnect(2), ipccontrol(2), ipccreate(2), ipcdest(2), ipcgetnodename(2), ipclookup(2), ipcname(2), ipcnamerase(2), ipcrecv(2), ipcrecvcn(2), ipcselect(2), ipcsend(2), ipcsetnodename(2), addopt(3N), initopt(3N), ipcerrmsg(3N), optoverhead(3N), readopt(3N).

NAME

kill, raise - send a signal to a process or a group of processes

SYNOPSIS

```
#include <signal.h>

int kill(pid_t pid, int sig);

int raise(int sig);
```

DESCRIPTION

`kill()` sends a signal to a process or a group of processes. The process or group of processes to which the signal is to be sent is specified by *pid*. The signal to be sent is specified by *sig* and is either one from the list given in *signal*(2), or `0`.

`raise()` sends signal *sig* to the executing program. The signal to be sent is specified by *sig* and is either one from the list given in *signal*(2), or `0`.

If *sig* is `0` (the null signal), error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

The real or effective user ID of the sending process must match the real or saved user ID of the receiving process unless the effective user ID of the sending process is a user who has appropriate privileges. As a single special case, the continue signal SIGCONT can be sent to any process that is a member of the same session as the sending process.

The value `KILL_ALL_OTHERS` is defined in the file `<sys/signal.h>` and is guaranteed not to be the ID of any process in the system or the negation of the ID of any process in the system.

If *pid* is greater than zero and not equal to `KILL_ALL_OTHERS`, *sig* is sent to the process whose process ID is equal to *pid*. *pid* can equal `1` unless *sig* is `SIGKILL` or `SIGSTOP`.

If *pid* is `0`, *sig* is sent to all processes excluding special system processes whose process group ID is equal to the process group ID of the sender.

If *pid* is `-1` and the effective user ID of the sender is not a user who has appropriate privileges. *sig* is sent to all processes excluding special system processes whose real or saved user ID is equal to the real or effective user ID of the sender.

If *pid* is `-1` and the effective user ID of the sender is a user who has appropriate privileges, *sig* is sent to all processes excluding special system processes.

If *pid* is `KILL_ALL_OTHERS`, `kill()` behaves much as when *pid* is equal to `-1`, except that *sig* is not sent to the calling process.

If *pid* is negative but not `-1` or `KILL_ALL_OTHERS`, *sig* is sent to all processes (excluding special system processes) whose process group ID is equal to the absolute value of *pid*, and whose real and/or effective user ID meets the constraints described above for matching user IDs.

RETURN VALUE

Upon successful completion, a value of `0` is returned. Otherwise, a value of `-1` is returned and `errno` is set to indicate the error.

ERRORS

`kill()` fails and no signal is sent if one or more of the following is true:

[EINVAL]	*sig* is neither a valid signal number nor zero.
[EINVAL]	*sig* is SIGKILL or SIGSTOP and *pid* is `1` (proc1).
[EPERM]	The user ID of the sending process is not a user who has appropriate privileges and its real or effective user ID does not match the real or saved user ID of the receiving process.
[EPERM]	The sending and receiving processes are not in the same session.
[ESRCH]	No process or process group can be found corresponding to that specified by *pid*.

`raise()` fails and no signal is sent if the following is true:

[EINVAL] *sig* is not a valid signal number or zero.

AUTHOR

`kill()` was developed by HP, AT&T, and the University of California, Berkeley.

SEE ALSO

kill(1), getpid(2), setpgrp(2), signal(2), privilege(5).

STANDARDS CONFORMANCE

`kill()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

`raise()`: AES, XPG4, ANSI C

NAME

link - link to a file

SYNOPSIS

```
#include <unistd.h>

int link(const char *path1, const char *path2);
```

DESCRIPTION

`link()` creates a new link (directory entry) for the existing file. *path1* points to a path name naming an existing file. *path2* points to a path name naming the new directory entry to be created.

RETURN VALUE

Upon successful completion, `link()` returns 0; otherwise, it returns −1 and sets `errno` to indicate the error.

ERRORS

`link()` fails and no link is created if one or more of the following is true:

[EACCES]	A component of either path prefix denies search permission.
[EACCES]	The requested link requires writing in a directory that does not permit writing.
[EDQUOT]	User's disk quota block limit has been reached for this file system.
[EEXIST]	The link named by *path2* exists.
[ENOENT]	The file named by *path1* does not exist.
[ENOENT]	A component of either path prefix does not exist.
[ENOENT]	*path2* points to a null path name.
[ENOSPC]	The directory to contain the file cannot be extended.
[ENOTDIR]	A component of either path prefix is not a directory.
[EPERM]	The file named by *path1* is a directory and the effective user ID is not a user who has appropriate privileges.
[EXDEV]	The link named by *path2* and the file named by *path1* are on different logical devices (file systems).
[EROFS]	The requested link requires writing in a directory on a read-only file system.
[EFAULT]	*path* points outside the allocated address space of the process. The reliable detection of this error is implementation dependent.
[ENOENT]	*path1* or *path2* is null.
[EMLINK]	The maximum number of links to a file would be exceeded.
[ENAMETOOLONG]	Either specified path exceeds **PATH_MAX** bytes, or a component of either specified path exceeds **NAME_MAX** while **POSIX_NO_TRUNC** is in effect.
[ELOOP]	Too many symbolic links were encountered in translating either path name.

DEPENDENCIES

Series 300, 400, and 700:

If *path2* names a symbolic link, `link()` fails without creating the link, −1 is returned, and `errno` is set to:

[EEXIST]	*path2* names a symbolic link.

SEE ALSO

cp(1), link(1M), symlink(2), symlink(4), unlink(2).

STANDARDS CONFORMANCE

`link()`: AES [Series 300/400/700 only], SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

listen - listen for connections on a socket

SYNOPSIS

```
int listen(int s, int backlog);
```

DESCRIPTION

To accept connections, a socket is first created using **socket()**, a queue for incoming connections is specified using **listen()**, and then connections are accepted using **accept()**. **listen()** applies only to unconnected sockets of type SOCK_STREAM. If the socket has not been bound to a local port before **listen()** is invoked, the system automatically binds a local port for the socket to listen on (see *inet*(7F)). For sockets in the address family AF_CCITT, the socket *must* be bound to an address by using **bind()** before connection establishment can continue, otherwise an EADDREQUIRED error is returned.

The listen queue is established for the socket specified by the *s* parameter, which is a socket descriptor.

backlog defines the maximum allowable length of the queue for pending connections. If a connection request arrives when the queue is full, the client receives an ETIMEDOUT error.

backlog is currently limited (silently) to be in the range of 1 to 20. If any other value is specified, the system automatically assigns the closest value within range.

DEPENDENCIES

AF_CCITT:

Call-acceptance can be controlled by the **X25_CALL_ACPT_APPROVAL** **ioctl()** call described in RETURN VALUE Upon successful completion, **listen()** returns 0; otherwise, it returns -1 and sets **errno** to indicate the error.

ERRORS

listen() fails if any of the following conditions are encountered:

[EBADF]	The argument *s* is not a valid descriptor.
[EDESTADRREQ]	No bind address was established.
[ENOTSOCK]	The argument *s* is not a socket.
[EOPNOTSUPP]	The socket is not of a type that supports the **listen()** operation.
[ENOBUFS]	(Series 300/400 only) No buffer space is available. **listen()** cannot be started at this time.
[EINVAL]	The socket has been shut down or is already connected (see *socketx25*(7)).

AUTHOR

listen() was developed by the University of California, Berkeley.

SEE ALSO

accept(2), connect(2), socket(2), socketx25(7), af_ccitt(7F), inet(7F).

NAME

lockf - provide semaphores and record locking on files

SYNOPSIS

```
#include <unistd.h>

int lockf(int fildes, int function, off_t size);
```

DESCRIPTION

`lockf()` allows regions of a file to be used as semaphores (advisory locks) or restricts access to only the locking process (enforcement-mode record locks). Other processes that attempt to access the locked resource either return an error or sleep until the resource becomes unlocked. All locks for a process are released upon the first close of the file, even if the process still has the file opened, and all locks held by a process are released when the process terminates.

fildes is an open file descriptor. The file descriptor must have been opened with write-only permission (`O_WRONLY`) or read-write permission (`O_RDWR`) in order to establish a lock with this function call (see *open*(2)).

If the calling process is a member of a group that has the `PRIV_LOCKRDONLY` privilege (see *setprivgrp*(2)), it can also use `lockf()` to lock files opened with read-only permission (`O_RDONLY`).

function is a control value that specifies the action to be taken. Permissible values for *function* are defined in <`unistd.h`> as follows:

```
#define F_ULOCK    0       /* unlock a region */
#define F_LOCK     1       /* lock a region */
#define F_TLOCK    2       /* test and lock a region */
#define F_TEST     3       /* test region for lock */
```

All other values of *function* are reserved for future extensions and result in an error return if not implemented.

`F_TEST` is used to detect whether a lock by another process is present on the specified region. `lockf()` returns zero if the region is accessible and `-1` if it is not; in this case `errno` is set to EACCES. `F_LOCK` and `F_TLOCK` both lock a region of a file if the region is available. `F_ULOCK` removes locks from a region of the file.

size is the number of contiguous bytes to be locked or unlocked. The resource to be locked starts at the current offset in the file, and extends forward for a positive *size*, and backward for a negative *size* (the preceding bytes up to but not including the current offset). If *size* is zero, the region from the current offset through the end of the largest possible file is locked (that is, from the current offset through the present or any future end-of-file). An area need not be allocated to the file in order to be locked, because such locks can exist past the end of the file.

Regions locked with `F_LOCK` or `F_TLOCK` can, in whole or in part, contain or be contained by a previously locked region for the same process. When this occurs or if adjacent regions occur, the regions are combined into a single region. If the request requires that a new element be added to the table of active locks but the table is already full, an error is returned, and the new region is not locked.

`F_LOCK` and `F_TLOCK` requests differ only by the action taken if the resource is not available: `F_LOCK` causes the calling process to sleep until the resource is available, whereas `F_TLOCK` returns an EACCES error if the region is already locked by another process.

`F_ULOCK` requests can, in whole or part, release one or more locked regions controlled by the process. When regions are not fully released, the remaining regions are still locked by the process. Releasing the center section of a locked region requires an additional element in the table of active locks. If this table is full, an EDEADLK error is returned, and the requested region is not released.

Regular files with the file mode of `S_ENFMT`, not having the group execute bit set, will have an enforcement policy enabled. With enforcement enabled, reads and writes that would access a locked region sleep until the entire region is available if `O_NDELAY` is clear, but return -1 with `errno` set if `O_NDELAY` is set. File access by other system functions, such as `exec()`, are not subject to the enforcement policy. Locks on directories, pipes, and special files are advisory only; no enforcement policy is used.

A potential for deadlock occurs if a process controlling a locked resource is put to sleep by accessing the locked resource of another process. Thus, calls to `fcntl()`, `lockf()`, `read()`, or `write()` (see

fcntl(2), *lockf*(2), *read*(2), and *write*(2)) scan for a deadlock prior to sleeping on a locked resource. Deadlock is not checked for the `wait()` and `pause()` system calls (see *wait*(2) and *pause*(2)), so potential for deadlock is not eliminated. A `creat()` call or an `open()` call with the `O_CREATE` and `O_TRUNC` flags set on a regular file returns error EAGAIN if another process has locked part of the file and the file is currently in enforcement mode.

NETWORKING FEATURES

NFS

The advisory record-locking capabilities of `lockf()` are implemented throughout the network by the "network lock daemon" (see *lockd*(1M)). If the file server crashes and is rebooted, the lock daemon attempts to recover all locks associated with the crashed server. If a lock cannot be reclaimed, the process that held the lock is issued a `SIGLOST` signal.

Only advisory record locking is implemented for NFS files.

RETURN VALUE

Upon successful completion, a value of `0` is returned. Otherwise, a value of `-1` is returned and `errno` is set to indicate the error.

ERRORS

`lockf()` fails if any of the following occur:

[EACCES]	*function* is `F_TLOCK` or `F_TEST` and the region is already locked by another process.
[EBADF]	*fildes* is not a valid, open file descriptor.
[EDEADLK]	A deadlock would occur or the number of entries in the system lock table would exceed a system-dependent maximum. HP-UX guarantees this value to be at least 50.
[EAGAIN]	*function* is `F_LOCK` or `F_TLOCK` and the file is mapped in to virtual memory via the `mmap()` system call (see *mmap*(2)).
[EINTR]	A signal was caught during the `lockf()` system call.
[EINVAL]	*function* is not one of the functions specified above.
[EINVAL]	*size* plus current offset produces a negative offset into the file.
[ENOLCK]	*function* is `F_TLOCK` or `F_LOCK` and the file is an NFS file with access bits set for enforcement mode.
[ENOLCK]	The file is an NFS file and a system error occurred on the remote node.

WARNINGS

Deadlock conditions may arise when either the `wait()` or `pause()` system calls are used in conjunction with enforced locking; see *wait*(2) and *pause*(2) for details.

File and record locking using file descriptors obtained through `dup()` or `link()` may not work as expected (see *dup*(2) or *link*(2)). For example, unlocking regions that were locked using either file descriptor may also unlock regions that were locked using the other file descriptor.

Unexpected results may occur in processes that use buffers in the user address space. The process may later read or write data which is or was locked. The standard I/O package, *stdio*(3S), is the most common source of unexpected buffering.

In a hostile environment, locking can be misused by holding key public resources locked. This is particularly true with public read files that have enforcement enabled.

It is not recommended that the `PRIV_LOCKRDONLY` capability be used because it is provided for backward compatibility only. This feature may be modified or dropped from future HP-UX releases.

Locks default to advisory mode unless the `setgid` bit of the file permissions is set.

APPLICATION USAGE

Because in the future the variable `errno` will be set to EAGAIN rather than EACCES when a section of a file is already locked by another process, portable application programs should expect and test for either value. For example:

```
if (lockf(fd, F_TLOCK, siz) == -1)
    if ((errno == EAGAIN) || (errno == ACCES))
    /*
    * section locked by another process
    * check for either EAGAIN or EACCES
    * due to different implementations
    */
    else if ...
    /*
    * check for other errors
    */
```

SEE ALSO

lockd(1M), statd(1M), chmod(2), close(2), creat(2), fcntl(2), open(2), pause(2), read(2), stat(2), wait(2), write(2), unistd(5).

FUTURE DIRECTIONS

The error condition that currently sets `errno` to EACCES will instead set `errno` to EAGAIN (see also APPLICATION USAGE above).

STANDARDS CONFORMANCE

`lockf()`: SVID2, XPG2

NAME

lseek - move read/write file pointer; seek

SYNOPSIS

```
#include <unistd.h>

off_t lseek(int fildes, off_t offset, int whence);
```

DESCRIPTION

`lseek()` sets the file pointer associated with the file descriptor as follows:

- If *whence* is **SEEK_SET**, the pointer is set to *offset* bytes.
- If *whence* is **SEEK_CUR**, the pointer is set to its current location plus *offset*.
- If *whence* is **SEEK_END**, the pointer is set to the size of the file plus *offset*.

These symbolic constants are defined in `<unistd.h>`.

RETURN VALUE

When `lseek()` completes successfully, it returns an integer, which is the resulting file offset as measured in bytes from the beginning of the file. Otherwise, a value of `-1` is returned and `errno` is set to indicate the error.

For all files that are not character or block special files, the integer returned on successful completion is non-negative. For character or block special files that correspond to disk sections larger than 2 gigabytes, a non-negative integer is returned for successful seeks beyond 2 gigabytes. This value is the resulting file offset as measured in bytes from the beginning of the file, when taken as an unsigned value. `-1` always indicates an error return, even when encountered on greater than 2 gigabyte disk sections.

ERRORS

`lseek()` fails and the file offset remains unchanged if one or more of the following is true:

[EBADF]	*fildes* is not an open file descriptor.
[ESPIPE]	*fildes* is associated with a pipe or FIFO.
[EINVAL]	*whence* is not one of the supported values.
[EINVAL]	The resulting file offset would be negative.

WARNINGS

Some devices are incapable of seeking. The value of the file offset associated with such a device is undefined.

Using `lseek()` with a *whence* of **SEEK_END** on device special files is not supported and the results are not defined.

SEE ALSO

creat(2), dup(2), fcntl(2), open(2), unistd(5).

STANDARDS CONFORMANCE

`lseek()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

madvise - advise the system of a process' expected paging behavior

SYNOPSIS

```
#include <sys/mman.h>

int madvise(
    caddr_t addr,
    size_t len,
    int behav );
```

DESCRIPTION

madvise permits a process to advise the system about its expected future behavior in referencing a mapped file or anonymous memory region. Certain implementations may use this information to optimize use of resources.

addr and *len* specify the address and length in bytes of the region to which the advice refers. If these are not the address and length of a region created by a successful call to **mmap()**, **madvise()** fails with an EINVAL error.

The *behav* argument is constructed from the bitwise inclusive OR of one or more of the following flags defined in the header **<sys/mman.h>**:

MADV_NORMAL	No further special treatment.
MADV_RANDOM	Expect random page references.
MADV_SEQUENTIAL	Expect sequential page references.
MADV_WILLNEED	Will need these pages.
MADV_DONTNEED	Will not need these pages.
MADV_SPACEAVAIL	Ensure that resources are reserved.

IMPLEMENTATION NOTES

The current implementation of **madvise()** is a null operation.

RETURN VALUE

madvise() returns 0 upon success; otherwise, it returns –1 and sets **errno** to indicate the error.

ERRORS

madvise() fails if any of the following conditions are encountered:

[EFAULT]	The range specified by (*addr*, *addr+len*) is invalid for a process' address space.
[EINVAL]	*addr* is not a multiple of the page size as returned by **sysconf(_SC_PAGE_SIZE)**, or *behav* contains invalid values or incompatible combinations of flags.
[EINVAL]	The address range specified by *addr* and *len* was not created by a successful call to **mmap()**.

AUTHOR

madvise() was developed by HP and OSF.

SEE ALSO

mmap(2), sysconf(2).

STANDARDS CONFORMANCE

madvise(): AES

NAME

mkdir - make a directory file

SYNOPSIS

```
#include <sys/stat.h>

int mkdir(const char *path, mode_t mode);
```

DESCRIPTION

`mkdir()` creates a new directory file named by *path*. The file permission bits of the new directory are initialized from *mode*, and are modified by the process's file mode creation mask. For each bit set in the process's file mode creation mask, the corresponding bit in the new directory's mode is cleared (see *umask*(2)).

The directory's owner ID is set to the process's effective-user-ID. If the set-group-ID bit of the parent directory is set, the directory's group ID is set to group ID of the parent directory. Otherwise, the directory's group ID is set to the process's effective-group-ID. The set-group-ID bit of the new directory is set to the same value as the set-group-ID bit of the parent directory.

Symbolic constants defining the access permission bits are found in the `<sys/stat.h>` header and are used to construct the argument *mode*. The value of the argument *mode* is the bit-wise inclusive OR of the values of the desired permissions.

`S_IRUSR`	Read by owner.
`S_IWUSR`	Write by owner.
`S_IXUSR`	Execute (search) by owner.
`S_IRGRP`	Read by group.
`S_IWGRP`	Write by group.
`S_IXGRP`	Execute (search) by group.
`S_IROTH`	Read by others (that is, anybody else).
`S_IWOTH`	Write by others.
`S_IXOTH`	Execute (search) by others.

Access Control Lists (ACLs)

On systems implementing access control lists, the directory is created with three base ACL entries, corresponding to the file access permission bits (see *acl*(5)).

RETURN VALUE

Upon successful completion, `mkdir()` returns a value of `0`; a return value of `-1` indicates an error, and an error code is stored in `errno`.

ERRORS

`mkdir()` fails and no directory is created if any of the following is true:

[EACCES] A component of the path prefix denies search permission.

[EACCES] The parent directory of the new directory denies write permission.

[EEXIST] The named file already exists.

[EFAULT] *path* points outside the process's allocated address space. The reliable detection of this error is implementation dependent.

[EIO] An I/O error occurred while writing to the file system.

[ELOOP] Too many symbolic links are encountered in translating the path name.

[EMLINK] The maximum number of links to the parent directory, `{LINK_MAX}`, would be exceeded.

[ENAMETOOLONG] The length of the specified path name exceeds `PATH_MAX` bytes, or the length of a component of the path name exceeds `NAME_MAX` bytes while `_POSIX_NO_TRUNC` is in effect.

[ENOENT] A component of the path prefix does not exist.

[ENOSPC] Not enough space on the file system.

[ENOTDIR]	A component of the path prefix is not a directory.
[EROFS]	The named file resides on a read-only file system.
[EDQUOT]	User's disk quota block or inode limit has been reached for this file system.

WARNINGS

Access Control Lists

Access control list descriptions in this entry apply only to standard HP-UX operating systems. If HP-UX BLS software has been installed, access control lists are handled differently. Refer to HP-UX BLS documentation for information about access control lists in the HP-UX BLS environment.

AUTHOR

`mkdir()` was developed by the University of California, Berkeley.

SEE ALSO

chmod(2), setacl(2), stat(2), umask(2), acl(5), limits(5).

STANDARDS CONFORMANCE

`mkdir()`: AES, SVID2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

mknod, mkrnod - make a directory, or a special or regular file

SYNOPSIS

```
#include <sys/stat.h>

int mknod(const char *path, mode_t mode, dev_t dev);

int mkrnod(
    const char *path,
    mode_t mode,
    dev_t dev,
    cnode_t cnodeid
);
```

DESCRIPTION

mknod() creates a new file named by the path name pointed to by *path*. The mode of the new file is specified by the *mode* argument. **mkrnod()** is the same as **mknod()** but is used to make device files that can be accessed from a different cnode in the cluster as identified by the additional parameter *cnodeid*. A *cnodeid* value of **0** creates a "generic" device file that can be accessed by any cnode.

Symbolic constants defining the file type and file access permission bits are found in the **<sys/stat.h>** header file and are used to construct the *mode* argument. The value of the *mode* argument should be the bit-wise inclusive OR of the values of the desired file type, miscellaneous mode bits, and access permissions. See *stat*(5) for a description of the components of the file mode.

The owner ID of the file is set to the effective-user-ID of the process. If the set-group-ID bit of the parent directory is set, the new file's group ID is set to the group ID of the parent directory. Otherwise, the new file's group ID is set to the effective-group-ID of the process.

The file access permission bits of *mode* are modified by the process's file mode creation mask: for each bit set in the process's file mode creation mask, the corresponding bit in the file's mode is cleared (see *umask*(2)).

The new file is created with three base access-control-list (ACL) entries, corresponding to the file access permission bits (see *acl*(5)).

The *dev* argument is meaningful only if *mode* indicates a block or character special file, and is ignored otherwise. It is an implementation- and configuration-dependent specification of a character or block I/O device. The value of *dev* is created by using the **makedev()** macro defined in **<sys/mknod.h>**. The **makedev()** macro takes as arguments the major and minor device numbers, and returns a device identification number which is of type **dev_t**. The value and interpretation of the major and minor device numbers are implementation-dependent. For more information, see *mknod*(5) and the System Administration manuals for your system.

Only users having appropriate privileges can invoke **mknod()** for file types other than FIFO files.

WARNINGS

Proper discretion should be used when using **mkrnod()** to create generic device files in an HP Clustered Environment. A generic device file accessed from different cnodes in a cluster applies to different physical devices. Thus the file's ownership and permissions might not be appropriate in the context of every individual cnode in the cluster.

RETURN VALUE

Upon successful completion a value of **0** is returned. Otherwise, a value of –1 is returned and **errno** is set to indicate the error.

ERRORS

mknod() fails and the new file is not created if:

[EACCES]	The directory in which *path* would be created denies write permission, *mode* is for a FIFO file and the caller does not have appropriate privileges.
[EACCES]	A component of the path prefix denies search permission.
[EEXIST]	The named path already exists.

[EFAULT] The *path* argument points outside the process's allocated address space. The reliable detection of this error is implementation dependent.

[ELOOP] Too many symbolic links were encountered in translating the path name.

[ENAMETOOLONG] The length of the specified path name exceeds **PATH_MAX** bytes, or the length of a component of the path name exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect.

[ENOENT] The *path* argument is null.

[ENOENT] A component of the path prefix does not exist.

[ENOSPC] Not enough space on the file system.

[ENOTDIR] A component of the path prefix is not a directory.

[EPERM] The effective-user-ID of the process does not match that of a user who has appropriate privileges, and the file type is not FIFO special.

[EROFS] The directory in which the file is to be created is located on a read-only file system.

[EDQUOT] User's disk quota block or inode limit has been reached for this file system.

AUTHOR

mknod( ) was developed by AT&T and HP.

SEE ALSO

mknod(1M), chmod(2), exec(2), mkdir(2), setacl(2), umask(2), cdf(4), fs(4), acl(5), mknod(5), stat(5), types(5), privilege(5).

STANDARDS CONFORMANCE

mknod( ): SVID2, XPG2

NAME

mmap - map object into virtual memory

SYNOPSIS

```
#include <sys/mman.h>

caddr_t mmap(
    caddr_t addr,
    size_t len,
    int prot,
    int flags,
    int fildes,
    off_t off);
```

DESCRIPTION

mmap() creates a new memory mapped file or anonymous memory region. The format of the call is as follows:

pa = mmap(*addr*, *len*, *prot*, *flags*, *fildes*, *off*);

mmap() establishes a mapping between the process's address space at an address *pa* for *len* bytes to an object represented by the file descriptor *fildes* at offset *off* for *len* bytes, or to an anonymous region of physical memory of size *len* bytes. A successful **mmap()** call returns *pa* as its result, where *pa* is an implementation-dependent function of the requested starting address and length for the new region, *addr* and *len*, as further described below.

If *len* is not a multiple of the page size returned by **sysconf(_SC_PAGE_SIZE)**, then references are permitted to an address between *pa+len* and the next higher address that is an integer multiple of the page size; however, the results of any such references are undefined.

The *flags* argument specifies the attributes of the region. Values of the *flags* argument are constructed by bitwise-inclusive ORing flags from the following list of symbolic names defined in **<sys/mman.h>**:

MAP_FILE	Create a mapped file region.
MAP_ANONYMOUS	Create an unnamed memory region.
MAP_VARIABLE	Place region at implementation-computed address.
MAP_FIXED	Place region at specified address.
MAP_SHARED	Share changes between processes and underlying file object, if any.
MAP_PRIVATE	Changes are private to a process.

The **MAP_FILE** and **MAP_ANONYMOUS** flags control whether the region to be mapped is a mapped file region or an anonymous shared memory region. Exactly one of these flags must be selected.

If **MAP_FILE** is set in *flags*:

- A new mapped file region is created, mapping the file associated with *fildes*.
- *off* specifies the file byte offset at which the mapping starts. This offset must be a multiple of the page size returned by **sysconf(_SC_PAGE_SIZE)**.
- If the end of the mapped file region is beyond the end of the file, any reference to an address in the mapped file region corresponding to an offset beyond the end of the file results in the delivery of a **SIGBUS** signal to the process, unless the address lies in the last partial page corresponding to the range beyond the end of the file. The last partial page mapping the range beyond the end of the file is always initialized to zeros, and any modified portions of the last page of a file which are beyond its end are not written back to the file.

If **MAP_ANONYMOUS** is set in *flags*:

- A new memory region is created and initialized to all zeros. This memory region can be shared only with descendants of the current process.
- If the *fildes* argument is not –1, an EINVAL error is generated.

- The value of *off* is meaningless because there is no underlying file object for the memory region.

The **MAP_VARIABLE** and **MAP_FIXED** flags control the placement of the region as described below. Exactly one of these flags must be selected.

If **MAP_VARIABLE** is set in *flags*:

- If the requested address is NULL, or if it is not possible for the system to place the region at the requested address, the region is placed at an address selected by the system. If the requested address is not a multiple of the page size returned by **sysconf(_SC_PAGE_SIZE)**, the system treats the address as if it were rounded up to the next larger page size multiple.

If **MAP_FIXED** is set in *flags*:

- If it is not possible for the system to place the region at the requested address, the **mmap()** function fails.
- *addr* must be a multiple of the page size returned by **sysconf(_SC_PAGE_SIZE)**.

A region is never placed at an address where it would overlap with an existing region or a portion of the process address space that is already in use or reserved for other purposes. A region is always placed at a starting address that is an exact multiple of the page size returned by **sysconf(_SC_PAGE_SIZE)**.

The **MAP_PRIVATE** and **MAP_SHARED** flags control the visibility of modifications to the mapped file or anonymous memory region. Exactly one of these flags must be selected.

If **MAP_SHARED** is set in *flags*:

- Modifications to the region are visible to other processes which have mapped the same file using **MAP_SHARED**.
- If the region is a mapped file region, modifications to the region are written to the underlying file.

If **MAP_PRIVATE** is set in *flags*:

- Modification to the mapped region by the calling process is not visible to other processes which have mapped the same region using either **MAP_PRIVATE** or **MAP_SHARED**. Modifications are not visible to descendant processes that have inherited the mapped region across a **fork()**.
- If the region is a mapped file region, modifications to to the region are not written to the underlying file.

It is unspecified whether modifications by processes that have mapped a file using **MAP_SHARED** are visible to other processes that have mapped the same file using **MAP_PRIVATE**.

The *prot* argument specifies the mapped region's access permissions. Header file **<sys/mman.h>** defines the following access permissions:

PROT_READ	Region can be read
PROT_WRITE	Region can be written
PROT_EXEC	Region can be executed
PROT_NONE	Region cannot be accessed

The *prot* argument can be **PROT_NONE**, or any combination of **PROT_READ**, PROT_WRITE , and **PROT_EXEC** OR-ed together. If **PROT_NONE** is not specified, the system may grant other access permissions to the region in addition to those explicitly requested, except that write access will not be granted unless **PROT_WRITE** is specified.

mmap() cannot create a mapped file region unless the file descriptor used to map the file is open for reading. For a mapped file region that is mapped with **MAP_SHARED**, **mmap()** grants write access permission only if the file descriptor is open for writing. If a region was mapped with either **MAP_PRIVATE** or **MAP_ANONYMOUS**, **mmap()** grants all requested access permissions.

After the successful completion of **mmap()**, *fildes* can be closed without effect on the mapped region or on the contents of the mapped file. Each mapped region creates a file reference, similar to an open file descriptor, that prevents the file data from being deallocated.

Whether modifications made to the file using **write()** are visible to mapped regions, and whether modification to a mapped region are visible with **read()**, is undefined except for the effect of **msync()**.

If an enforcement-mode file lock is in effect for any range of a file, a call to **mmap()** to map any range of the file with access rights that would violate the lock fails. The **msem_lock()** and **msem_unlock()** semaphore interfaces can be used to coordinate shared access to a region created with the **MAP_SHARED** flag. The advisory locks of the **lockf()** or **fcntl()** interfaces have no effect on memory mapped access, but they can be used to coordinate shared access to a **MAP_SHARED** mapped file region.

For a memory mapped file, the **st_atime** and **st_mtime** values returned by **stat()** are updated when a page in the memory mapped region is read from or written to the file system.

After a call to **fork()**, the child process inherits all mapped regions with the same data and the same sharing and protection attributes as in the parent process. Each mapped file and anonymous memory region created with **mmap()** is unmapped upon process exit, and by a successful call to any of the **exec** functions.

A **SIGBUS** signal is delivered to a process when a write reference to a mapped file region would cause a file system error condition such as exceeding quota or file system space limits.

A **SIGBUS** signal is delivered to a process upon a write reference to a region without **PROT_WRITE** protection, or any reference to a region with **PROT_NONE** protection.

A call to **mmap()** with **PROT_EXECUTE** specified, but without **PROT_WRITE** specified for a **MAP_SHARED|MAP_FILE** mapping is treated by the system as the execution of the underlying file. This implies that such a call fails if the file is currently open for writing or mapped with **MAP_SHARED|PROT_WRITE** options by any process, and that if the call succeeds, the file cannot be opened for writing or subsequently mapped with **MAP_SHARED|PROT_WRITE** options as long as such mappings are present. A file's status as an active executable file is determined only at the time of an **exec()**, **exit()**, or **mmap()** operation. **mprotect()** operations on a **MAP_FILE|MAP_SHARED** mapping have no effect on the underlying file's status as an active executable file.

IMPLEMENTATION NOTES

Only regular files (not directories, named pipes, or device special files) can be mapped.

System swap resources are reserved for all mappings created with either **MAP_PRIVATE** or **MAP_ANONYMOUS**.

RETURN VALUE

Upon successful completion, **mmap()** returns the address at which the mapping was placed. Otherwise, **mmap()** returns −1 and sets **errno** to indicate the error.

ERRORS

mmap() fails if any of the following conditions are encountered:

[EACCESS] The file referred to by *fildes* is not open for read access, or the file is not open for write access and **PROT_WRITE** was set for a **MAP_SHARED** mapping operation, or **PROT_EXECUTE** was set for a **MAP_SHARED** mapping operation and the underlying file does not have execute permission.

[EBADF] *fildes* is not a valid file descriptor.

[EINVAL] *flags* or *prot* is invalid, or *addr* (with **MAP_FIXED** set) or *off* (with **MAP_FILE** set) is not a multiple of the page size returned by **sysconf(_SC_PAGE_SIZE)**.

[ENODEV] *fildes* refers to an object that cannot be mapped, such as a terminal.

[ENOMEM] There is not enough address space to map *len* bytes, or **MAP_FIXED** was set and part of the address space range [*addr*, *addr+len*) (from, and including, *addr* to, but not including, *addr+len*) is not available for use.

[ENXIO] The addresses specified by the range [*off*, *off+len*) (from, and including, *off* to, but not including, *off+len*) are invalid for *fildes*.

[EAGAIN] The file represented by *fildes* has enforcement-mode file locking in effect for some range in the file. (see *lockf*(2), or *fcntl*(2)).

[ETXTBSY] **MAP_SHARED** and **MAP_FILE** are set, and **PROT_EXECUTE** is set and **PROT_WRITE** is not set, and the file being mapped is currently open for writing.

DEPENDENCIES

Series 700/800

Because the PA-RISC memory architecture utilizes a globally shared virtual address space between processes, and discourages multiple virtual address translations to the same physical address, all concurrently existing **MAP_SHARED** mappings of a file range must share the same virtual address offsets and hardware translations. PA-RISC-based HP-UX systems allocate virtual address ranges for shared memory and shared mapped files in the range 0x80000000 through 0xefffffff. This address range is used globally for *all* memory objects shared between processes.

This implies the following:

- Any single range of a file cannot be mapped multiply into different virtual address ranges.
- After the initial **MAP_SHARED** **mmap()** of a file range, all subsequent **MAP_SHARED** calls to **mmap()** to map the same range of a file must either specify **MAP_VARIABLE** in *flags* and inherit the virtual address range the system has chosen for this range, or specify **MAP_FIXED** with an *addr* that corresponds exactly to the address chosen by the system for the initial mapping. Only after all mappings for a file range have been destroyed can that range be mapped to a different virtual address.
- In most cases, two separate calls to **mmap()** cannot map overlapping ranges in a file. The virtual address range reserved for a file range is determined at the time of the initial mapping of the file range into a process address space. The system allocates only the virtual address range necessary to represent the initial mapping. As long as the initial mapping exists, subsequent attempts to map a different file range that includes any portion of the initial range may fail with an ENOMEM error if an extended contiguous address range that preserves the mappings of the initial range cannot be allocated.
- Separate calls to **mmap()** to map contiguous ranges of a file do not necessarily return contiguous virtual address ranges. The system may allocate virtual addresses for each call to **mmap()** on a first available basis.
- The use of **MAP_FIXED** is strongly discouraged because it is not portable. Using **MAP_FIXED** is generally unsuccessful on this implementation, and when it is successful, it may prevent the system from optimally allocating virtual address space.

The following combinations of protection modes are supported:

```
PROT_NONE
PROT_READ
PROT_READ|PROT_EXECUTE
PROT_READ|PROT_WRITE
PROT_READ|PROT_WRITE|PROT_EXECUTE
```

If a **MAP_PRIVATE** mapping is created of a file for which a **MAP_SHARED** mapping exists, a separate copy of a page for the **MAP_PRIVATE** mapping is created at the time of the first access to the page through the private mapping.

Series 300/400

The following combinations of protection modes are supported:

```
PROT_NONE
PROT_READ
PROT_READ|PROT_EXECUTE
PROT_READ|PROT_WRITE
PROT_READ|PROT_WRITE|PROT_EXECUTE
```

In addition, for protection modes that do not explicitly have **PROT_EXECUTE** set, individual pages within the region can be promoted to include **PROT_EXECUTE** permissions simply by executing code located within the region.

If a **MAP_PRIVATE** mapping is created of a file for which a **MAP_SHARED** mapping exists, a separate copy of a page for the **MAP_PRIVATE** mapping is created at the time of the first write reference to the page through the private mapping.

HP Clustered Environment

In a clustered environment, modifications to a **MAP_SHARED** mapped file region on one cluster node may not be visible to other processes on other cluster nodes that have the same file mapped with the **MAP_SHARED** option.

AUTHOR

mmap( ) was developed by HP, AT&T, and OSF.

SEE ALSO

fcntl(2), fork(2), ftruncate(2), lockf(2), madvise(2), mprotect(2), msem_init(2), msem_lock(2), msem_unlock(2), msync(2), munmap(2), sysconf(2), mman(5), stat(5).

STANDARDS CONFORMANCE

mmap( ): AES [Series 300/400/700 only]

NAME

mount - mount a file system

SYNOPSIS

```
#include <sys/mount.h>

int mount(const char *spec, const char *dir, int rwflag);
```

DESCRIPTION

`mount ( )` requests that a removable file system contained on the block special device file identified by *spec* be mounted on the directory identified by *dir*. *spec* and *dir* are pointers to path names.

Upon successful completion, references to the file *dir* refer to the root directory on the mounted file system.

The low-order bit of *rwflag* is used to control write permission on the mounted file system. If it is `1`, writing is forbidden; otherwise, writing is permitted according to individual file accessibility.

`mount ( )` can be invoked only by a user who has appropriate privileges.

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

ERRORS

`mount ( )` fails if one or more of the following is true:

[EPERM] The effective user ID is not a user who has appropriate privileges.

[ENOENT] The named file does not exist (for example, *path* is null or a component of *path* does not exist).

[ENOTDIR] A component of a path prefix is not a directory.

[ENOTBLK] *spec* is not a block special device.

[ENXIO] The device associated with *spec* does not exist.

[ENOTDIR] *dir* is not a directory.

[EFAULT] *spec* or *dir* points outside the allocated address space of the process. The reliable detection of this error is implementation dependent.

[EBUSY] *dir* is currently mounted on, is someone's current working directory, or is otherwise busy.

[EBUSY] The device associated with *spec* is currently mounted.

[EBUSY] There are no more mount table entries.

[ENOENT] *spec* or *dir* is null.

[EACCES] A component of the path prefix denies search permission.

[ENAMETOOLONG] The length of a specified path name exceeds `PATH_MAX` bytes, or the length of a component of the path name exceeds `NAME_MAX` bytes while `_POSIX_NO_TRUNC` is in effect.

[ELOOP] Too many symbolic links were encountered in translating either path name.

WARNINGS

If `mount ( )` is called from the program level (i.e. not called from *mount*(1M)), the table of mounted devices contained in `/etc/mnttab` is not updated. Updating of `/etc/mnttab` is performed by *mount*(1M) and *syncer*(1M). See corresponding manual entries for more information.

In the HP Clustered environment, the *spec* and *dir* arguments should always be fully expanded pathnames.

SEE ALSO

mount(1M), syncer(1M), umount(2).

STANDARDS CONFORMANCE

`mount ( )`: SVID2, XPG2

NAME

mprotect - modify access protections of memory mapping

SYNOPSIS

```
#include <sys/mman.h>

int mprotect(
    caddr_t addr,
    size_t len,
    int prot );
```

DESCRIPTION

`mprotect()` modifies the access protection of the memory mappings specified by the address range starting at *addr* and continuing for *len* bytes, rounded up to the next multiple of the page size, to be that specified by *prot*. If the address range does not correspond to one created by a successful call to `mmap()`, `mprotect()` returns an error. *prot* determines whether read, write, execute, or some combination of accesses are permitted to the data being mapped. Legitimate values for *prot* are the same as those permitted for `mmap()` (see *mmap*(2)).

If the address range being modified corresponds to a mapped file that was mapped with **MAP_SHARED**, `mprotect()` grants write access permission only if the file descriptor used to map the file was opened for writing. If the address range corresponds to a mapped file that was mapped with the **MAP_PRIVATE** or the **MAP_ANONYMOUS** flag, `mprotect()` grants all requested access permissions.

If `mprotect()` fails under a condition other than that specified by EINVAL, the access protection of some of the pages in the range [*addr*, *addr+len*) (from, and including, *addr* to, but not including, *addr+len*) may have been changed. For example, suppose an error occurs on some page at an *addr2*; `mprotect()` may have modified the protections of all whole pages in the range [*addr*,*addr2*].

RETURN VALUE

`mprotect()` returns 0 upon success; otherwise, it returns –1 and sets `errno` to indicate the error.

ERRORS

`mprotect()` fails if any of the following conditions are encountered:

[EACCES]	*prot* specifies a protection that conflicts with the access permission set for the underlying file.
[EINVAL]	*prot* is invalid, or *addr* is not a multiple of the page size as returned by `sysconf(_SC_PAGE_SIZE)`.
[ENOMEM]	The range specified by [*addr*, *addr+len*) (from, and including, *addr* to, but not including, *addr+len*) is invalid for a process' address space, or the range specifies one or more unmapped pages.

AUTHOR

`mprotect()` was developed by HP, AT&T, and OSF.

SEE ALSO

mmap(2), sysconf(2).

STANDARDS CONFORMANCE

`mprotect()`: AES

NAME

msem_init - initialize a semaphore in a mapped file or anonymous memory region

SYNOPSIS

```
#include <sys/mman.h>

msemaphore *msem_init(msemaphore *sem, int initial_value);
```

DESCRIPTION

msem_init() allocates a new binary semaphore and initializes the state of the new semaphore.

sem points to an **msemaphore** structure in which the state of the semaphore is to be stored.

If *initial_value* is **MSEM_LOCKED**, the new semaphore is initialized in the locked state. If *initial_value* is **MSEM_UNLOCKED**, the new semaphore is initialized in the unlocked state.

The **msemaphore** structure must be located within a mapped file or anonymous memory region created by a successful call to **mmap()** and have both read and write access.

If a semaphore is created in a mapped file region, any reference by a process that has mapped the same file, using a **(struct msemaphore *)** pointer that resolves to the same file offset is interpreted as a reference to the same semaphore. If a semaphore is created in an anonymous memory region, any reference by a process sharing the same region by use of a **(struct msemaphore *)** pointer that resolves to the same offset from the start of the region is interpreted as a reference to the same semaphore.

Any previous semaphore state stored in the **msemaphore** structure is be ignored and overwritten.

IMPLEMENTATION NOTES

In order to ensure that an **msemaphore** structure is entirely contained in a single memory page, *sem* must be at an address that is an exact multiple of **sizeof(struct**msemaphore**)**. The size of the **msemaphore** structure is guaranteed to prevent semaphores that cross page boundaries given the above restriction.

For a memory mapped file region, the system deallocates memory that corresponds to a range of the file that has been truncated with **ftruncate()** or **truncate()**. If a semaphore is located in memory so deallocated, the effect is equivalent to an **msem_remove()** on the semaphore.

RETURN VALUE

msem_init() returns the address of the initialized **msemaphore** structure; otherwise, it returns NULL and sets **errno** to indicate the error. NOTE: This error return value may change to –1 in a future HP-UX release. For portability, applications should check for a zero or negative value for error returns.

ERRORS

msem_init() fails if any of the following conditions are encountered:

[EINVAL]	*sem* points to an **msemaphore** structure that is not located in a mapped region created by **mmap()** and with read and write access, or *initial_value* is not valid.
[ENOMEM]	A new semaphore could not be created.
[EFAULT]	*sem* is an invalid pointer.

AUTHOR

msem_init() was developed by HP and OSF.

SEE ALSO

mmap(2), msem_lock(2), msem_remove(2), msem_unlock(2), mman(5).

STANDARDS CONFORMANCE

msem_init(): AES

NAME

msem_lock - lock a semaphore

SYNOPSIS

```
#include <sys/mman.h>

int msem_lock(msemaphore *sem, int condition);
```

DESCRIPTION

`msem_lock()` attempts to lock a binary semaphore.

sem points to an `msemaphore` structure which specifies the semaphore to be locked.

If the semaphore is not currently locked, it becomes locked and the function returns successfully.

If the semaphore is currently locked, and *condition* is `MSEM_IF_NOWAIT`, then the function returns with an error. If the semaphore is currently locked and *condition* is zero, the function does not return until either the calling process is able to successfully lock the semaphore, or an error condition occurs.

All calls to `msem_lock()` and `msem_unlock()` by multiple processes sharing a common `msemaphore` structure behave as if the calls were serialized.

If the `msemaphore` structure contains any value not resulting from a call to `msem_init()` followed by a (possibly empty) sequence of calls to `msem_lock()` and `msem_unlock()`, the results are undefined. The address of an `msemaphore` uniquely identifies the semaphore. If the `msemaphore` structure contains any value copied from an `msemaphore` structure at a different address, the result is undefined.

IMPLEMENTATION NOTES

If blocked on a locked semaphore, `msem_lock()` suspends the calling process at a priority such that the process can be interrupted by a signal.

The system attempts to ignore or recover from invalid values written to the `msemaphore` structure, but this is not guaranteed for all cases.

`msem_lock()` successfully acquires a semaphore that is locked by a process that has exited.

RETURN VALUE

Upon success, `msem_lock()` returns zero; otherwise, it returns –1 and sets `errno` to indicate the error.

ERRORS

`msem_lock()` fails if any of the following conditions are encountered:

[EAGAIN]	`MSEM_IF_NOWAIT` was specified and the semaphore was already locked.
[EINVAL]	*sem* points to an `msemaphore` structure that has been removed, or *condition* is invalid.
[EINTR]	`msem_lock()` was interrupted by a signal that was caught.
[EDEADLK]	The semaphore is currently locked, *condition* is zero, and waiting to lock the semaphore would create a deadlock.
[EFAULT]	*sem* is not a properly aligned address or is otherwise an invalid pointer.

AUTHOR

`msem_lock()` was developed by HP and OSF.

SEE ALSO

msem_init(2), msem_remove(2), msem_unlock(2), mman(5).

STANDARDS CONFORMANCE

`msem_lock()`: AES

NAME

msem_remove - remove a semaphore in mapped file or anonymous region

SYNOPSIS

```
#include <sys/mman.h>

int *msem_remove(msemaphore *sem);
```

DESCRIPTION

msem_remove() removes a binary semaphore.

sem points to an **msemaphore** structure that specifies the semaphore to be removed. Any subsequent use of the **msemaphore** structure before it is again initialized by calling **msem_init()** produces undefined results.

msem_remove() also causes any process waiting in the **msem_lock()** function on the removed semaphore to return with an error.

If the **msemaphore** structure contains any value not resulting from a call to **msem_init()** followed by a (possibly empty) sequence of calls to **msem_lock()** and **msem_unlock()**, the results are undefined. The address of an **msemaphore** uniquely identifies the semaphore. If the **msemaphore** structure contains any value copied from a **msemaphore** structure at a different address, the result is undefined.

RETURN VALUE

Upon success, **msem_remove()** returns zero; otherwise, it returns –1 and sets **errno** to indicate the error.

ERRORS

msem_remove() fails if any of the following conditions are encountered:

[EINVAL] *sem* points to an **msemaphore** structure that has been removed.

[EFAULT] *sem* is an invalid pointer.

AUTHOR

msem_remove() was developed by HP and OSF.

SEE ALSO

msem_init(2), msem_lock(2), msem_remove(2), mman(5).

STANDARDS CONFORMANCE

msem_remove(): AES

NAME

msem_unlock - unlock a semaphore

SYNOPSIS

```
#include <sys/mman.h>

int msem_unlock(msemaphore *sem, int condition);
```

DESCRIPTION

msem_unlock() unlocks a binary semaphore.

sem points to an **msemaphore** structure that specifies the semaphore to be unlocked.

If the *condition* argument is zero, the semaphore will be unlocked, whether or not any other processes are currently attempting to lock it. If the *condition* argument is **MSEM_IF_WAITERS**, and some other process is waiting to lock the semaphore or the implementation cannot reliably determine whether some process is waiting to lock the semaphore, the semaphore is unlocked by the calling process. If the *condition* argument is **MSEM_IF_WAITERS**, and no process is waiting to lock the semaphore, the semaphore is not unlocked and an error is returned.

All calls to **msem_lock()** and **msem_unlock()** by multiple processes sharing a common **msemaphore** structure behave as if the calls were serialized.

If the **msemaphore** structure contains any value not resulting from a call to **msem_init()** followed by a (possibly empty) sequence of calls to **msem_lock()** and **msem_unlock()**, the results are undefined. The address of an **msemaphore** uniquely identifies the semaphore. If the **msemaphore** structure contains any value copied from a **msemaphore** structure at a different address, the result is undefined.

IMPLEMENTATION NOTES

The system attempts to ignore or recover from invalid values placed in the **msemaphore** structure, but this is not guaranteed for all cases.

RETURN VALUE

Upon success, **msem_unlock()** returns zero; otherwise, it returns –1 and sets **errno** to indicate the error.

ERRORS

msem_unlock() fails if any of the following conditions are encountered:

[EAGAIN]	**MSEM_IF_NOWAIT** was specified and there were no waiters.
[EINVAL]	*sem* points to an **msemaphore** structure that has been removed, or *condition* is invalid.
[EFAULT]	*sem* is an invalid pointer.

AUTHOR

msem_unlock() was developed by HP and OSF.

SEE ALSO

msem_init(2), msem_lock(2), msem_remove(2), mman(5).

STANDARDS CONFORMANCE

msem_unlock(): AES

NAME

msgctl - message control operations

SYNOPSIS

```
#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

DESCRIPTION

`msgctl()` provides a variety of message control operations as specified by *cmd*. The following *cmd*s are available:

`IPC_STAT` Place the current value of each member of the data structure associated with *msqid* into the structure pointed to by *buf*. The contents of this structure are defined in *glossary*(9).

`IPC_SET` Set the value of the following members of the data structure associated with *msqid* to the corresponding value found in the structure pointed to by *buf*:

```
msg_perm.uid
msg_perm.gid
msg_perm.mode    /* only low 9 bits */
msg_qbytes
```

This *cmd* can only be executed by a process that has an effective user ID equal to either that of super-user or to the value of either `msg_perm.uid` or `msg_perm.cuid` in the data structure associated with *msqid*. Only super-user can raise the value of `msg_qbytes`.

`IPC_RMID`
Remove the message queue identifier specified by *msqid* from the system and destroy the message queue and data structure associated with it. This *cmd* can only be executed by a process that has an effective user ID equal to either that of super-user or to the value of either `msg_perm.uid` or `msg_perm.cuid` in the data structure associated with *msqid*.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

ERRORS

`msgctl()` fails if one or more of the following is true:

[EINVAL] *msqid* is not a valid message queue identifier.

[EINVAL] *cmd* is not a valid command.

[EACCES] *cmd* is equal to `IPC_STAT` and Read operation permission is denied to the calling process (see *message operation permissions* in *glossary*(9)).

[EPERM] *cmd* is equal to `IPC_RMID` or `IPC_SET` and the effective user ID of the calling process is not equal to that of a user who has appropriate privileges and it is not equal to the value of either `msg_perm.uid` or `msg_perm.cuid` in the data structure associated with *msqid*.

[EPERM] *cmd* is equal to `IPC_SET`, an attempt is being made to increase to the value of `msg_qbytes`, and the effective user ID of the calling process is not equal to that of super-user.

[EFAULT] *buf* points to an illegal address. Reliable detection of this error is implementation dependent.

SEE ALSO

ipcrm(1), ipcs(1), msgget(2), msgop(2), stdipc(3C).

STANDARDS CONFORMANCE

`msgctl()`: SVID2, XPG2, XPG3, XPG4

NAME

msgget - get message queue

SYNOPSIS

```
#include <sys/msg.h>

int msgget(key_t key, int msgflg);
```

DESCRIPTION

msgget() returns the message queue identifier associated with *key*.

A message queue identifier and associated message queue and data structure are created for *key* if one of the following is true:

key is equal to **IPC_PRIVATE**. This call creates a new identifier, subject to available resources. The identifier will never be returned by another call to **msgget()** until it has been released by a call to **msgctl()**. The identifier should be used among the calling process and its descendents; however, it is not a requirement. The resource can be accessed by any process having the proper permissions.

key does not already have a message queue identifier associated with it, and (*msgflg* & **IPC_CREAT**) is "true".

Upon creation, the data structure associated with the new message queue identifier is initialized as follows:

msg_perm.cuid, **msg_perm.uid**, **msg_perm.cgid**, and **msg_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.

The low-order 9 bits of **msg_perm.mode** are set equal to the low-order 9 bits of *msgflg*.

msg_qnum, **msg_lspid**, **msg_lrpid**, **msg_stime**, and **msg_rtime** are set equal to 0.

msg_ctime is set equal to the current time.

msg_qbytes is set equal to the system limit.

RETURN VALUE

Upon successful completion, a non-negative integer, namely a message queue identifier, is returned. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

ERRORS

msgget() fails if one or more of the following is true:

[EACCES] A message queue identifier exists for *key*, but operation permission as specified by the low-order 9 bits of *msgflg* would not be granted.

[ENOENT] A message queue identifier does not exist for *key* and (*msgflg* & **IPC_CREAT**) is "false".

[ENOSPC] A message queue identifier is to be created but the system-imposed limit on the maximum number of allowed message queue identifiers system wide would be exceeded.

[EEXIST] A message queue identifier exists for *key* but ((*msgflg* & **IPC_CREAT**) && (*msgflg* & **IPC_EXCL**)) is "true".

SEE ALSO

ipcrm(1), ipcs(1), msgctl(2), msgop(2), stdipc(3C).

STANDARDS CONFORMANCE

msgget(): SVID2, XPG2, XPG3, XPG4

NAME

msgsnd, msgrcv - message operations

SYNOPSIS

```
#include <sys/msg.h>

int msgsnd(
    int msqid,
    const void *msgp,
    size_t msgsz,
    int msgflg
);

int msgrcv(
    int msqid,
    void *msgp,
    size_t msgsz,
    long msgtyp,
    int msgflg
);
```

DESCRIPTION

msgsnd() is used to send a message to the queue associated with the message queue identifier specified by *msqid*.

msgp points to a user-defined buffer that must contain first a field of type **long** that specifies the type of the message, followed by a data portion that will hold the data bytes of the message. The structure below is an example of what this user-defined buffer might look like:

```
long  mtype;      /* message type */
char  mtext[];    /* message text */
```

mtype is a positive integer that can be used by the receiving process for message selection (see **msgrcv()** below). **mtext** is any text of length **msgsz** bytes. *msgsz* can range from 0 to a system-imposed maximum.

msgflg specifies the action to be taken if one or more of the following is true:

The number of bytes already on the queue is equal to **msg_qbytes** (see *message queue identifier* in *glossary*(9)).

The total number of messages on all queues system-wide is equal to the system-imposed limit.

These actions are as follows:

If (*msgflg* & **IPC_NOWAIT**) is "true", the message is not sent and the calling process returns immediately.

If (*msgflg* & **IPC_NOWAIT**) is "false", the calling process suspends execution until one of the following occurs:

The condition responsible for the suspension no longer exists, in which case the message is sent.

msqid is removed from the system (see *msgctl*(2)). When this occurs, **errno** is set equal to EIDRM, and a value of -1 is returned.

The calling process receives a signal to be caught. In this case the message is not sent and the calling process resumes execution in the manner prescribed in *signal*(5).

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid*:

msg_qnum is incremented by 1.

msg_lspid is set equal to the process ID of the calling process.

msg_stime is set equal to the current time.

msgrcv() reads a message from the queue associated with the message queue identifier specified by *msqid* and places it in the structure pointed to by *msgp*. This structure is composed of the following members:

```
long   mtype;      /* message type */
char   mtext[];    /* message text */
```

mtype is the received message's type as specified by the sending process. **mtext** is the text of the message. *msgsz* specifies the size in bytes of **mtext**. The received message is truncated to *msgsz* bytes if it is larger than *msgsz* and (*msgflg* **&B MSG_NOERROR**) is "true". The truncated part of the message is lost and no indication of the truncation is given to the calling process.

msgtyp specifies the type of message requested as follows:

msgtyp = 0 First message on the queue is received.

msgtyp > 0 First message of type *msgtyp* is received.

msgtyp < 0 First message of the lowest type that is less than or equal to the absolute value of *msgtyp* is received.

msgflg specifies the action to be taken if a message of the desired type is not on the queue. These are as follows:

(*msgflg* **& IPC_NOWAIT**) is "true":
Calling process returns immediately with a return value of -1 and **errno** set to ENOMSG.

(*msgflg* **& IPC_NOWAIT**) is "false":
Calling process suspends execution until one of the following occurs:

- A message of the desired type is placed on the queue.
- *msqid* is removed from the system. When this occurs, **errno** is set equal to EIDRM, and a value of -1 is returned.
- The calling process receives a signal that is to be caught. In this case, a message is not received and the calling process resumes execution in the manner prescribed in *signal*(5)).

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid*.

msg_qnum is decremented by 1.

msg_lrpid is set equal to the process ID of the calling process.

msg_rtime is set equal to the current time.

RETURN VALUES

Upon successful completion, the return value is as follows:

msgsnd() returns a value of 0.

msgrcv() returns a value equal to the number of bytes actually placed into *mtext*.

Otherwise, a value of **-1** is returned and **errno** is set to indicate the error.

ERRORS

msgsnd() fails and no message is sent if one or more of the following is true:

[EINVAL] *msqid* is not a valid message queue identifier.

[EACCES] Operation permission is denied to the calling process.

[EINVAL] *mtype* is less than 1.

[EAGAIN] The message cannot be sent for one of the reasons cited above and (*msgflg* & **IPC_NOWAIT**) is "true".

[EINVAL] *msgsz* is less than zero or greater than the system-imposed limit.

[EFAULT] *msgp* points to an illegal address. The reliable detection of this error is implementation dependent.

[EIDRM] The message queue identifier *msqid* has been removed from the system.

[EINTR] **msgsnd()** was interrupted by a signal.

msgrcv() fails and no message is received if one or more of the following is true:

[EINVAL] *msqid* is not a valid message queue identifier.

[EACCES] Operation permission is denied to the calling process.

[EINVAL] *msgsz* is less than 0.

[E2BIG] **mtext** is greater than *msgsz* and (*msgflg* & **MSG_NOERROR**) is "false".

[ENOMSG] The queue does not contain a message of the desired type and (*msgflg* & **IPC_NOWAIT**) is "true".

[EFAULT] *msgp* points to an illegal address. The reliable detection of this error is implementation dependent.

[EIDRM] The message queue identifier *msqid* has been removed from the system.

[EINTR] The function **msgrcv()** was interrupted by a signal.

WARNINGS

Check all references to *signal*(5) for appropriateness on systems that support *sigvector*(2). *sigvector*(2) can affect the behavior described on this page.

SEE ALSO

ipcs(1), msgctl(2), msgget(2), signal(5), stdipc(3C).

STANDARDS CONFORMANCE

msgrcv(): SVID2, XPG2, XPG3, XPG4

msgsnd(): SVID2, XPG2, XPG3, XPG4

NAME

msync - synchronize a mapped file

SYNOPSIS

```
#include <sys/mman.h>

int msync(caddr_t addr,size_t len, int flags);
```

DESCRIPTION

msync controls the caching operations of a mapped file region. **msync()** writes all modified pages in the region to the file's underlying storage device, and ensures the visibility of modifications made to the region with respect to file system operations.

addr and *len* specify the region to be synchronized. If these are not the address and length of a region created by a previous successful call to **mmap()**, **msync()** returns an error. The behavior of **msync()** upon a region created with the **MAP_ANONYMOUS** or **MAP_PRIVATE** flags is undefined.

flags is constructed from the bitwise inclusive OR of one or more of the following flags defined in **<sys/mman.h>**:

MS_SYNC	Perform synchronous writes
MS_ASYNC	Perform asynchronous writes
MS_INVALIDATE	Invalidate cached pages

If **MS_SYNC** is specified, **msync()** does not return until the system completes all I/O operations. If **MS_ASYNC** is specified, **msync()** returns after the system schedules all I/O operations. Either **MS_SYNC** or **MS_ASYNC** can be set in *flags,* but not both.

If **MS_INVALIDATE** is specified, **msync()** invalidates all cached copies of the pages. Subsequent references to the mapped data is obtained from the file's permanent storage locations. If either **MS_SYNC** or **MS_ASYNC** is also specified, a page is invalidated after it has been written to the file.

After a successful call to **msync()** with **MS_SYNC** specified, all previous modifications to the mapped region are visible to processes using **read()**. Previous modifications to the file using **write()** may be lost.

After a successful call to **msync()** with only **MS_INVALIDATE** specified, all previous modifications to the file using **write()** are visible to the mapped region. Previous direct modifications to the mapped region may be lost.

RETURN VALUE

msync() returns 0 upon success; otherwise, it returns –1 and sets **errno** to indicate the error.

ERRORS

msync() fails if any of the following conditions are encountered.:

[EIO]	An I/O error occurred while reading from or writing to the file system.
[ENOMEM]	The range specified by [*addr*, *addr+len*) (from, and including, *addr* to, but not including, *addr+len*) is invalid for a process' address space, or the range specifies one or more unmapped pages.
[EINVAL]	*addr* is not a multiple of the page size as returned by **sysconf(_SC_PAGE_SIZE)**.
[EINVAL]	The address range specified by *addr* and *len* was not created by a successful call to **mmap()**.

AUTHOR

msync() was developed by HP, AT&T, and OSF.

SEE ALSO

mmap(2), sysconf(2).

STANDARDS CONFORMANCE

msync(): AES

NAME

munmap - unmap a mapped region

SYNOPSIS

```
#include <sys/mman.h>

int munmap(caddr_t addr, size_t len);
```

DESCRIPTION

`munmap()` unmaps a mapped file or anonymous memory region.

`munmap()` unmaps pages in the address range starting at *addr* and continuing for *len* bytes rounded up to the next multiple of the page size. Further references to these pages result in the delivery of a `SIGSEGV` signal to the process.

If the address range specified by *addr* and *len* was not created by a successful call to `mmap()`, `munmap()` returns an error.

If the specified address range was created by multiple calls to `mmap()`, `munmap()` succeeds in unmapping all of the specified regions, provided they form a contiguous address range.

If the region was created with the `MAP_PRIVATE` option, any modifications made to the region are discarded.

RETURN VALUE

`munmap()` returns 0 upon success; otherwise, it returns –1 and sets `errno` to indicate the error.

ERRORS

`munmap()` fails if any of the following conditions are encountered:

[EINVAL] *addr* is not a multiple of the page size as returned by `sysconf(_SC_PAGE_SIZE)`.

[EINVAL] The address range specified by *addr* and *len* was not created by a successful call to `mmap()`.

AUTHOR

`munmap()` was developed by HP, AT&T, and OSF.

SEE ALSO

mmap(2), sysconf(2).

STANDARDS CONFORMANCE

`munmap()`: AES

NAME

nfssvc, async_daemon - NFS daemons

SYNOPSIS

```
int nfssvc(int sock);

void async_daemon();
```

DESCRIPTION

`nfssvc()` starts an NFS daemon listening on socket *sock*. The socket must be AF_INET and SOCK_DGRAM (protocol UDP/IP). The system call returns only if the process is killed.

async_daemon implements the NFS daemon that handles asynchronous I/O for an NFS client. The system call never returns.

ERRORS

`nfssvc()` fails if any of the following conditions is encountered, and sets `errno` accordingly:

[EBADF] *sock* is not a valid socket descriptor.

[EINVAL] *sock* refers to a socket that is not an AF_INET and SOCK_DGRAM socket.

async_daemon fails if the following condition is encountered, and sets `errno` accordingly:

[ENOMEM] There are not enough resources to create the process.

WARNINGS

This call should be used only by HP-supplied commands and is not recommended for use by non-HP-supplied programs.

These two system calls allow kernel processes to have user context.

AUTHOR

`nfssvc()` was developed by Sun Microsystems, Inc.

SEE ALSO

mountd(1M), nfsd(1M).

NAME

nice - change priority of a process

SYNOPSIS

```
#include <unistd.h>

int nice(int priority_change);
```

DESCRIPTION

`nice()` adds the value of *priority_change* to the nice value of the calling process. A process's **nice value** is a positive number for which a more positive value results in lower CPU priority.

A maximum nice value of 39 and a minimum nice value of 0 are imposed by the system. Requests for values above or below these limits result in the nice value being set to the corresponding limit.

RETURN VALUE

Upon successful completion, `nice()` returns the new nice value minus 20. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

Note that `nice()` assumes a user process priority value of 20. If a user having appropriate privileges has changed the user process priority value to something less than 20, certain values for *priority_change* can cause `nice()` to return –1, which is indistinguishable from an error return.

ERRORS

[EPERM] `nice()` fails and does not change the nice value if *priority_change* is negative or greater than 40, and the effective user ID of the calling process is not a user having appropriate privileges.

SEE ALSO

nice(1), renice(1), exec(2).

STANDARDS CONFORMANCE

`nice()`: AES, SVID2, XPG2, XPG3, XPG4

NAME

open - open file for reading or writing

SYNOPSIS

```
#include <fcntl.h>

int open(
    const char *path,
    int oflag, ...
    /* mode_t mode */
);
```

DESCRIPTION

`open()` opens a file descriptor for the named file and sets the file status flags according to the value of *oflag*. *path* points to a path name naming a file, and must not exceed **PATH_MAX** bytes in length. *oflag* values are constructed by OR-ing flags from the list below.

Exactly one of the flags `O_RDONLY`, `O_WRONLY`, or `O_RDWR` must be used in composing the value of *oflag*. If none or more than one is used, the behavior is undefined. Several other flags listed below can be changed by using `fcntl()` while the file is open. See *fcntl*(2) and *fcntl*(5) for details.

`O_RDONLY` Open for reading only.

`O_WRONLY` Open for writing only.

`O_RDWR` Open for reading and writing.

`O_NDELAY` This flag might affect subsequent reads and writes. See *read*(2) and *write*(2).

When opening a FIFO with `O_RDONLY` or `O_WRONLY` set:

If `O_NDELAY` is set:
An `open()` for reading-only returns without delay. An `open()` for writing-only returns an error if no process currently has the file open for reading.

If `O_NDELAY` is clear:
An `open()` for reading-only does not return until a process opens the file for writing. An `open()` for writing-only does not return until a process opens the file for reading.

When opening a file associated with a communication line:

If `O_NDELAY` is set:
The `open()` returns without waiting for carrier.

If `O_NDELAY` is clear:
The `open()` does not return until carrier is present.

`O_NONBLOCK`
Same effect as `O_NDELAY` for *open*(2), but slightly different effect in *read*(2) and *write*(2). Only one of `O_NONBLOCK` and `O_NDELAY` can be specified.

`O_APPEND`
If set, the file offset is set to the end of the file prior to each write.

`O_CREAT`
If the file exists, this flag has no effect, except as noted under `O_EXCL` below. Otherwise, the owner ID of the file is set to the effective user ID of the process, the group ID of the file is set to the effective group ID of the process if the set-group-ID bit of the parent directory is not set, or to the group ID of the parent directory if the set-group-ID bit of the parent directory is set. The file access permission bits of the file mode are set to the value of *mode* modified as follows (see *creat*(2)):

- For each bit set in the file mode creation mask of the process, the corresponding bit in the new file's mode is cleared (see *umask*(2)).
- The "save text image after execution" bit of the mode is cleared. See *chmod*(2).
- On systems with access control lists, three base ACL entries are created corresponding to the file access permissions (see *acl*(5)).

O_TRUNC
If the file exists, its length is truncated to 0 and the mode and owner are unchanged.

O_EXCL
If O_EXCL and O_CREAT are set, open() fails if the file exists.

O_NOCTTY
If set, and *path* identifies a terminal device, open() does not cause the terminal to become the controlling terminal for the process.

O_SYNC
If a file is opened with O_SYNC or if that flag is set with the F_SETFL option of fcntl(), file system writes for the file are done through the cache to the disk as soon as possible, and the process blocks until the data is written to the buffer cache. This flag is ignored by all I/O calls except write(), and is ignored for files other than ordinary files and block special devices on those systems that permit I/O to block special devices.

The name O_SYNCIO is a synonym for O_SYNC, and is defined for backward compatibility in <fcntl.h>.

The file pointer used to mark the current position within the file is set to the beginning of the file.

The new file descriptor is set to remain open across *exec* system calls; see *fcntl*(2).

EXAMPLES

The following call to open() opens file *inputfile* for reading only and returns a file descriptor for *inputfile*. For an example of reading from file *inputfile*, see the *read*(2) manual entry.

```
int myfd;

myfd = open ("inputfile", O_RDONLY);
```

The following call to open() opens file *outputfile* for writing and returns a file descriptor for *outputfile*. For an example of preallocating disk space for *outputfile*, see the *prealloc*(2) manual entry. For an example of writing to *outputfile*, see the *write*(2) manual entry.

```
int outfd;  outfd = open ("outputfile", O_WRONLY);
```

RETURN VALUE

Upon successful completion, the file descriptor is returned. Otherwise, a value of -1 is returned and errno is set to indicate the error.

ERRORS

open() fails and the file is not opened if any of the following conditions are encountered:

[EACCES]	*oflag* permission is denied for the named file.
[EACCES]	A component of the path prefix denies search permission.
[EACCES]	The file does not exist and the directory in which the file is to be created does not permit writing.
[EAGAIN]	One or more segments of a pre-existing file have been locked with *lockf* or *fcntl* by some other process, and O_TRUNC is set.
[EAGAIN]	The file exists, enforcement mode file/record locking is set, and there are outstanding record locks on the file (see *chmod*(2)).
[EDQUOT]	User's disk quota block or inode limit has been reached for this file system.
[EEXIST]	O_CREAT and O_EXCL are set and the named file exists.
[EFAULT]	*path* points outside the allocated address space of the process.
[EINTR]	A signal was caught during the open() system call, and the system call was not restarted (see *signal*(5) and *sigvector*(2)).
[EINVAL]	*oflag* specifies both O_WRONLY and O_RDWR.
[EINVAL]	*oflag* specifies both O_NONBLOCK and O_NDELAY.

[EISDIR] The named file is a directory and *oflag* is write or read/write.

[ELOOP] Too many symbolic links are encountered in translating the path name.

[EMFILE] The maximum number of file descriptors allowed are currently open.

[ENAMETOOLONG]
The length of the specified path name exceeds **PATH_MAX** bytes, or the length of a component of the path name exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect.

[ENFILE] The system file table is full.

[ENOENT] The named file does not exist (for example, *path* is null or a component of *path* does not exist, or the file itself does not exist and **O_CREAT** is not set).

[ENOTDIR] A component of the path prefix is not a directory.

[ENXIO] **O_NDELAY** is set, the named file is a FIFO, **O_WRONLY** is set, and no process has the file open for reading.

[ENODEV] The named file is a character special or block special file, and the device associated with this special file either does not exist, or the driver for this device has not been configured into the kernel.

[EROFS] The named file resides on a read-only file system and *oflag* is write or read/write.

[ETXTBSY] The file is open for execution and *oflag* is write or read/write. Normal executable files are only open for a short time when they start execution. Other executable file types can be kept open for a long time, or indefinitely under some circumstances.

DEPENDENCIES

HP Clustered Environment:

Attempting to open a device file with a *st_rcnode* value that does not match the cnode ID of the machine on which the calling process is running (or 0) fails with an EOPNOTSUPP error.

WARNINGS

Access Control Lists

Access control list descriptions in this entry apply only to standard HP-UX operating systems. If HP-UX BLS software has been installed, access control lists are handled differently. Refer to HP-UX BLS documentation for information about access control lists in the HP-UX BLS environment.

AUTHOR

open() was developed by HP, AT&T, and the University of California, Berkeley.

SEE ALSO

chmod(2), close(2), creat(2), dup(2), fcntl(2), lockf(2), lseek(2), read(2), select(2), setacl(2), umask(2), write(2), acl(5), fcntl(5), signal(5).

STANDARDS CONFORMANCE

open(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

pathconf(), fpathconf() - get configurable pathname variables

SYNOPSIS

```
#include <unistd.h>

long pathconf(const char *path, int name);

long fpathconf(int fildes, int name);
```

DESCRIPTION

`pathconf()` and `fpathconf()` provide a method for applications to determine the value of a configurable limit or option associated with a file or directory (see *limits*(5) and `<unistd.h>`).

For `pathconf()`, the *path* argument points to the path name of a file or directory.

For `fpathconf()`, the *fildes* argument is an open file descriptor.

For both functions, the *name* argument represents the variable to be queried regarding the file or directory to which the other argument refers.

The following table lists the configuration variables available from `pathconf()` and `fpathconf()`, and lists for each variable the associated value of the *name* argument:

Variable	Value of *name*	Notes
LINK_MAX	_PC_LINK_MAX	1
MAX_CANON	_PC_MAX_CANON	2
MAX_INPUT	_PC_MAX_INPUT	2
NAME_MAX	_PC_NAME_MAX	3, 4
PATH_MAX	_PC_PATH_MAX	4, 5
PIPE_BUF	_PC_PIPE_BUF	6
_POSIX_CHOWN_RESTRICTED	_PC_CHOWN_RESTRICTED	7, 8
_POSIX_NO_TRUNC	_PC_NO_TRUNC	3, 4
_POSIX_VDISABLE	_PC_V_DISABLE	2

The variables in the table are defined as constants in `<limits.h>` or `<unistd.h>` if they do not vary from one pathname to another. The associated values of the *name* argument are defined in `<unistd.h>`.

RETURN VALUE

The following notes further qualify the table above.

1. If *path* or *fildes* refers to a directory, the value returned applies to the directory itself.
2. If the variable is constant, the value returned is identical to the variable's definition in `<limits.h>` or `<unistd.h>` regardless of the type of *fildes* or *path*. The behavior is undefined if *path* or *fildes* does not refer to a terminal file.
3. If *path* or *fildes* refers to a directory, the value returned applies to the filenames within the directory.
4. If *path* or *fildes* does not refer to a directory, `pathconf()` or `fpathconf()` returns -1 and sets `errno` to EINVAL.
5. If *path* or *fildes* refers to a directory, the value returned is the maximum length of a relative path name when the specified directory is the working directory.
6. If *path* refers to a FIFO, or *fildes* refers to a pipe or FIFO, the value returned applies to the pipe or FIFO itself. If *path* or *fildes* refers to a directory, the value returned applies to any FIFOs that exist or can be created within the directory. If `PIPE_BUF` is a constant, the value returned is identical to the definition of `PIPE_BUF` in `<limits.h>` regardless of the type of *fildes* or *path*. The behavior is undefined for a file other than a directory, FIFO, or pipe.
7. If *path* or *fildes* refers to a directory, the value returned applies to files of any type, other than directories, that exist or can be created within the directory.
8. `_POSIX_CHOWN_RESTRICTED` is defined if the privilege group `PRIV_GLOBAL` has been granted the `CHOWN` privilege (see *getprivgrp*(2) and *chown*(2)). In all other cases, `_POSIX_CHOWN_RESTRICTED` is undefined and *pathconf* or *fpathconf* returns -1 without changing `errno`. To determine if *chown* can be performed on a file, it is simplest to attempt the

`chown( )` operation and check the return value for failure or success.

If the variable corresponding to *name* is not defined for *path* or *fildes*, the *pathconf* and *fpathconf* functions succeed and return a value of `-1`, without changing the value of `errno`.

Upon any other successful completion, these functions return the value of the named variable with respect to the specified file or directory, as described above.

Otherwise, a value of `-1` is returned and `errno` is set to indicate the error.

ERRORS

pathconf and *fpathconf* fail if any of the following conditions are encountered:

[EACCES]	A component of the path prefix denies search permission.
[EBADF]	The *fildes* argument is not a valid open file descriptor.
[EFAULT]	*path* points outside the allocated address space of the process.
[EINVAL]	The value of *name* is not valid or the implementation does not support an association of the variable *name* with the specified file.
[ELOOP]	Too many symbolic links were encountered in translating *path*.
[ENAMETOOLONG]	The length of the specified path name exceeds **PATH_MAX** bytes, or the length of a component of the path name exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect.
[ENOENT]	The file named by *path* does not exist (for example, *path* is null or a component of *path* does not exist).
[ENOTDIR]	A component of the path prefix is not a directory.

EXAMPLES

The following example sets *val* to the value of **MAX_CANON** for the device file being used as the standard input. If the standard input is a terminal, this value is the maximum number of input characters that can be entered on a single input line before typing the newline character:

```
if (isatty(0))
    val = fpathconf(0, _PC_MAX_CANON);
```

The following code segment shows two calls to *pathconf*, one to determine whether a file name longer than **NAME_MAX** bytes will be truncated to **NAME_MAX** bytes in the `/tmp` directory, and if so, another call to determine the actual value of **NAME_MAX** so that an error can be printed if a user-supplied file name stored in *filebuf* will be truncated in this directory:

```
extern int errno;
char *filebuf;
   errno = 0;  /* reset errno */
   if ( pathconf("/tmp" _PC_NO_TRUNC) == -1 ) {
            /* _POSIX_NO_TRUNC is not in effect for this directory */
            if (strlen(filebuf) > pathconf("/tmp", PC_NAME_MAX)) {
                     fprintf(stderr, "Filename %s too long.\n", filebuf);
                     /* take error action */
            }
            else
                     if (errno) {
                              perror("pathconf");
                              /* take error action */
                     }
            }
/* otherwise, _POSIX_NO_TRUNC is in effect for this directory */
   if ((fd = open(filebuf, O_CREAT, mode)) < 0)
            perror(filebuf);
```

DEPENDENCIES

NFS

The following error can occur:

[EOPNOTSUPP] *path* or *fildes* refers to a file for which a value for *name* cannot be determined. In particular, `_PC_LINK_MAX`, `_PC_NAME_MAX`, `_PC_PATH_MAX`, `_PC_NO_TRUNC`, and `_PC_CHOWN_RESTRICTED`, cannot be determined for an NFS file.

AUTHOR

`pathconf()` and `fpathconf()` were developed by HP.

SEE ALSO

errno(2), chown(2), limits(5), unistd(5), termio(7).

STANDARDS CONFORMANCE

`pathconf()`: AES, XPG3, XPG4, FIPS 151-2, POSIX.1, POSIX.2

`fpathconf()`: AES, XPG3, XPG4, FIPS 151-2, POSIX.1, POSIX.2

NAME

pause - suspend process until signal

SYNOPSIS

```
#include <unistd.h>

int pause(void);
```

DESCRIPTION

`pause()` suspends the calling process until it receives a signal. The signal must be one that is not currently set to be ignored or blocked (masked) by the calling process.

If the signal causes termination of the calling process, `pause()` does not return.

If the signal is *caught* by the calling process and control is returned from the signal-catching function (see *signal*(5)), the calling process resumes execution from the point of suspension; with a return value of -1 from `pause()` and `errno` set to EINTR.

WARNING

Check all references to *signal*(5) for appropriateness on systems that support *sigvector*(2). `sigvector()` can affect the behavior described on this page.

SEE ALSO

alarm(2), kill(2), sigvector(2), wait(2), signal(5).

STANDARDS CONFORMANCE

`pause()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

pipe - create an interprocess channel

SYNOPSIS

```
int pipe(int fildes[2]);
```

DESCRIPTION

`pipe()` creates an I/O mechanism called a pipe and returns two file descriptors, *fildes*[0] and *fildes*[1]. *fildes*[0] is opened for reading and *fildes*[1] is opened for writing.

A read-only file descriptor *fildes*[0] accesses the data written to *fildes*[1] on a first-in-first-out (FIFO) basis. For details of the I/O behavior of pipes see *read*(2) and *write*(2).

EXAMPLES

The following example uses `pipe()` to implement the command string `ls | sort`:

```
#include <sys/types.h>
pid_t pid;
int pipefd[2];

/* Assumes file descriptor 0 and 1 are open */
pipe (pipefd);

if ((pid = fork()) == (pid_t)0) {
     close(1);  /* close stdout */
     dup (pipefd[1]);
     close (pipefd[0]);
     execlp ("ls", "ls", (char *)0);
}
else if (pid > (pid_t)0) {
     close(0);  /* close stdin  */
     dup (pipefd[0]);
     close (pipefd[1]);
     execlp ("sort", "sort", (char *)0);
}
```

RETURN VALUE

Upon successful completion, a value of `0` is returned. Otherwise, a value of `-1` is returned and `errno` is set to indicate the error.

ERRORS

`pipe()` fails if one or more of the following is true:

[EMFILE]	NFILE - 1 or more file descriptors are currently open.
[ENFILE]	The system file table is full.
[ENOSPC]	The file system lacks sufficient space to create the pipe.

SEE ALSO

sh(1), read(2), write(2), popen(3S).

STANDARDS CONFORMANCE

`pipe()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

plock - lock process, text, or data in memory

SYNOPSIS

```
#include <sys/lock.h>

int plock(int op);
```

DESCRIPTION

`plock()` allows the calling process to lock the text segment of the process (text lock), its data segment (data lock), or both its text and data segment (process lock) into memory. Locked segments are immune to all routine swapping. `plock()` also allows these segments to be unlocked. To use this call, the calling process must be a member of a privilege group allowing `plock()` (see `setprivgrp()` on *getprivgrp*(2)) or the effective user ID of the calling process must be a user having appropriate privileges. *op* specifies the following:

PROCLOCK	lock text and data segments into memory (process lock)
TXTLOCK	lock text segment into memory (text lock)
DATLOCK	lock data segment into memory (data lock)
UNLOCK	remove locks

EXAMPLES

The following call to `plock()` locks the calling process in memory:

```
plock (PROCLOCK);
```

RETURN VALUE

Upon successful completion, `plock()` returns 0 to the calling process. Otherwise, it returns -1 and sets `errno` to indicate the error.

ERRORS

`plock()` fails and does not perform the requested operation if any of the following conditions are encountered:

[EPERM]	The effective user ID of the calling process is not super-user and the user does not have the **PRIV_MLOCK** privilege.
[EINVAL]	*op* is equal to **PROCLOCK** and a process lock, a text lock, or a data lock already exists on the calling process.
[EINVAL]	*op* is equal to **TXTLOCK** and a text lock or process lock already exists on the calling process.
[EINVAL]	*op* is equal to **DATLOCK** and a data lock, or process lock already exists on the calling process.
[EINVAL]	*op* is equal to **UNLOCK** and no type of lock exists on the calling process.
[EINVAL]	*op* is not equal to either **PROCLOCK**, **TXTLOCK**, **DATLOCK**, or **UNLOCK**.
[EINVAL]	`plock()` not allowed in [vfork, exec] window (see *vfork*(2)).
[ENOMEM]	There is not sufficient lockable memory in the system to satisfy the locking request.

SEE ALSO

exec(2), exit(2), fork(2).

STANDARDS CONFORMANCE

`plock()`: SVID2, XPG2

NAME

poll - monitor I/O conditions on multiple file descriptors

SYNOPSIS

```
#include <poll.h>

int poll(
    struct pollfd fds[],
    int nfds,
    int timeout
);
```

DESCRIPTION

`poll()` provides a general mechanism for reporting I/O conditions associated with a set of file descriptors and for waiting until one or more specified conditions becomes true. Specified conditions include the ability to read or write data without blocking, and error conditions.

Arguments

fds	Points to an array of `pollfd` structures, one for each file descriptor of interest.
nfds	Specifies the number of `pollfd` structures in the *fds* array.
timeout	Specifies the maximum length of time (in milliseconds) to wait for at least one of the specified conditions to occur.

Each `pollfd` structure includes the following members:

`int fd`	File descriptor
`short events`	Requested conditions
`short revents`	Reported conditions

The `fd` member of each `pollfd` structure specifies an open file descriptor. The `poll()` function uses the `events` member to determine what conditions to report for this file descriptor. If one or more of these conditions is true, `poll()` sets the associated `revents` member.

`poll()` ignores any `pollfd` structure whose `fd` member is negative. If the `fd` member of all `pollfd` structures is negative, `poll()` returns 0 and has no other results.

The `events` and `revents` members of the `pollfd` structure are bit masks. The calling process sets the `events` bit mask, and `poll()` sets the `revents` bit masks. These bit masks contain ORed combinations of condition flags. The following condition flags are defined:

`POLLIN`	Data can be read without blocking. For streams, this flag means that a message that is not high priority is at the front of the stream head read queue. This message can be of zero length.
`POLLNORM`	Synonym for `POLLIN`
`POLLPRI`	A high priority message is available. For streams, this message can be of zero length.
`POLLOUT`	Data can be written without blocking. For streams, this flag specifies that normal data (not high priority or priority band > 0) can be written without being blocked by flow control. This flag is not used for high priority data, because it can be written even if the stream is flow controlled.
`POLLERR`	An error has occurred on the file descriptor.
`POLLHUP`	The device has been disconnected. For streams, this flag in `revents` is mutually exclusive with `POLLOUT`, since a stream cannot be written to after a hangup occurs. This flag and `POLLIN`, `POLLPRI`, `POLLRDNORM`, `POLLRDBAND`, and `POLLMSG` are not mutually exclusive.
`POLLNVAL`	`fd` is not a valid file descriptor.
`POLLRDNORM`	A non-priority message is available. For streams, this flag means that a normal message (not high priority or priority band > 0) is at the front of the stream head read queue. This message can be of zero length.
`POLLRDBAND`	A priority message (priority band > 0) is at the front of the stream head read queue. This message can be read without blocking. The message can be of zero length.

POLLWRNORM	Same as POLLOUT
POLLWRBAND	Priority data (priority band > 0) can be written without being blocked by flow control. Only previously written bands are checked.
POLLMSG	A M_SIG or M_PCSIG message specifying SIGPOLL has reached the front of the stream head read queue.

The conditions indicated by POLLNORM and POLLOUT are true if and only if at least one byte of data can be read or written without blocking. The exception is regular files, which always poll true for POLLNORM and POLLOUT. Also, streams return POLLNORM in **revents** even if the available message is of zero length.

The condition flags POLLERR, POLLHUP, and POLLNVAL are always set in **revents** if the conditions they indicate are true for the specified file descriptor, whether or not these flags are set in **events**.

For each call to **poll()**, the set of reportable conditions for each file descriptor consists of those conditions that are always reported, together with any further conditions for which flags are set in **events**. If any reportable condition is true for any file descriptor, **poll()** returns with flags set in **revents** for each true condition for that file descriptor.

If no reportable condition is true for any of the file descriptors, **poll()** waits up to *timeout* milliseconds for a reportable condition to become true. If, in that time interval, a reportable condition becomes true for any of the file descriptors, **poll()** reports the condition in the file descriptor's associated **revents** member and returns. If no reportable condition becomes true, **poll()** returns without setting any **revents** bit masks.

If the *timeout* parameter is a value of –1, **poll()** does not return until at least one specified event has occurred. If the value of the *timeout* parameter is 0, **poll()** does not wait for an event to occur but returns immediately, even if no specified event has occurred. The behavior of **poll()** is not affected by whether the O_NONBLOCK flag is set on any of the specified file descriptors.

RETURN VALUES

Upon successful completion, **poll()** returns a nonnegative value. If the call returns 0, **poll()** has timed out and has not set any of the **revents** bit masks. A positive value indicates the number of file descriptors for which **poll()** has set the **revents** bit mask. If **poll()** fails, it returns –1 and sets **errno** to indicate the error.

ERRORS

poll() fails if any of the following conditions are encountered:

[EAGAIN].	Allocation of internal data structures failed. A later call to **poll()** may complete successfully.
[EINTR]	A signal was delivered before any of the selected for conditions occurred or before the time limit expired.
[EINVAL]	*timeout* is a negative number other than –1, or *nfds* is negative.
[EFAULT]	The *fds* parameter in conjunction with the *nfds* parameter addresses a location outside of the allocated address space of the process. Reliable detection of this error is implementation-dependent.

EXAMPLES

Wait for input on file descriptor 0:

```
#include <poll.h>
struct pollfd fds;

fds.fd = 0;
fds.events = POLLNORM;
poll(&fds, 1, -1);
```

Wait for input on **ifd1** and **ifd2**, output on **ofd**, giving up after 10 seconds:

```
#include <poll.h>
struct pollfd fds[3];
int ifd1, ifd2, ofd, count;

fds[0].fd = ifd1;
```

```
fds[0].events = POLLNORM;
fds[1].fd = ifd2;
fds[1].events = POLLNORM;
fds[2].fd = ofd;
fds[2].events = POLLOUT;
count = poll(fds, 3, 10000);
if (count == -1) {
        perror("poll failed");
        exit(1);
}
if (count==0)
        printf("No data for reading or writing\n");
if (fds[0].revents & POLLNORM)
        printf("There is data for reading fd %d\n", fds[0].fd);
if (fds[1].revents & POLLNORM)
        printf("There is data for reading fd %d\n", fds[1].fd);
if (fds[2].revents & POLLOUT)
        printf("There is room to write on fd %d\n", fds[2].fd);
```

Check for input or output on file descriptor 5 without waiting:

```
#include <poll.h>
struct pollfd fds;

fds.fd = 5;
fds.events = POLLNORM|POLLOUT;
poll(&fds, 1, 0);
if (fds.revents & POLLNORM)
        printf("There is data available on fd %d\n", fds.fd);
if (fds.revents & POLLOUT)
        printf("There is room to write on fd %d\n", fds.fd);
```

Wait 3.5 seconds:

```
#include <stdio.h>
#include <poll.h>

poll((struct pollfd *) NULL, 0, 3500);
```

Wait for a high priority, priority, or normal message on streams file descriptor 0:

```
#include <poll.h>
struct pollfd fds;

fds.fd = 0;
fds.events = POLLIN|POLLPRI;
poll(&fds, 1, -1);
```

WARNINGS

In some countries, electioneering is illegal within one hundred feet of a polling place.

SEE ALSO

read(2), write(2), select(2), getmsg(2), putmsg(2), streamio(7).

STANDARDS CONFORMANCE

`poll()`: AES, SVID2

NAME

prealloc - preallocate fast disk storage

SYNOPSIS

```
#include <unistd.h>

int prealloc(int fildes, off_t size);
```

DESCRIPTION

`prealloc()` is used to preallocate space on a disk for faster storage operations.

fildes is a file descriptor obtained from a `creat()`, `pen()`, `dup()`, or `fcntl()` system call for an ordinary file of zero length. It must be opened writable, because it will be written to by `prealloc()`. *size* is the size in bytes to be preallocated for the file specified by *fildes*. At least *size* bytes will be allocated. Space is allocated in an implementation-dependent fashion for fast sequential reads and writes. The EOF in an extended file is left at the end of the preallocated area. The current file pointer is left at zero. The file is zero-filled.

Using `prealloc()` on a file does *not* give the file an attribute that is inherited when copying or restoring the file using a program such as `cp` or `tar` (see *cp*(1) and *tar*(1)). It simply ensures that disk space has been preallocated for *size* bytes in a manner suited for sequential access. The file can be extended beyond these limits by `write()` operations past the original end of file. However, this space will not necessarily be allocated using any special strategy.

EXAMPLES

Assuming a process has opened a file for writing, the following call to `prealloc()` preallocates at least 50 000 bytes on disk for the file represented by file descriptor *outfd*:

```
prealloc (outfd, 50000);
```

DEPENDENCIES

Since the exact effect and performance benefits obtainable by using this call vary with the implementation of the file system, performance related details are described in the system administrator manuals for each specific machine.

RETURN VALUE

Upon successful completion, `prealloc()` returns 0; otherwise, it returns -1 and sets `errno` to indicate the error.

ERRORS

`prealloc()` fails and no disk space is allocated if any of the following conditions are encountered:

[EBADF]	*fildes* is not a valid open file descriptor opened for writing.
[EDQUOT]	User's disk quota block limit has been reached for this file system.
[EFBIG]	*size* exceeds the maximum file size or the process's file size limit. See *ulimit*(2).
[ENOSPC]	Not enough space is left on the device to allocate the requested amount; no space was allocated.
[ENOTEMPTY]	*fildes* not associated with an ordinary file of zero length.

AUTHOR

`prealloc()` was developed by HP.

SEE ALSO

prealloc(1), creat(2), dup(2), fcntl(2), open(2), read(2), ulimit(2), write(2).

WARNINGS

Allocation of the file space is highly dependent on current disk usage. A successful return does not tell you how fragmented the file actually might be if the disk is nearing its capacity.

NAME

profil - execution time profile

SYNOPSIS

```
#include <time.h>

void profil(
    unsigned short int *buff,
    size_t bufsiz,
    size_t offset,
    unsigned int scale
);
```

DESCRIPTION

`profil()` controls profiling, by which the system maintains estimates of the amount of time the calling program spends executing at various places in its address space.

The *buff* argument must point to an area of memory whose length (in bytes) is given by *bufsiz*. When profiling is on, the process's program counter (pc) is examined each clock tick (`CLK_TCK` times per second), *offset* is subtracted from the pc value, and the result is multiplied by *scale*. If the resulting number corresponds to an element inside the array of `unsigned short ints` to which *buff* points, that element is incremented.

The number of samples per second for a given implementation is given by `CLK_TCK`, which is defined in `<time.h>`.

The scale is interpreted as an unsigned, sixteen bit, fixed-point fraction with binary point at the left: 0177777 (octal) gives a one-to-one mapping of pc's to words in *buff*; 077777 (octal) maps each pair of instruction words together. 02(octal) maps all instructions onto the beginning of *buff* (producing a non-interrupting core clock).

Profiling is turned off by giving a *scale* of 0 or 1. It is rendered ineffective by giving a *bufsiz* of 0. Profiling is turned off when one of the `exec()` functions is executed, but remains on in child and parent both after a `fork()`. Profiling is turned off if an update in *buff* would cause a memory fault.

RETURN VALUE

No value is returned.

SEE ALSO

prof(1), monitor(3C).

STANDARDS CONFORMANCE

`profil()`: SVID2, XPG2

NAME

ptrace - process trace

SYNOPSIS

```
#include <sys/ptrace.h>

int ptrace(
    int request,
    pid_t pid,
    int addr,
    int data,
    int addr2
);
```

REMARKS

Much of the functionality of this capability is highly dependent on the underlying hardware. An application that uses this system call should not be expected to be portable across architectures or implementations.

DESCRIPTION

`ptrace()` provides a means by which a process can control the execution of another process. Its primary use is for the implementation of breakpoint debugging; see *adb*(1). The traced process behaves normally until it encounters a signal (see *signal*(2) for the list), at which time it enters a stopped state and the tracing process is notified via `wait()` (see *wait*(2)). When the traced process is in the stopped state, the tracing process can examine and modify the "core image" using `ptrace()`. Also, the tracing process can cause the traced process either to terminate or continue, with the possibility of ignoring the signal that caused it to stop.

The *request* argument determines the precise action to be taken by `ptrace()` and is one of the following:

`PT_SETTRC` This request must be issued by a child process if it is to be traced by its parent. It turns on the child's trace flag which stipulates that the child should be left in a stopped state upon receipt of a signal rather than the state specified by *func*; see *signal*(2). The *pid*, *addr*, *data*, and *addr2* arguments are ignored, and a return value is not defined for this request. Peculiar results occur if the parent does not expect to trace the child.

The remainder of the requests can only be used by the tracing process. For each, *pid* is the process ID of the process being traced, which must be in a stopped state before these requests are made.

`PT_RIUSER, PT_RDUSER` With these requests, the word at location *addr* in the address space of the traced process is returned to the tracing process. If instruction (I) and data (D) space are separated, request `PT_RIUSER` returns a word from I space, and request `PT_RDUSER` returns a word from D space. If I and D space are not separated, either request `PT_RIUSER` or request `PT_RDUSER` can be used with equivalent results. The *data* and *addr2* arguments are ignored. These two requests fail if *addr* is not the start address of a word, in which case a value of -1 is returned to the tracing process and its `errno` is set to EIO.

`PT_RUAREA` With this request, the word at location *addr* in the USER area of the traced process in the system's address space (see `<sys/user.h>`) is returned to the tracing process. Addresses in this area are system dependent, but start at zero. The limit can be derived from `<sys/user.h>`. The *data* and *addr2* arguments are ignored. This request fails if *addr* is not the start address of a word or is outside the USER area, in which case a value of -1 is returned to the tracing process and its `errno` is set to EIO.

`PT_WIUSER, PT_WDUSER` With these requests, the value given by the *data* argument is written into the address space of the traced process at location *addr*. Request `PT_WIUSER` writes a word into I space, and request `PT_WDUSER` writes a word in D space. Upon successful completion, the value written into the address space of the traced process is returned to the tracing process. The *addr2* argument is ignored. These two requests fail if *addr* is not the start address of a word, or if *addr* is a location in a

pure procedure space and either another process is executing in that space or the tracing process does not have write access for the executable file corresponding to that space. Upon failure a value of -1 is returned to the tracing process and its `errno` is set to EIO.

PT_WUAREA With this request, a few entries in the traced process' USER area can be written. *data* gives the value that is to be written and *addr* is the location of the entry. The *addr2* argument is ignored. The few entries that can be written are dependent on the architecture of the system, but include the user data registers, auxiliary data registers, and status register (the set of registers, or bits in registers, that the user's program could modify).

PT_CONTIN This request causes the traced process to resume execution. If the *data* argument is 0, all pending signals, including the one that caused the traced process to stop, are canceled before it resumes execution. If the *data* argument is a valid signal number, the traced process resumes execution as if it had incurred that signal, and any other pending signals are canceled. The *addr* argument must be equal to 1 for this request. The *addr2* argument is ignored. Upon successful completion, the value of *data* is returned to the tracing process. This request fails if *data* is not 0 or a valid signal number, in which case a value of -1 is returned to the tracing process and its `errno` is set to EIO.

PT_EXIT This request causes the traced process to terminate with the same consequences as `exit()`. The *addr*, *data*, and *addr2* arguments are ignored.

PT_SINGLE This request causes a flag to be set so that an interrupt occurs uponthe completion of one machine instruction, and then executes the same steps as listed above for request **PT_CONTIN**. If the processor does not provide a trace bit, this request returns an error. This effectively allows single stepping of the traced process.

Whether or not the trace bit remains set after this interrupt is a function of the hardware.

PT_ATTACH This request stops the process identified by *pid* and allows the calling process to trace it. Process *pid* does not have to be a child of the calling process, but the effective user ID of the calling process must match the real and saved *uid* of process *pid* unless the effective user ID of the tracing process is super-user. The calling process can use the `wait()` system call to wait for process *pid* to stop. The *addr*, *data*, and *addr2* arguments are ignored.

PT_DETACH This request detaches the traced process *pid* and allows it to continue its execution in the manner of **PT_CONTIN**.

To forestall possible fraud, `ptrace()` inhibits the set-user-ID facility on subsequent `exec()` calls. If a traced process calls `exec()`, it stops before executing the first instruction of the new image showing signal **SIGTRAP**.

ERRORS

In general, `ptrace()` fails if any of the following conditions are encountered:

[EIO]	*request* is an illegal number.
[EPERM]	The specified process cannot be attached for tracing.
[ESRCH]	*pid* identifies a process to be traced that does not exist or has not executed a `ptrace()` with request **PT_SETTRC**.

DEPENDENCIES

Series 300/400

The following additional requests are available:

PT_RFPREGS With this request, the child's floating-point accelerator register set is returned to the parent process in *addr*. *addr* must be the address of a buffer of at least 136 bytes. The first 128 bytes contains the 16 double-precision floating-point registers and the next 8 bytes contains the status and control registers. The data argument is ignored. This request fails if the child process is not using the floating-point accelerator, in

which case a value of -1 is returned to the parent process and the parent's `errno` is set to EIO. This request also fails if *addr* is a bad address, in which case a value of -1 is returned to the parent process and the parent's `errno` is set to EFAULT.

`PT_WFPREGS` With this request, the child's floating-point accelerator register set is written from the buffer pointed to by *addr*. *addr* must be the address of a buffer of at least 136 bytes. The first 128 bytes contains the new values for the 16 double-precision floating point registers and the next 8 bytes contains the new values for the status and control registers. The data argument is ignored. This request fails if the child process is not using the floating-point accelerator, in which case a value of -1 is returned to the parent process and the parent's `errno` is set to EIO. This request also fails if *addr* is a bad address, in which case a value of -1 is returned to the parent process and the parent's `errno` is set to EFAULT.

Series 700/800

The request `PT_WUAREA` is not supported. Therefore, it returns -1, sets `errno` to EIO and does not affect the USER area of the traced process.

If the *addr* argument to a `PT_CONTIN` or `PT_SINGLE` request is not 1, the Instruction Address Offset Queue (program counter) is loaded with the values *addr* and *addr*+4 before execution resumes. Otherwise, execution resumes from the point where it was interrupted.

If the *addr* argument to a `PT_DETACH` request is not 1, the Instruction Address Offset Queue is loaded with the values *addr* and *addr2*.

Additional requests are available:

`PT_RUREGS` With this request, the word at location *addr* in the `save_state` structure at the base of the per-process kernel stack is returned to the tracing process. *addr* must be word-aligned and less than `STACKSIZE*NBPG` (see <`sys/param.h`> and <`machine/param.h`>). The `save_state` structure contains the registers and other information about the process. The *data* and *addr2* arguments are ignored.

`PT_WUREGS` The `save_state` structure at the base of the per-process kernel stack is written as it is read with request `PT_RUREGS`. Only a few locations can be written in this way: the general registers, most floating-point registers, a few control registers, and certain bits of the interruption processor status word. The *addr2* argument is ignored.

`PT_RDTEXT, PT_RDDATA`
These requests are identical to `PT_RIUSER` and `PT_RDUSER`, except that the *data* argument specifies the number of bytes to read and the *addr2* argument specifies where to store that data in the tracing process.

`PT_WRTEXT , PT_WRDATA`
These requests are identical to `PT_WIUSER` and `PT_WDUSER` except that the *data* argument specifies the number of bytes to write and the *addr2* argument specifies where to read that data in the tracing process.

SEE ALSO

adb(1), exec(2), signal(2), wait(2).

STANDARDS CONFORMANCE

`ptrace()`: SVID2, XPG2

NAME

quotactl - manipulate disk quotas

SYNOPSIS

```
#include <sys/quota.h>

int quotactl(int cmd, const char *special, uid_t uid, void *addr);
```

DESCRIPTION

`quotactl()` manipulates disk quotas. *cmd* indicates a command to be applied to the user ID *uid*. Parameter *special* is a pointer to a null-terminated string containing the path name of the block special device for the file system being manipulated. The block special device must be mounted as an hfs file system (see `mount`(2)). The parameter *addr* is the address of an optional, command-specific, data structure which is copied in or out of the system. The interpretation of *addr* is explained with each command below:

Q_QUOTAON Turn on quotas for a file system. The parameter *addr* points to the path name of file containing the quotas for the file system. The quota file must exist; it is normally created using the `quotacheck` command (see *quotacheck*(1M)). The *uid* parameter is ignored. This call is restricted to users having appropriate privileges.

Q_QUOTAOFF Turn off quotas for a file system. The *addr* and *uid* parameters are ignored. This call is restricted to the user with appropriate privileges.

Q_GETQUOTA Get disk quota limits and current usage for user *uid*. *addr* is a pointer to a `dqblk` structure (defined in `<sys/quota.h>`). Only users having appropriate privileges can get the quotas of a user other than himself.

Q_SETQUOTA Set disk quota limits and current usage of files and blocks for user *uid*. *addr* is a pointer to a `dqblk` structure (defined in `<sys/quota.h>`). This call is restricted to users with appropriate privileges.

Q_SETQLIM Set disk quota limits for user *uid*. The parameter *addr* is a pointer to a `dqblk` structure (defined in `<sys/quota.h>`). This call is restricted to users with appropriate privileges.

Q_SYNC Update the on-disk copy of quota usages for a file system. If *special* is null, all file systems with active quotas are synced. The parameters *addr* and *uid* are ignored.

RETURN VALUE

Upon successful completion, `quotactl()` returns 0; otherwise, it returns -1 and sets `errno` to indicate the error.

ERRORS

`quotactl()` fails when any of the following occurs:

[ENOSYS] The kernel has not been configured with the disk quota subsystem.

[EINVAL] The parameter *cmd* is invalid.

[ESRCH] No disc quota is found for the indicated user or quotas have not been turned on for this file system.

[EPERM] The call is privileged and the calling process does not have appropriate privileges.

[ENODEV] The parameter *special* is not a mounted HFS file system.

[ENOTBLK] The parameter *special* is not a block device.

[EACCES] (**Q_QUOTAON**) The quota file pointed to by *addr* exists but is either not a regular file or is not on the file system pointed to by *special*.

[EBUSY] **Q_QUOTAON** attempted while another **Q_QUOTAON** or **Q_QUOTAOFF** is in progress.

[ENOENT] The file specified by *special* or *addr* does not exist.

[EFAULT] The *addr* or *special* parameter points to an invalid address. Reliable detection of this error is implementation-dependent.

[EDQUOT] User's disk quota block limit has been reached for this file system.

WARNINGS

The `quotactl()` system call is incompatible with the 4.2/4.3BSD implementation of Melbourne quotas which uses a different system call interface and on-disk data structure.

AUTHOR

`quotactl()` was developed by HP and Sun Microsystems, Inc.

SEE ALSO

quota (1), edquota (1M), rquotad (1M), quotacheck (1M), quotaon (1M), mount (2), quota (5), privilege(5).

NAME

read, readv - read input

SYNOPSIS

```
#include <unistd.h>

size_t read(int fildes, void *buf, size_t nbyte);

#include <sys/uio.h>

ssize_t readv(
    int fildes,
    const struct iovec *iov,
    size_t iovcnt
);
```

DESCRIPTION

read() attempts to read *nbyte* bytes from the file associated with the file descriptor into the buffer pointed to by *buf*. **readv()** performs the same action, but scatters the input data into the *iovcnt* buffers specified by the elements of the *iovec* array: *iov*[0], *iov*[1], ..., *iov*[*iovcnt* - 1].

For **readv()**, the **iovec** structure is defined as:

```
struct iovec {
    caddr_t  iov_base;
    int      iov_len;
};
```

Each **iovec** entry specifies the base address and length of an area in memory where data should be placed. **readv()** always fills one area completely before proceeding to the next area. The **iovec** array can be at most **MAXIOV** long.

On devices capable of seeking, the **read()** starts at a position in the file given by the file offset associated with *fildes*. Upon return from **read()**, the file offset is incremented by the number of bytes actually read.

Devices incapable of seeking always read from the current position. The value of a file offset associated with such a device is undefined.

When attempting to read from a regular file with enforcement-mode file and record locking set (see *chmod*(2)), and the segment of the file to be read is blocked by a write lock owned by another process, the behavior is determined by the **O_NDELAY** and **O_NONBLOCK** file status flags:

- If **O_NDELAY** or **O_NONBLOCK** is set, **read()** returns -1 and **errno** is set to EAGAIN.
- If **O_NDELAY** and **O_NONBLOCK** are clear, **read()** does not return until the blocking write lock is removed.

When attempting to read from an empty pipe (or FIFO):

- If no process has the pipe open for writing, the read returns a 0.
- If some process has the pipe open for writing and **O_NONBLOCK** is set, the read returns -1 and **errno** is set to EAGAIN.
- If **O_NDELAY** is set, the read returns a 0.
- If some process has the pipe open for writing and **O_NDELAY** and **O_NONBLOCK** are clear, the read blocks until data is written to the file or the file is no longer open for writing.

When attempting to read a file associated with a tty that has no data currently available:

- If **O_NONBLOCK** is set, the read returns -1 and **errno** is set to EAGAIN.
- If **O_NDELAY** is set, the read returns 0.
- If **O_NDELAY** and **O_NONBLOCK** are clear, the read blocks until data becomes available.

If **read()** is interrupted by a signal after it has successfully read some data, it returns the number of bytes actually read and placed in the buffer before the interrupt occurred. If **read()** is interrupted before any data is successfully read, **read()** returns -1 and sets **errno** to EINTR.

RETURN VALUE

Upon successful completion, `read()` returns the number of bytes actually read and placed in the buffer; this number may be less than *nbyte* if:

- The file is associated with a communication line (see *ioctl*(2) and *termio*(7)), or
- The number of bytes left in the file is less than *nbyte* bytes.
- `read()` was interrupted by a signal after it had successfully read some, but not all of the data requested.

When an end-of-file is reached, a value of 0 is returned. Otherwise, a -1 is returned and `errno` is set to indicate the error.

ERRORS

`read()` fails if any of the following conditions are encountered:

[EBADF]	*fildes* is not a valid file descriptor open for reading.
[EINTR]	A signal was caught before any data was transferred (see *sigvector*(2)).
[EAGAIN]	Enforcement-mode file and record locking is set, `O_NDELAY` or `O_NONBLOCK` is set, and there is a blocking write lock.
[EDEADLK]	A resource deadlock would occur as a result of this operation (see *lockf*(2) and *fcntl*(2)).
[EFAULT]	*buf* points outside the allocated address space. Reliable detection of this error is implementation dependent.
[EIO]	The process is in a background process group and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the `SIGTTIN` signal or the process group of the process is orphaned.
[EIO]	An I/O error occurred while reading from the device corresponding to *fildes*.
[EISDIR]	An attempt was made to read a directory on an NFS file system using the `read()` system call.
[ENOLCK]	The system record lock table is full, preventing the read from sleeping until the blocking write lock is removed.

In addition, `readv()` can return one of the following errors:

[EFAULT]	*iov_base* or *iov* points outside of the allocated address space. The reliable detection of this error is implementation dependent.
[EINVAL]	*iovcnt* is less then or equal to 0, or greater than `MAXIOV`.
[EINVAL]	The sum of *iov_len* values in the *iov* array exceeded `UINT_MAX` defined in `<limits.h>` (see *limits*(5)).

EXAMPLES

Assuming a process opened a file for reading, the following call to *read*(2) reads `BUFSIZ` bytes from the file into the buffer pointed to by *mybuf*:

```
#include <stdio.h>    /* include this for BUFSIZ definition */

char mybuf[BUFSIZ];
int nbytes, fildes;

nbytes = read (fildes, mybuf, BUFSIZ);
```

WARNINGS

Record locking might not be enforced by the system, depending on the setting of the file's mode bits (see *lockf*(2)).

Character-special devices, and raw disks in particular, apply constraints on how `read()` can be used. See the specific Section (7) entries for details on particular devices.

Check all references to *signal*(5) for appropriateness on systems that support *sigvector*(2). `sigvector()` can affect the behavior described on this page.

In general, avoid using `read()` to get the contents of a directory; use the `readdir()` library routine (see *directory*(3C)).

DEPENDENCIES

NFS

When obtaining the contents of a directory on an NFS file system, the `readdir()` library routine must be used (see *directory*(3C)). `read()` returns with an error if used to read a directory using NFS.

AUTHOR

`read()` was developed by HP, AT&T, and the University of California, Berkeley.

SEE ALSO

creat(2), dup(2), fcntl(2), ioctl(2), lockf(2), open(2), pipe(2), select(2), ustat(2), tty(7), directory(3C).

STANDARDS CONFORMANCE

`read()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

readlink - read value of a symbolic link

SYNOPSIS

```
#include <symlink.h>

ssize_t readlink(
    const char *path,
    char *buf,
    size_t bufsiz
);
```

DESCRIPTION

`readlink()` obtains the path name pointed to by the symbolic link, *path*. This path name is placed in the buffer *buf*, which has size *bufsiz*.

RETURN VALUE

If `readlink()` succeeds, it returns the count of characters placed in the buffer. If an error occurs, it returns -1 and sets `errno` to indicate the error.

ERRORS

`readlink()` fails if any of the following conditions is encountered:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENAMETOOLONG]	A component of *path* exceeds bytes while is in effect, or *path* exceeds bytes.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[ELOOP]	Too many symbolic links were encountered in translating the path name.
[EINVAL]	The named file is not a symbolic link.
[EFAULT]	*buf* points outside the process' allocated address space. Reliable detection of this error is implemenation dependent.

DEPENDENCIES

Series 300,400, and 700:

If the length of the path name string is less than *bufsiz*, the string will be null terminated when returned. If the length of the path name string is exactly *bufsiz*, the string will not be null terminated when returned. If the length of the path name string exceeds *bufsiz*, `readlink()` returns –1 and sets `errno` to:

[ERANGE]	The length of the path name string read from the symbolic link exceeds *bufsiz*.

Series 800:

The path name is not null terminated when returned.

AUTHOR

`readlink()` was developed by the University of California, Berkeley.

SEE ALSO

stat(2), lstat(2), symlink(2), symlink(4).

STANDARDS CONFORMANCE

`readlink()`: AES [Series 300/400/700 only]

NAME

reboot - boot the system

SYNOPSIS

```
#include <sys/reboot.h>

int reboot (int howto, ...
    /* const char *device_file,
       const char *filename,
       const char *filename,
       const char *server_linkaddress */
);
```

DESCRIPTION

`reboot()` causes the system to reboot. *howto* is a mask of reboot options (see `<sys/reboot.h>`), specified as follows:

`RB_AUTOBOOT`	A file system sync is performed (unless `RB_NOSYNC` is set) and the processor is rebooted from the default device and file.
`RB_HALT`	The processor is simply halted. A sync of the file system is performed unless the `RB_NOSYNC` flag is set. `RB_HALT` should be used with caution.
`RB_NOSYNC`	A sync of the file system is not performed.
`RB_NEWDEVICE`	The *device_file* argument is used as the file name of the device from which to reboot.
`RB_NEWFILE`	The *filename* argument is used as the name of the file being rebooted.
`RB_NEWSERVER`	The additional optional parameter, *server_linkaddress*, specifies the ETHERNET link address of a new boot server. The *server_linkaddress* is a 12-character hexadecimal number that has the same format as the machine ID field of `/etc/clusterconf`. The `0x` prefix is optional. This allows a standalone system or HP cluster server to reboot and join an HP cluster as a client node, or for an existing client to join a different HP cluster.

device_file specifies the "boot device", the device from which the reboot occurs. *device_file* must be a block or character special file name and is used only if the `RB_NEWDEVICE` option is set.

If the `RB_NEWFILE` option is set, *filename* specifies the "boot file", the name of the file being rebooted. This file is loaded into memory by the bootstrap then control is passed to it.

If the `RB_NEWSERVER` option is set, *reboot*(2) does not verify that *server_linkaddress* is a valid ETHERNET address, nor that the specified server is valid or provides the required service.

If the boot device is not a LAN device, the *server_linkaddress* information is ignored. The boot device is considered a LAN device if the previous boot was from a LAN device or if a LAN device is specified via the `RB_NEWDEVICE` option.

Unless the `RB_NOSYNC` flag has been specified, *reboot*(2) unmounts all mounted file systems and marks them clean so that it will not be necessary to run *fsck*(1M) on these file systems when the system reboots.

Only users with appropriate privileges can reboot a machine.

RETURN VALUE

If successful, this call never returns. Otherwise, a -1 is returned and `errno` is set to indicate the error.

ERRORS

`reboot()` fails if any of the following conditions are encountered:

[EFAULT]	*device_file* points outside the allocated address space of the process.
[ENAMETOOLONG]	the path name specified by *device_file* exceeds `PATH_MAX` bytes, or the length of a component of the path name exceeds `NAME_MAX` bytes while `_POSIX_NO_TRUNC` is in effect.
[EINVAL]	*device_file* is not a block or a character device.

[ENET]	The device specified by *device_file* is remote.
[ENOENT]	The file specified by *device_file* does not exist.
[ENOTDIR]	A component of the path prefix specified by *device_file* is not a directory.
[ENXIO]	The device named by *device_file* does not exist.
[EPERM]	The effective user ID of the caller is not a user with appropriate privileges.

DEPENDENCIES

Series 300/400

filename must be one of the files listed by the boot ROM at power-up.

The default device, file, and server for **RB_AUTOBOOT** are those from which the system was previously booted.

If the **RB_NEWDEVICE** option is used and *device_file* specifies a LAN device, the **RB_NEWSERVER** option and *server_linkaddress* parameter must also be used.

If an invalid *server_linkaddress* is specified with the **RB_NEWSERVER** option, or if the requested server does not respond, the Series 300/400 boot ROM displays the message **BOOTING A SYSTEM** and retries indefinitely, or until the requested server responds, or the system is rebooted manually.

Series 700/800

The **RB_NEWDEVICE**, **RB_NEWFILE**, and **RB_NEWSERVER** options and the *device_file*, *filename* and *server_linkaddress* parameters are ignored. Therefore, none of the errors associated with them are returned.

The default file and device for **RB_AUTOBOOT** are **/hp-ux** on the current root device.

AUTHOR

reboot() was developed by HP and the University of California, Berkeley.

SEE ALSO

reboot(1M), clusterconf(4).

NAME

recv, recvfrom, recvmsg - receive a message from a socket

SYNOPSIS

```
#include <sys/socket.h>

int recv(int s, void *buf, int len, int flags);

int recvfrom(
    int s,
    void *buf,
    int len,
    int flags,
    void *from,
    int *fromlen);

int recvmsg(int s, struct msghdr msg[], int flags);
```

DESCRIPTION

recv(), **recvfrom()**, and **recvmsg()** are used to receive messages from a socket.

s is a socket descriptor from which messages are received. *buf* is a pointer to the buffer into which the messages are placed. *len* is the maximum number of bytes that can fit in the buffer referenced by *buf*.

If the socket uses connection-based communications, such as a SOCK_STREAM socket, these calls can only be used after the connection has been established (see *connect*(2)). For connectionless sockets such as SOCK_DGRAM, these calls can be used whether a connection has been specified or not.

recvfrom() operates in the same manner as **recv()** except that it is able to return the address of the socket from which the message was sent. For connected datagram sockets, **recvfrom()** simply returns the same address as **getpeername()** (see *getpeername*(2)). For stream sockets, **recvfrom()** retrieves data in the same manner as **recv()**, but does not return the socket address of the sender. If *from* is non-zero, the source address of the message is placed in the socket address structure pointed to by *from*. *fromlen* is a value-result parameter, initialized to the size of the structure associated with *from*, and modified on return to indicate the actual size of the address stored there. If the memory pointed to by *from* is not large enough to contain the entire address, only the first *fromlen* bytes of the address are returned.

The length of the message is returned.

For message-based sockets such as SOCK_DGRAM, the entire message must be read in a single operation. If a message is too long to fit in the supplied buffer, the excess bytes are discarded. For stream-based sockets such as SOCK_STREAM, there is no concept of message boundaries. In this case, data is returned to the user as soon as it becomes available, and no data is discarded. See the AF_CCITT section below for a list of the exceptions to this behavior for connections in the address family AF_CCITT.

recvmsg() performs the same action as **recv()**, but scatters the read data into the buffers specified in the **msghdr** structure. This structure is defined in **<sys/socket.h>**, and has the following form :

```
struct msghdr {
    caddr_t   msg_name;          /* optional address */
    int       msg_namelen;       /* size of address  */
    struct    iovec *msg_iov;    /* scatter array for data */
    int       msg_iovlen;        /* # of elements in msg_iov */
    caddr_t   msg_accrights;     /* access rights */
    int       msg_accrightslen;  /* size of msg_accrights */
}
```

msg_name is the destination address if the socket is unconnected; *msg_name* may be a null pointer if no name is specified. *msg_iov* is the location of the scatter/gather data. *msg_accrights* specifies a buffer to receive any access rights sent along with the message. Access rights are limited to file descriptors of size int. If access rights are not being transferred, set the msg_acccrights field to NULL. Access rights are supported only for AF_UNIX.

If no data is available to be received, **recv()** waits for a message to arrive unless non-blocking mode is enabled. There are three ways to enable non-blocking mode:

- With the FIOSNBIO ioctl() request,
- With the O_NONBLOCK fcntl() flag,
- With the O_NDELAY fcntl() flag.

If non-blocking I/O is enabled using FIOSNBIO or the equivalent FIONBIO request (defined in <sys/ioctl.h> and explained in *ioctl*(2), *ioctl*(5) and *socket*(7), although use of FIONBIO is not recommended), the recv() request completes in one of three ways:

- If there is enough data available to satisfy the entire request, recv() completes successfully, having read all of the data, and returns the number of bytes read.
- If there is not enough data available to satisfy the entire request, recv() complete successfully, having read as much data as possible, and returns the number of bytes it was able to read.
- If there is no data available, recv() fails and errno is set to EWOULDBLOCK.

If non-blocking I/O is disabled using FIOSNBIO, recv() always executes completely (blocking as necessary) and returns the number of bytes read.

If the O_NONBLOCK flag is set using fcntl() (defined in <sys/fcntl.h> and explained in *fcntl*(2) and *fcntl*(5)), POSIX-style non-blocking I/O is enabled. In this case, the recv() request completes in one of three ways:

- If there is enough data available to satisfy the entire request, recv() completes successfully, having read all the data, and returns the number of bytes read.
- If there is not enough data available to satisfy the entire request, recv() completes successfully, having read as much data as possible, and returns the number of bytes it was able to read.
- If there is no data available, recv() completes, having read no data, and returns -1 with errno set to EAGAIN.

If the O_NDELAY flag is set using fcntl() (defined in <sys/fcntl.h> and explained in *fcntl*(2) and *fcntl*(5)), non-blocking I/O is enabled. In this case, the recv() request completes in one of three ways:

- If there is enough data available to satisfy the entire request, recv() completes successfully, having read all the data, and returns the number of bytes read.
- If there is not enough data available to satisfy the entire request, recv() completes successfully, having read as much data as possible, and returns the number of bytes it was able to read.
- If there is no data available, recv() completes successfully, having read no data, and returns 0.

If the O_NONBLOCK or O_NDELAY flag is cleared using fcntl(), the corresponding style of non-blocking I/O, if previously enabled, is disabled. In this case, recv() always executes completely (blocking as necessary) and returns the number of bytes read.

Since both the fcntl() O_NONBLOCK and O_NDELAY flags and ioctl() FIOSNBIO request are supported, some clarification on how these features interact is necessary. If the O_NONBLOCK or O_NDELAY flag has been set, recv() requests behave accordingly, regardless of any FIOSNBIO requests. If neither the O_NONBLOCK nor O_NDELAY flag has been set, FIOSNBIO requests control the behavior of recv(). The default is that non-blocking I/O is not enabled.

select() can be used to determine when more data arrives by selecting the socket for reading.

The *flags* parameter can be set to MSG_PEEK, MSG_OOB, both, or zero. If it is set to MSG_PEEK, any data returned to the user still is treated as if it had not been read. The next recv() re-reads the same data. The MSG_OOB flag is used to alert the other process with an urgent message, using a logically independent transmission channel associated with a pair of connected stream sockets. Refer to SEE ALSO below for details. For stream-based TCP SOCK_STREAM sockets, both the MSG_PEEK and MSG_OOB flags can be set at the same time. The MSG_OOB flag value is supported for stream-based TCP SOCK_STREAM sockets only. MSG_OOB is not supported for AF_UNIX sockets.

A read() call made to a socket behaves in exactly the same way as a recv() with *flags* set to zero.

AF_CCITT only:
Connections in the address family AF_CCITT support message-based sockets only. Although the user specifies connection-based communications (SOCK_STREAM), the X.25 subsystem communicates via

messages. This address family does not support SOCK_DGRAM socket types.

Normally, each **recv()** returns one complete X.25 message. If the socket is in non-blocking mode, **recv()** behaves as described above. Note that if the user specifies *len* less than the actual X.25 message size, the excess data and no error indication is returned. The size of the next available message as well as the state of MDTF, D, and Q bits can be obtained with **ioctl (X25_NEXT_MSG_STAT)**.

Connections of the address family AF_CCITT receive data in the same way as message-based connections described above, with the following additions and exceptions:

- **recvfrom()** is supported; however, the *from* and *fromlen* parameters are ignored (that is, it works in the same manner as **recv()**).
- To receive a message in fragments of the complete X.25 message, use **ioctl(X25_SET_FRAGMENT_SIZE)**. The state of the MDTF bit is 1 on all except the last fragment of the message.
- The **MSG_OOB** flag is supported.
- The **MSG_PEEK** flag is supported; the two flags can be combined.
- If a message is received that is larger than the user-controlled maximum message size (see *af_ccitt*(7F)), the X.25 subsystem RESETs the circuit, discards the data, and sends the out-of-band event **OOB_VC_MESSAGE_TOO_BIG** to the socket.

DEPENDENCIES

AF_CCITT

recvfrom() is supported; however, the *from* and *fromlen* parameters are ignored (i.e., it works in the same manner as **recv()**).

The **O_NDELAY fcntl()** call is not supported over X.25 links. Use the **FIOSNBIO ioctl()** call instead to enable non-blocking I/O.

RETURN VALUE

upon successful completion, **recv()** returns the number of bytes received. Otherwise, it returns -1 and sets **errno** to indicate the error. **recv()** returns 0 if the socket is blocking and the transport connection to the remote node fails.

DIAGNOSTICS

The call to **recv()** or **recvfrom()** fails if any of the following conditions are encountered:

[EBADF]	The argument *s* is an invalid descriptor.
[ENOTSOCK]	The argument *s* is not a socket.
[EWOULDBLOCK]	The socket is marked non-blocking and the receive operation would block.
[EINTR]	The receive was interrupted by delivery of a signal before any data was available for the receive.
[EFAULT]	An invalid pointer was specified in the *buf*, *from*, or *fromlen* parameter, or in the msghdr structure.
[EMSGSIZE]	A length in the msghdr structure is invalid.
[ETIMEDOUT]	The connection timed out during connection establishment, or due to a transmission timeout on active connection.
[ENOTCONN]	Receive on a SOCK_STREAM socket that is not yet connected.
[EINVAL]	The *len* parameter or a length in the msghdr structure is invalid; or no data is available on receive of out of band data.
[EOPNOTSUPP]	The **MSG_OOB** flag was set for a UDP SOCK_DGRAM message-based socket; or **MSG_OOB** or **MSG_PEEK** was set for any AF_UNIX socket. The **MSG_OOB** flag is only supported for stream-based TCP SOCK_STREAM sockets. Neither **MSG_PEEK** nor **MSG_OOB** is supported for AF_UNIX sockets.
	AF_CCITT Only: **recv()** was issued on a **listen()** socket.

[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[ECONNRESET]	A connection was forcibly closed by a peer.

AUTHOR

`recv()` was developed by the University of California, Berkeley

SEE ALSO

getsockopt(2), read(2), select(2), send(2), socket(2), af_ccitt(7F), inet(7F), socket(7), socketx25(7), tcp(7P), udp(7P), unix(7P).

NAME

rename - change the name of a file

SYNOPSIS

```
#include <stdio.h>

int rename(const char *source, const char *target);
```

DESCRIPTION

rename() causes file *source* to be renamed to *target*. If *target* exists, it is first removed. Both *source* and *target* must be of the same type (that is, either directories or non-directories), and must reside on the same file system.

If *target* can be created or if it existed before the call, **rename()** guarantees that an instance of *target* will exist, even if the system crashes in the midst of the operation.

If the final component of *source* is a symbolic link, the symbolic link is renamed; not the file or directory to which the symbolic link points.

RETURN VALUE

If the operation succeeds, **rename()** returns 0; otherwise it returns -1 and sets **errno** to indicate the reason for the failure.

ERRORS

rename() fails and neither file is affected if any of the following conditions are encountered:

[EACCES]	A component of either path prefix denies search permission.
[EACCES]	The requested link requires writing to a directory without write permission.
[EBUSY]	*target* or *source* is an existing directory that is the mount point for a mounted file system.
[EDQUOT]	User's disk quota block or inode limit has been reached for this file system.
[EEXIST]	*target* is a directory and is not empty.
[EFAULT]	*source* or *target* points outside the allocated address space of the process. Reliable detection of this error is implementation dependent.
[EINVAL]	*source* is a parent directory of *target*, or an attempt is made to rename . or ...
[EISDIR]	*target* is a directory, but *source* is not.
[ELOOP]	Too many symbolic links were encountered in translating either path name.
[ENAMETOOLONG]	A component of either path name exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect, or the entire length of either path name exceeds **PATH_MAX** bytes.
[ENOENT]	A component of the *source* path does not exist, or a path prefix of *target* does not exist.
[ENOSPC]	The destination directory cannot be extended because of a lack of space on the file system containing the directory.
[ENOTDIR]	A component of either path prefix is not a directory.
[ENOTDIR]	*source* is a directory, but *target* is not. [EPERM] The directory containing *source* has the sticky bit set, and neither the containing directory nor the *source* are owned by the effective user ID.
[EPERM]	The *target* file exists, the directory containing *target* has the sticky bit set, and neither the containing directory nor the *target* are owned by the effective user ID.
[EROFS]	The requested link requires writing in a directory on a read-only file system.
[EXDEV]	The paths named by *source* and *target* are on different logical devices (file systems).

AUTHOR

rename ( ) was developed by the University of California, Berkeley.

SEE ALSO

open(2).

STANDARDS CONFORMANCE

rename ( ): AES, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

rmdir - remove a directory file

SYNOPSIS

```
int rmdir(const char *path);
```

DESCRIPTION

`rmdir()` removes a directory file whose name is given by *path*. The directory must be empty (except for files `.` and `..`) before it can be removed.

RETURN VALUE

`rmdir()` returns 0 if the directory removal succeeds; otherwise, it returns –1 and sets `errno` to indicate the error.

ERRORS

`rmdir()` fails and the directory is not removed if any of the following conditions are encountered:

[EACCES]	A component of the path prefix denies search permission.
[EACCES]	Write permission is denied on the directory containing the link to be removed.
[EBUSY]	The directory to be removed is the mount point for a mounted file system.
[EEXIST]	The named directory is not empty. It contains files other than `.` and `...`
[EFAULT]	*path* points outside the process's allocated address space. The reliable detection of this error is implementation dependent.
[EINVAL]	The path is `..`
[ELOOP]	Too many symbolic links were encountered in translating the path name.
[ENAMETOOLONG]	The length of the specified path name exceeds **PATH_MAX** bytes, or the length of a component of the path name exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect.
[ENOENT]	The named file does not exist.
[ENOTDIR]	A component of the path is not a directory.
[EPERM]	The directory containing the directory to be removed has the sticky bit set and neither the containing directory nor the directory to be removed are owned by the effective user ID.
[EROFS]	The directory entry to be removed resides on a read-only file system.

AUTHOR

`rmdir()` was developed by the University of California, Berkeley.

SEE ALSO

mkdir(2), unlink(2).

STANDARDS CONFORMANCE

`rmdir()`: AES, SVID2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

rtprio - change or read real-time priority

SYNOPSIS

```
#include <sys/rtprio.h>

int rtprio(pid_t pid, int prio);
```

DESCRIPTION

`rtprio()` is used to set or read the real-time priority of a process. If *pid* is zero, it names the calling process; otherwise it gives the pid of the process. When setting the real-time priority of another process, the real or effective user ID of the calling process must match the real or saved user ID of the process to be modified, or the effective user ID of the calling process must be that of a user having appropriate privileges. The calling process must also be a member of a privilege group allowing `rtprio()` (see *getprivgrp*(2)) or the effective user ID of the calling process must be a user having appropriate privileges. Simply reading real-time priorities requires no special privilege.

Real-time scheduling policies differ from normal timesharing policies in that the real-time priority is used to absolutely order all real-time processes; this priority is not degraded over time. All real-time processes are of higher priority than normal user and system processes, although some system processes may run at real-time priorities. If there are several eligible processes at the same priority level, they are run in a round robin fashion as long as no process with higher priority intervenes. A real-time process receives CPU service until it either voluntarily gives up the CPU or is preempted by a process of equal or higher priority. Interrupts can also preempt a real-time process.

Valid real-time priorities run from zero to 127. Zero is the highest (most important) priority. This real-time priority is inherited across `fork()`s and `exec()`s.

prio specifies the following:

`0-127`	Set process to this real-time priority.
`RTPRIO_NOCHG`	Do not change real-time priority. This is used for reading the process real-time priority.
`RTPRIO_RTOFF`	Set this process to no longer have a real-time priority. It resumes a normal timesharing priority. Any process, regardless of privilege, is allowed to turn off its own real-time priority using a *pid* of zero.

EXAMPLES

The following call to `rtprio()` sets the calling process to a real-time priority of 90:

```
rtprio (0, 90);
```

RETURN VALUE

If no error occurs, `rtprio()` returns the *pid*'s former (before the call) real-time priority. If the process was not a real-time process, `RTPRIO_RTOFF` is returned. If an error occurs, `rtprio()` returns -1 and sets `errno` to indicate the error.

ERRORS

`rtprio()` fails if any of the following conditions are encountered:

[EINVAL]	*prio* is not `RTPRIO_NOCHG`, `RTPRIO_RTOFF`, or in the range of 0 through 127.
[EPERM]	The calling process is not a user having appropriate privileges, and neither its real or effective user-id match the real or saved user ID of the process indicated by *pid*.
[EPERM]	The group access list of the calling process does not contain a group having `PRIV_RTPRIO` capability and *prio* is not `RTPRIO_NOCHG`, or `RTPRIO_RTOFF` with a *pid* of zero.
[ESRCH]	No process can be found corresponding to that specified by *pid*.

DEPENDENCIES

Series 800:

Because processes executing at real-time priorities get scheduling preference over a system process executing at a lower priority, unexpected system behavior can occur after a power failure on systems that support

power-fail recovery. For example, when *init*(1M) receives the powerfail signal `SIGPWR`, it normally reloads programmable hardware such as terminal multiplexers. If a higher-priority real-time process is eligible to run after the power failure, running of `init` is delayed. This condition temporarily prevents terminal input to any process, including real-time shells of higher priority than the eligible real-time process. To avoid this situation, a real-time process should catch `SIGPWR` and suspend itself until `init` has finished its powerfail processing.

AUTHOR

`rtprio()` was developed by HP.

SEE ALSO

rtprio(1), getprivgrp(2), nice(2), plock(2), privilege(5).

WARNINGS

Normally, compute-bound programs should not be run at real-time priorities, because all time sharing work on the CPU would come to a complete halt.

NAME

select - synchronous I/O multiplexing

SYNOPSIS

```
#include <time.h>

int select(
    size_t nfds,
    int *readfds,
    int *writefds,
    int *exceptfds,
    const struct timeval *timeout
);
```

DESCRIPTION

`select()` examines the file descriptors specified by the bit masks *readfds*, *writefds*, and *exceptfds*. The bits from 0 through *nfds*-1 are examined. File descriptor *f* is represented by the bit 1<<*f* in the masks. More formally, a file descriptor is represented by:

fds[(f / BITS_PER_INT)] & (1 << (f % BITS_PER_INT))

When `select()` completes successfully it returns the three bit masks modified as follows: For each file descriptor less than *nfds*, the corresponding bit in each mask is set if the bit was set upon entry and the file descriptor is ready for reading or writing, or has an exceptional condition pending.

If *timeout* is a non-zero pointer, it specifies a maximum interval to wait for the selection to complete. If *timeout* is a zero pointer, the select waits until an event causes one of the masks to be returned with a valid (non-zero) value. To poll, the *timeout* argument should be non-zero, pointing to a zero valued timeval structure. Specific implementations may place limitations on the maximum timeout interval supported. The constant **MAX_ALARM** defined in **<sys/param.h>** specifies the implementation-specific maximum (in seconds). Whenever *timeout* specifies a value greater than this maximum, it is silently rounded down to this maximum. On all implementations, **MAX_ALARM** is guaranteed to be at least 31 days (in seconds). Note that the use of a timeout does not affect any pending timers set up by `alarm()` or `setitimer()` (see *alarm*(2) or *setitimer*(2)).

Any or all of *readfds*, *writefds*, and *exceptfds* can be given as 0 if no descriptors are of interest. If all the masks are given as 0 and *timeout* is not a zero pointer, `select()` blocks for the time specified, or until interrupted by a signal. If all the masks are given as 0 and *timeout* is a zero pointer, `select()` blocks until interrupted by a signal.

Ordinary files always select true whenever selecting on reads, writes, and/or exceptions.

EXAMPLES

The following call to `select()` checks if any of 4 terminals are ready for reading. `select()` times out after 5 seconds if no terminals are ready for reading. Note that the code for opening the terminals or reading from the terminals is not shown in this example. Also, note that this example must be modified if the calling process has more than 32 file descriptors open. Following this first example is an example of select with more than 32 file descriptors.

```
#define MASK(f) (1 << (f))
#define NTTYS 4

int tty[NTTYS];
int ttymask[NTTYS];
int readmask = 0;
int readfds;
int nfound, i;
struct timeval timeout;

   /* First open each terminal for reading and put the
    * file descriptors into array tty[NTTYS].  The code
    * for opening the terminals is not shown here.
    */

for (i=0; i < NTTYS; i++) {
   ttymask[i] = MASK(tty[i]);
```

```
    readmask |= ttymask[i];
}

timeout.tv_sec  = 5;
timeout.tv_usec = 0;
readfds = readmask;

/* select on NTTYS+3 file descriptors if stdin, stdout
 * and stderr are also open
 */
if ((nfound = select (NTTYS+3, &readfds, 0, 0, &timeout)) == -1)
   perror ("select failed");
else if (nfound == 0)
   printf ("select timed out \n");
else for (i=0; i < NTTYS; i++)
   if (ttymask[i] & readfds)
      /* Read from tty[i].  The code for reading
       * is not shown here.
       */
   else printf ("tty[%d] is not ready for reading \n",i);
```

The following example is the same as the previous example, except that it works for more than 32 open files. Definitions for **howmany**, **fd_set**, and **NFDBITS** are in **<sys/types.h>**.

```
#include <sys/param.h>
#include <sys/types.h>
#include <sys/time.h>

#define MASK(f) (1 << (f))
#define NTTYS NOFILE - 3
#define NWORDS  howmany(FD_SETSIZE, NFDBITS)

int tty[NTTYS];
int ttymask[NTTYS];
struct fd_set readmask, readfds;
int nfound, i, j, k;
struct timeval timeout;

   /* First open each terminal for reading and put the
    * file descriptors into array tty[NTTYS].  The code
    * for opening the terminals is not shown here.
    */

   for (k=0; k < NWORDS; k++)
      readmask.fds_bits[k] = 0;

   for (i=0, k=0; i < NTTYS && k < NWORDS; k++)
      for (j=0; j < NFDBITS && i < NTTYS; j++, i++) {
         ttymask[i] = MASK(tty[i]);
         readmask.fds_bits[k] |= ttymask[i];
      }

   timeout.tv_sec  = 5;
   timeout.tv_usec = 0;
   for (k=0; k < NWORDS; k++)
      readfds.fds_bits[k] = readmask.fds_bits[k];

   /* select on NTTYS+3 file descriptors if stdin, stdout
    * and stderr are also open
    */
   if ((nfound = select (NTTYS+3, &readfds, 0, 0, &timeout)) == -1)
      perror ("select failed");
   else if (nfound == 0)
      printf ("select timed out \n");
   else for (i=0, k=0; i < NTTYS && k < NWORDS; k++)
```

```
for (j=0; j < NFDBITS && i < NTTYS; j++, i++)
   if (ttymask[i] & readfds.fds_bits[k])
      /* Read from tty[i].  The code for reading
       * is not shown here.
       */
   else printf ("tty[%d] is not ready for reading \n",i);
```

RETURN VALUE

`select()` returns the number of descriptors contained in the bit masks, or -1 if an error occurred. If the time limit expires, `select()` returns 0 and all the masks are cleared.

ERRORS

`select()` fails if any of the following conditions are encountered:

[EBADF]	One or more of the bit masks specified an invalid descriptor.
[EINTR]	A signal was delivered before any of the selected for events occurred or before the time limit expired.
[EFAULT]	One or more of the pointers was invalid. The reliable detection of this error is implementation dependent.
[EINVAL]	Invalid timeval passed for timeout.
[EINVAL]	The value of *nfds* is less than zero.

WARNINGS

Check all references to *signal*(5) for appropriateness on systems that support *sigvector*(2). *sigvector*(2) can affect the behavior described on this page.

The file descriptor masks are always modified on return, even if the call returns as the result of a timeout.

DEPENDENCIES

Series 300/400

`select()` supports the following devices and file types:

- pipes
- fifo special files (named pipes)
- All serial interfaces
- All ITEs (internal terminal emulators) and HP-HIL input devices
- *pty*(7) special files
- sockets
- HP 98643 LAN interface card driver

File types not supporting `select()` always return true.

Series 700/800

`select()` supports the following devices and file types:

- pipes
- fifo special files (named pipes)
- all serial devices
- All ITEs (internal terminal emulators) and HP-HIL input devices
- *hpib*(7) special files
- *gpio*(7) special files (Series 800 Only for Release 8.0)
- *lan*(7) special files
- *pty*(7) special files
- sockets

The convention for device files that do not support `select()` is to always return true for those conditions the user is selecting on.

Consult individual device manual entries to determine the extent to which any particular driver supports *select*.

HP Clustered Environment

In a clustered environment, `select()` is not supported for distributed fifos; i.e., fifos that are open simultaneously on multiple machines. In this case an error of EINVAL is returned.

AUTHOR

`select()` was developed by HP and the University of California, Berkeley.

SEE ALSO

fcntl(2), read(2), write(2).

NAME

semctl - semaphore control operations

SYNOPSIS

```
#include <sys/sem.h>

int semctl(int semid,
int semnum,
int cmd, ...
/* arg */
);
```

DESCRIPTION

`semctl()` provides a variety of semaphore control operations as specified by *cmd*.

The following *cmd*s are executed with respect to the semaphore specified by *semid* and *semnum*:

`GETVAL` Return the value of semval (see *semaphore identifier* in *glossary*(9)). Requires Read permission.

`SETVAL` Set the value of semval to *arg*, where *arg* is the fourth argument of `semctl()` taken as an `int`. When this *cmd* is successfully executed, the semadj value corresponding to the specified semaphore in all processes is cleared. Requires Alter permission.

`GETPID` Return the value of sempid. Requires Read permission.

`GETNCNT` Return the value of semncnt. Requires Read permission.

`GETZCNT` Return the value of semzcnt. Requires Read permission.

The following *cmd*s return and set, respectively, every semval in the set of semaphores.

`GETALL` Place semvals into array pointed to by *arg*, where *arg* is the fourth argument of `semctl()` taken as a pointer to `unsigned short int`. Requires Read permission.

`SETALL` Set semvals according to the array pointed to by *arg*, where *arg* is the fourth argument of `semctl()` taken as a pointer to `unsigned short int`. When this *cmd* is successfully executed, the semadj values corresponding to each specified semaphore in all processes are cleared. Requires Alter permission.

The following *cmd*s are also available:

`IPC_STAT` Place the current value of each member of the data structure associated with *semid* into the structure pointed to by *arg*, where *arg* is the fourth argument of `semctl()` taken as a pointer to `struct semid_ds`. The contents of this structure are defined in *glossary*(9). Requires Read permission.

`IPC_SET` Set the value of the following members of the data structure associated with *semid* to the corresponding value found in the structure pointed to by *arg*, where *arg* is the fourth argument of `semctl()` taken as a pointer to `struct semid_ds`:

```
sem_perm.uid
sem_perm.gid
sem_perm.mode    /* only low 9 bits */
```

This *cmd* can only be executed by a process that has an effective user ID equal to either that of super-user or to the value of either `sem_perm.uid` or `sem_perm.cuid` in the data structure associated with *semid*.

`IPC_RMID`
Remove the semaphore identifier specified by *semid* from the system and destroy the set of semaphores and data structure associated with it. This *cmd* can only be executed by a process that has an effective user ID equal to either that of super-user or to the value of either `sem_perm.uid` or `sem_perm.cuid` in the data structure associated with *semid*.

EXAMPLES

The following call to `semctl()` initializes the set of 4 semaphores to the values 0, 1, 0, and 1 respectively.

This example assumes the process has a valid semid representing a set of 4 semaphores as shown in the *semget*(2) manual entry. For an example of performing "P" and "V" operations on the semaphores below, refer to *semop*(2).

```
ushort semarray[4];

semarray[0] = 0;
semarray[1] = 1;
semarray[2] = 0;
semarray[3] = 1;

semctl (mysemid, 0, SETALL, semarray);
```

RETURN VALUE

Upon successful completion, the value returned depends on *cmd* as follows:

GETVAL	The value of semval.
GETNCNT	The value of semncnt.
GETZCNT	The value of semzcnt.
GETPID	The value of sempid.

All others return a value of 0.

Otherwise, a value of −1 is returned and **errno** is set to indicate the error.

ERRORS

semctl() fails if any of the following conditions are encountered:

[EACCES]	Operation permission is denied to the calling process (see *semaphore operation permissions* in *glossary*(9).
[EFAULT]	*cmd* is equal to **GETVAL**, **SETVAL**, **GETALL**, **SETALL**, **IPC_STAT**, or **IPC_SET**, and *arg*.
[EINVAL]	*semid* is not a valid semaphore identifier.
[EINVAL]	*semnum* is less than zero or greater than or equal **sem_nsems**.
[EINVAL]	*cmd* is not a valid command.
[EPERM]	*cmd* is equal to **IPC_RMID** or **IPC_SET** and the effective user ID of the calling process is not equal to that of super-user and it is not equal to the value of either **sem_perm.uid** or **sem_perm.cuid** in the data structure associated with *semid*.
[ERANGE]	*cmd* is **SETVAL** or **SETALL** and the value to which semval is to be set is greater than the system imposed maximum.

SEE ALSO

ipcrm(1), ipcs(1), semget(2), semop(2), stdipc(3C).

STANDARDS CONFORMANCE

semctl(): SVID2, XPG2, XPG3, XPG4

NAME

semget - get set of semaphores

SYNOPSIS

```
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

DESCRIPTION

`semget()` returns the semaphore identifier associated with *key*.

A semaphore identifier and associated data structure and set containing *nsems* semaphores are created for *key* if one of the following is true:

key is equal to `IPC_PRIVATE`. This call creates a new identifier, subject to available resources. The identifier is never returned by another call to `semget()` until it has been released by a call to `semctl()`. The identifier should be used among the calling process and its descendents; however, it is not a requirement. The resource can be accessed by any process having the proper permissions.

key does not already have a semaphore identifier associated with it, and (*semflg* & `IPC_CREAT`) is "true".

Specific behavior can be requested by ORing the following masks into *semflg*.

`IPC_CREAT`: Create a semaphore identifier if one does not already exist for *key*.

`IPC_EXCL`: If `IPC_CREAT` is specified and *key* already has a semaphore identifier associated with it, return an error.

The low-order 9 bits of *semflg* are the semaphore operation permissions which are defined in *glossary*(9).

Upon creation, the data structure associated with the new semaphore identifier is initialized as follows:

In the operation-permission structure, `sem_perm.cuid` and `sem_perm.uid` are set equal to the effective-user-ID of the calling process, while `sem_perm.cgid` and `sem_perm.gid` are set to the effective-group-ID of the calling process.

The low-order 9 bits of `sem_perm.mode` are set equal to the low-order 9 bits of *semflg*.

`sem_nsems` is set equal to the value of *nsems*.

`sem_otime` is set equal to 0 and `sem_ctime` is set equal to the current time.

EXAMPLES

The following call to `semget()` returns a semid associated with the key returned by `ftok("myfile", 'A')`. If a semid associated with the key does not exist, a new semid, set of 4 semaphores, and associated data structure will be created. If a semid for the key already exists, the semid is simply returned.

```
int semid;
     mysemid = semget (ftok("myfile",'A'), 4, IPC_CREAT | 0600);
```

RETURN VALUE

Upon successful completion, a non-negative integer, namely a semaphore identifier, is returned. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

ERRORS

`semget()` fails if one or more of the following is true:

[EINVAL]	*nsems* is either less than or equal to zero or greater than the system-imposed limit.
[EACCES]	A semaphore identifier exists for *key*, but operation permission as specified by the low-order 9 bits of *semflg* would not be granted.
[EINVAL]	A semaphore identifier exists for *key*, but the number of semaphores in the set associated with it is less than *nsems*, and *nsems* is not equal to zero.
[ENOENT]	A semaphore identifier does not exist for *key* and (*semflg* & `IPC_CREAT`) is "false".
[ENOSPC]	A semaphore identifier is to be created, but the system-imposed limit on the maximum number of allowed semaphore identifiers system wide would be exceeded.

[EEXIST] A semaphore identifier exists for *key* but ((*semflg*& `IPC_CREAT`) && (*semflg* & `IPC_EXCL`)) is "true".

SEE ALSO

ipcrm(1), ipcs(1), semctl(2), semop(2), stdipc(3C).

STANDARDS CONFORMANCE

`semget()`: SVID2, XPG2, XPG3, XPG4

NAME

semop - semaphore operations

SYNOPSIS

```
#include <sys/sem.h>

int semop(
    int semid,
    struct sembuf *sops,
    unsigned int nsops
);
```

DESCRIPTION

semop() is used to atomically perform an array of semaphore operations on the set of semaphores associated with the semaphore identifier specified by *semid*. *sops* is a pointer to the array of semaphore-operation structures. *nsops* is the number of such structures in the array. The contents of each structure includes the following members:

```
ushort    sem_num;      /* semaphore number */
short     sem_op        /* semaphore operation */
short     sem_flg;      /* operation flags */
```

Each semaphore operation specified by *sem_op* is performed on the corresponding semaphore specified by *semid* and *sem_num*. Semaphore array operations are atomic in that none of the semaphore operations are performed until blocking conditions on all of the semaphores in the array have been removed.

sem_op specifies one of three semaphore operations as follows:

If *sem_op* is a negative integer, one of the following occurs:

If semval (see *semaphore identifier* in *glossary*(9)) is greater than or equal to the absolute value of *sem_op*, the absolute value of *sem_op* is subtracted from semval. Also, if (*sem_flg* & **SEM_UNDO**) is "true", the absolute value of *sem_op* is added to the calling process's semadj value (see *glossary*(9) and *exit*(2)) for the specified semaphore.

If semval is less than the absolute value of *sem_op* and (*sem_flg* & **IPC_NOWAIT**) is "true", **semop()** returns immediately.

If semval is less than the absolute value of *sem_op* and (*sem_flg* & **IPC_NOWAIT**) is "false", **semop()** increments the semncnt associated with the specified semaphore and suspend execution of the calling process until one of the following conditions occur:

semval becomes greater than or equal to the absolute value of *sem_op*. When this occurs, the value of semncnt associated with the specified semaphore is decremented, the absolute value of *sem_op* is subtracted from semval and, if (*sem_flg* & **SEM_UNDO**) is "true", the absolute value of *sem_op* is added to the calling process's semadj value for the specified semaphore.

The semid for which the calling process is awaiting action is removed from the system (see *semctl*(2)). When this occurs, **errno** is set equal to EIDRM, and a value of -1 is returned.

The calling process receives a signal that is to be caught. When this occurs, the value of semncnt associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in *signal*(5).

If *sem_op* is a positive integer, the value of *sem_op* is added to semval and, if (*sem_flg* & **SEM_UNDO**) is "true", the value of *sem_op* is subtracted from the calling process's semadj value for the specified semaphore.

If *sem_op* is zero, one of the following occurs:

If semval is zero, **semop()** proceeds to the next semaphore operation specified by *sops*, or returns immediately if this is the last operation.

If semval is not equal to zero and (*sem_flg* & **IPC_NOWAIT**) is "true", **semop()** returns immediately.

If semval is not equal to zero and (*sem_flg* & **IPC_NOWAIT**) is "false", **semop()** increments the semzcnt associated with the specified semaphore and suspends execution of the calling process until one of the following occurs:

semval becomes zero, at which time the value of semzcnt associated with the specified semaphore is decremented.

The semid for which the calling process is awaiting action is removed from the system. When this occurs, **errno** is set equal to EIDRM, and a value of -1 is returned.

The calling process receives a signal that is to be caught. When this occurs, the value of semzcnt associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in *signal*(5).

EXAMPLES

The following call to **semop()** atomically performs a "P" or "get" operation on the second semaphore in the semaphore set and a "V" or "release" operation on the third semaphore in the set. This example assumes the process has a valid semid which represents a set of 4 semaphores as shown on the *semget*(2) manual page. It also assumes that the semvals of the semaphores in the set have been initialized as shown in the *semctl*(2) manual entry.

```
struct sembuf sops[4];

sops[0].sem_num = 1;
sops[0].sem_op  = -1;          /* P (get) */
sops[0].sem_flg = 0;
sops[1].sem_num = 2;
sops[1].sem_op  =  1;          /* V (release) */
sops[1].sem_flg = 0;

semop (mysemid, sops, 2);
```

RETURN VALUE

If **semop()** returns due to the receipt of a signal, a value of -1 is returned to the calling process and **errno** is set to EINTR. If it returns due to the removal of a *semid* from the system, a value of -1 is returned and **errno** is set to EIDRM.

Upon successful completion, a non-negative value is returned. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

ERRORS

semop() fails if one or more of the following is true for any of the semaphore operations specified by *sops*:

[EINVAL] *semid* is not a valid semaphore identifier.

[EFBIG] *sem_num* is less than zero or greater than or equal to the number of semaphores in the set associated with *semid*.

[E2BIG] *nsops* is greater than the system-imposed maximum.

[EACCES] Operation permission is denied to the calling process (see *glossary*(9)).

[EAGAIN] The operation would result in suspension of the calling process but (*sem_flg* & **IPC_NOWAIT**) is "true".

[ENOSPC] The limit on the number of individual processes requesting an **SEM_UNDO** would be exceeded.

[EINVAL] The number of individual semaphores for which the calling process requests a **SEM_UNDO** would exceed the limit.

[ERANGE] An operation would cause a semval to overflow the system-imposed limit.

[ERANGE] An operation would cause a semadj value to overflow the system-imposed limit.

[EFAULT] *sops* points to an illegal address. The reliable detection of this error will be implementation dependent.

Upon successful completion, the value of sempid for each semaphore specified in the array pointed to by *sops* is set equal to the process ID of the calling process. The value of **sem_otime** in the data

structure associated with the semaphore identifier will be set to the current time.

WARNINGS

Check all references to *signal*(5) for appropriateness on systems that support *sigvector*(2). *sigvector*(2) can affect the behavior described on this page.

SEE ALSO

ipcs(1), exec(2), exit(2), fork(2), semctl(2), semget(2), stdipc(3C), signal(5).

STANDARDS CONFORMANCE

`semop()`: SVID2, XPG2, XPG3, XPG4

NAME

send, sendto, sendmsg - send a message from a socket

SYNOPSIS

```
#include <sys/socket.h>

int send(int s, const void *msg, int len, int flags);

int sendto(
    int s,
    const void *msg,
    int len,
    int flags,
    const void *to,
    int tolen);

int sendmsg(int s, const struct msghdr msg[], int flags);
```

DESCRIPTION

send(), **sendto()**, and **sendmsg()** are used to transmit a message to another socket. **send()** can be used only when the socket is in a connected state, whereas **sendto()** and **sendmsg()** can be used at any time. **sendmsg()** allows the send data to be gathered from several buffers specified in the **msghdr** structure. See *recv*(2) for a description of the **msghdr** structure.

s is a socket descriptor that specifies the socket on which the message will be sent. *msg* points to the buffer containing the message.

If the socket uses connection-based communications, such as a SOCK_STREAM socket, these calls can only be used after the connection has been established (see *connect*(2)). In this case, any destination specified by *to* is ignored. For connectionless sockets, such as SOCK_DGRAM, **sendto()** must be used unless the destination address has already been specified by **connect()**. If the destination address has been specified and **sendto()** is used, an error results if any address is specified by *to*.

The address of the target is contained in a socket address structure pointed at by *to*, with *tolen* specifying the size of the structure.

If a **sendto()** is attempted on a SOCK_DGRAM socket before any local address has been bound to it, the system automatically selects a local address to be used for the message. In this case, there is no guarantee that the same local address will be used for successive **sendto()** requests on the same socket.

The length of the message is given by *len*, in bytes. The length of data actually sent is returned. If the message is too long to pass atomically through the underlying protocol, the message is not transmitted, -1 is returned, and **errno** is set to EMSGSIZE. For SOCK_DGRAM sockets, this size is fixed by the implementation (see the DEPENDENCIES section below). Otherwise there is no size limit.

No indication of failure to deliver is implicit in a **send/sendto**. Return values of -1 indicate some locally-detected errors.

If no buffer space is available to hold the data to be transmitted, **send()** blocks unless non-blocking mode is enabled. There are three ways to enable non-blocking mode:

- With the **FIOSNBIO ioctl()** request,
- With the **O_NONBLOCK** flag, and
- With the **O_NDELAY fcntl()** flag.

If non-blocking I/O is enabled using **FIOSNBIO** or the equivalent **FIONBIO** request (defined in **<sys/ioctl.h>** and explained in *ioctl*(2) *ioctl*(5), and *socket*(7)), although use of **FIONBIO** is not recommended, the **send()** request completes in one of three ways:

- If there is enough space available in the system to buffer all the data, **send()** completes successfully, having written out all of the data, and returns the number of bytes written.
- If there is not enough space in the buffer to write out the entire request, **send()** completes successfully, having written as much data as possible, and returns the number of bytes it was able to write.

- If there is no space in the system to buffer any of the data, **send()** fails, having written no data, and **errno** is set to EWOULDBLOCK.

If non-blocking I/O is disabled using **FIOSNBIO**, **send()** always executes completely (blocking as necessary) and returns the number of bytes written.

If the **O_NONBLOCK** flag is set using **fcntl()** (defined in **<sys/fcntl.h>** and explained in *fcntl*(2) and *fcntl*(5)), POSIX-style non-blocking I/O is enabled. In this case, the **send()** request completes in one of three ways:

- If there is enough space available in the system to buffer all the data, **send()** completes successfully, having written out all of the data, and returns the number of bytes written.
- If there is not enough space in the buffer to write out the entire request, **send()** completes successfully, having written as much data as possible, and returns the number of bytes it was able to write.
- If there is no space in the system to buffer any of the data, **send()** completes, having written no data, and returns -1, with **errno** set to EAGAIN.

If the **O_NDELAY** flag is set using **fcntl()** (defined in **<sys/fcntl.h>** and explained in *fcntl*(2) and *fcntl*(5)), non-blocking I/O is enabled. In this case, the **send()** request completes in one of three ways:

- If there is enough space available in the system to buffer all the data, **send()** completes successfully, having written out all of the data, and returns the number of bytes written.
- If there is not enough space in the buffer to write out the entire request, **send()** completes successfully, having written as much data as possible, and returns the number of bytes it was able to write.
- If there is no space in the system to buffer any of the data, **send()** completes successfully, having written no data, and returns 0.

If the **O_NDELAY** flag is cleared using **fcntl()**, non-blocking I/O is disabled. In this case, the **send()** always executes completely (blocking as necessary) and returns the number of bytes written.

Since both the **fcntl()** **O_NONBLOCK** and **O_NDELAY** flags and **FIOSNBIO** **ioctl()** request are supported, some clarification on how these features interact is necessary. If the **O_NONBLOCK** or **O_NDELAY** flag has been set, **send()** requests behave accordingly, regardless of any **FIOSNBIO** **requests.** If neither the **O_NONBLOCK** nor **O_NDELAY** flag has been set, **FIOSNBIO** requests control the behavior of **send()**. If the **O_NDELAY** flag has not been set, **FIOSNBIO** requests control the behavior of **send()**.

The default is that non-blocking I/O is not enabled.

The supported values for *flags* are zero, or **MSG_OOB** (to send out-of-band data). A **write()** call made to a socket behaves in exactly the same way as **send()** with *flags* set to zero. **MSG_OOB** is not supported for AF_UNIX sockets.

The *select*(2) call can be used to determine when it is possible to send more data.

AF_CCITT only:

Sockets of the address family AF_CCITT operate in message mode. Although they are specified as connection-based (SOCK_STREAM) sockets, the X.25 subsystem communicates via messages. They require that a connection be established with the **connect()** or **accept()** calls.

The **O_NDELAY** flag is not supported, use **FIOSNBIO** requests to control non-blocking I/O. If the available buffer space is not large enough for the entire message, and the socket is in non-blocking mode, the error EWOULDBLOCK is returned. If the amount of data in the **send()** exceeds the maximum outbound message size, EMSGSIZE is returned.

The **sendto()** call is not supported.

Each call sends either a complete or a partial X.25 message. This is controlled by the setting of More-Data-To-Follow (MDTF) bit. If the user wants to send a partial message, MDTF should be set to 1 before the **send()** call. The MDTF bit should be cleared to 0 before sending the final message fragment.

Message fragment length may range from 0 bytes up to the size of the socket's send buffer (see *af_ccitt*(7F)). The MDTF bit and multiple **send()** calls can be combined to transmit complete X.25 packet sequences (i.e., zero or more DATA packets in which the More Data bit is set, followed by one DATA packet in which the More Data bit is clear) of arbitrary length. Note that a 0-byte message is not actually sent, but may be necessary to flush a complete X.25 message if the user is controlling the MDTF bit.

Sockets of the AF_CCITT address family can send 1 byte of out-of-band data (known as **INTERRUPT** Data packet in X.25 terminology), or up to 32 bytes if the X.25 interface is configured for 1984 CCITT X.25 recommendations. INTERRUPT data packets sent in blocking mode cause the process to block until confirmation is received. INTERRUPT data packets sent with the socket in non-blocking mode do not cause the process to block; instead, an out-of-band message is queued to the socket when the INTERRUPT confirmation packet is received (see *recv*(2)).

DEPENDENCIES

UDP messages are fragmented at the IP level into Maximum Transmission Unit (MTU) sized pieces; MTU varies for different link types. These pieces, called IP fragments, can be transmitted, but IP does not guarantee delivery. Sending large messages may cause so many fragments to be created that some of them overrun a receiver's ability to receive them, and hence are dropped. If this happens, even if most of the fragments ultimately arrive at the destination, the complete message cannot be re-assembled. This affects the apparent reliability and throughput of the network, as viewed by the end-user.

Default and **maximum** buffer sizes are protocol-specific. Refer to the appropriate entries in Sections 7F and 7P for details. The buffer size can be set by calling **setsockopt()** with **SO_SNDBUF**.

AF_CCITT

If the receiving process is on a Series 700/800 HP-UX system and the connection has been set up to use the D-bit, data sent with the D-bit set is acknowledged when the receiving process has read the data. Otherwise, the acknowledgement is sent when the firmware receives it.

If the receiving process is on a Series 300/400 HP-UX system, data sent with the D-bit set is acknowledged when the data reaches the X.25 interface card, but D-bit acknowledgement from a Series 300/400 does not imply that the receiving process has read the data.

RETURN VALUE

Upon successful completion, **send()**, **sendto()**, and **sendmsg()** return the number of bytes sent. Otherwise, they return -1 and set **errno** to indicate the error.

DIAGNOSTICS

send(), **sendto()**, and **sendmsg()** fail if any of the following conditions are encountered:

[EACCES]	Process doing a **send()** of a broadcast packet does not have broadcast capability enabled for the socket. Use **setsockopt()** to enable broadcast capability.
[EBADF]	An invalid descriptor was specified.
[ENOTSOCK]	The argument *s* is not a socket.
[EFAULT]	An invalid pointer was specified in the *msg* or *to* parameter, or in the msghdr structure.
[EMSGSIZE]	A length in the msghdr structure is invalid. The socket requires that messages be sent atomically, and the size of the message to be sent made this impossible. **SOCK_DGRAM/AF_INET** and/or **SOCK_STREAM/AF_CCITT Only:** The message size exceeded the outbound buffer size.
[EWOULDBLOCK]	The socket is in non-blocking mode and the requested operation would block.
[ENOBUFS]	Insufficient network memory resources were available in the system to perform the operation.
[EINTR]	The operation was interrupted by a signal before any data were sent. (If some data was sent, **send()** returns the number of bytes sent before the signal, and EINTR is not given.)
[EINVAL]	The *len* or *tolen* parameter, or a length in the msghdr structure is invalid. A **sendto()** system call was issued on an X.25 socket, or the connection is in

its reset sequence and cannot accept data.

[EDESTADDRREQ] The *to* parameter needs to specify a destination address for the message. This is also given if the specified address contains unspecified fields (see *inet*(7F)).

[ENOTCONN] A **send()** on a socket that is not connected, or a **send()** on a socket that has not completed the connect sequence with its peer, or is no longer connected to its peer.

[EISCONN] An address was specified by *to* for a SOCK_DGRAM socket which is already connected.

[EAFNOSUPPORT] Requested address does not match the address family of this socket.

[EPIPE] and SIGPIPE signal
An attempt was made to send on a socket that was connected, but the connection has been shut down, either by the remote peer or by this side of the connection. Note that the default action for **SIGPIPE**, unless the process has established a signal handler for this signal, is to terminate the process.

[EIO] A timeout occurred.

[ENETDOWN] The interface used for the specified address is "down" (see *ifconfig*(1M)), or no interface for the specified address can be found, (SO_DONTROUTE socket option in use), or the X.25 Level 2 is down.

[EOPNOTSUPP] The **MSG_OOB** flag was specified; it is not supported for AF_UNIX sockets.

[ENETUNREACH] (LAN) All encapsulations (e.g., ether, ieee) have been turned off (see also *lanconfig*(1M), and *ifconfig*(1M)).

(X.25) The X.25 Level 2 is down. The X.25 link layer is not working (wires might be broken, or connections are loose on the interface hoods at the modem, or the modem failed, or the packet switch at the remote end lost power or failed for some reason, or electrical noise interfered with the line for an extremely long period of time).

[ECONNRESET] A connection was forcibly closed by a peer.

AUTHOR

send() was developed at the University of California, Berkeley.

SEE ALSO

lanconfig(1M), ifconfig(1M), getsockopt(2), recv(2), select(2), setsockopt(2), socket(2), af_ccitt(7F), inet(7F), socket(7), socketx25(7), tcp(7P), udp(7P), unix(7P).

NAME

setacl, fsetacl - set access control list (ACL) information

SYNOPSIS

```
#include <sys/acl.h>

int setacl(
    const char *path,
    size_t nentries,
    const struct acl_entry *acl
);

int fsetacl(
    int fildes,
    size_t nentries,
    const struct acl_entry *acl
);
```

DESCRIPTION

setacl() sets an existing file's access control list (ACL) or deletes optional entries from it. *path* points to a path name of a file.

Similarly, **fsetacl()** sets an existing file's access control list for an open file known by the file descriptor *fildes*.

The effective user ID of the process must match the owner of the file or be the super-user to set a file's ACL.

A successful call to **setacl()** deletes all of a file's previous optional ACL entries (see explanation below), if any. *nentries* indicates how many valid entries are defined in the *acl* parameter. If *nentries* is zero or greater, the new ACL is applied to the file. If any of the file's base entries (see below) is not mentioned in the new ACL, it is retained but its access mode is set to zero (no access). Hence, routine calls of **setacl()** completely define the file's ACL.

As a special case, if *nentries* is negative (that is, a value of **ACL_DELOPT** (defined in **<sys/acl.h>**), the *acl* parameter is ignored, all of the file's optional entries, if any, are deleted, and its base entries are left unaltered.

Some of the miscellaneous mode bits in the file's mode might be turned off as a consequence of calling **setacl()**. See *chmod*(2).

Access Control Lists

An ACL consists of a series of entries. Entries can be categorized in four levels of specificity:

(*u*.*g*, *mode*)	applies to user *u* in group *g*
(*u*.%, *mode*)	applies to user *u* in any group
(%.*g*, *mode*)	applies to any user in group *g*
(%.%, *mode*)	applies to any user in any group

Entries in the ACL must be unique; no two entries can have the same user ID (*uid*) and group ID (*gid*) (see below). Entries can appear in any order. The system orders them as needed for access checking.

The **<sys/acl.h>** header file defines **ACL_NSUSER** as the non-specific *uid* value and **ACL_NSGROUP** as the non-specific *gid* value represented by **%** above. If *uid* in an entry is **ACL_NSUSER**, it is a **%.***g* entry. If *gid* in an entry is **ACL_NSGROUP**, it is a *u* **.%** entry. If both *uid* and *gid* are non-specific, the file's entry is **%.%**.

The **<unistd.h>** header file defines meanings of mode bits in ACL entries (**R_OK**, **W_OK**, and **X_OK**). Irrelevant bits in mode values must be zero.

Every file's ACL has three base entries which cannot be added or deleted, but only modified. The base ACL entries are mapped directly from the file's permission bits.

(<file's owner> . ACL_NSGROUP, <file's owner mode bits>)
(ACL_NSUSER . <file's group>, <file's group mode bits>)
(ACL_NSUSER . ACL_NSGROUP, <file's other mode bits>)

In addition, up to 13 optional ACL entries can be set to restrict or grant access to a file.

Altering a base ACL entry's modes with `setacl()` changes the file's corresponding permission bits. The permission bits can be altered also by using `chmod()` (see *chmod*(2)) and read using `stat()` (see *stat*(2)).

The number of entries allowed per file (see `NACLENTRIES` in `<sys/acl.h>`) is small for space and performance reasons. User groups should be created as needed for access control purposes. Since ordinary users cannot create groups, their ability to control file access with ACLs might be somewhat limited.

RETURN VALUE

Upon successful completion, `setacl()` and `fsetacl()` return a value of zero. If an error occurs, they return -1, the file's ACL is not modified, and `errno` is set to indicate the error.

ERRORS

`setacl()` and `fsetacl()` fail if any of the following conditions are encountered:

[ENOTDIR]	A component of the *path* prefix is not a directory.
[ENOENT]	The named file does not exist (for example, *path* is null or a component of *path* does not exist).
[EBADF]	*fildes* is not a valid file descriptor.
[EACCES]	A component of the *path* prefix denies search permission.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not super-user.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*path* or *acl* points outside the allocated address space of the process, or *acl* is not as large as indicated by *nentries*.
[EINVAL]	There is a redundant entry in the ACL, or *acl* contains an invalid *uid*, *gid*, or *mode* value.
[E2BIG]	An attempt was made to set an ACL with more than `NACLENTRIES` entries.
[EOPNOTSUPP]	`setacl()` is not supported on remote files by some networking services.
[ENOSPC]	Not enough space on the file system.
[ENFILE]	System file table is full.
[ENAMETOOLONG]	The length of *path* exceeds `PATH_MAX` bytes, or the length of a component of *path* exceeds `NAME_MAX` bytes while `_POSIX_NO_TRUNC` is in effect.
[ELOOP]	Too many symbolic links were encountered in translating the *path* name.
[EDQUOT]	User's disk quota block or inode limit has been reached for this file system.

EXAMPLES

The following code fragment defines and sets an ACL on file `../shared` which allows the file's owner to read, write, and execute or search the file, and allows user 103, group 204 to read the file.

```
#include <unistd.h>
#include <sys/stat.h>
#include <sys/acl.h>

char *filename = "../shared";
struct acl_entry acl [2];
struct stat statbuf;

if (stat (filename, & statbuf) <
    error (...);

acl [0] . uid  = statbuf . st_uid;   /* file owner */
acl [0] . gid  = ACL_NSGROUP;
acl [0] . mode = R_OK | W_OK | X_OK;

acl [1] . uid  = 103;
acl [1] . gid  = 204;
```

```
acl [1] . mode = R_OK;
if (setacl (filename, 2, acl))
     error (...);
```

The following call deletes all optional ACL entries from `file1`:

```
setacl ("file1", ACL_DELOPT, (struct acl_entry *) 0);
```

DEPENDENCIES

NFS

`setacl()` and `fsetacl()` are not supported on remote files.

AUTHOR

`setacl()` and `fsetacl()` were developed by HP.

SEE ALSO

access(2), chmod(2), getaccess(2), getacl(2), stat(2), acl(5), unistd(5).

NAME

setaudid - set the audit ID (aid) for the current process

SYNOPSIS

```
#include <sys/audit.h>

int setaudid(aid_t audid);
```

DESCRIPTION

`setaudid()` sets the audit ID (*aid*) for the current process. This call is restricted to the super-user.

RETURN VALUE

Upon successful completion, `setaudid()` returns a value of 0; otherwise, it returns -1 and sets `errno` to indicate the error.

ERRORS

`setaudid()` fails if any of the following conditions are encountered:

[EPERM]	The caller is not a superuser.
[EINVAL]	The audit ID (audid) is invalid.

AUTHOR

`setaudid()` was developed by HP.

SEE ALSO

getaudid(2).

NAME

setaudproc - controls process level auditing for the current process and its decendents

SYNOPSIS

```
#include <sys/audit.h>

int setaudproc(int aflag);
```

DESCRIPTION

`setaudproc()` controls process level auditing for the current process and its decendents. It accomplishes this by setting or clearing the `u_audproc` flag in the `u` area of the calling process. When this flag is set, the system audits the process; when it is cleared, the process is not audited. This call is restricted to superusers.

One of the following *aflags* must be used:

AUD_PROC	Audit the calling process and its decendents.
AUD_CLEAR	Do not audit the calling process and its decendents.

The `u_audproc` flag is inherited by the descendents of a process. consequently, the effect of a call to `setaudproc()` is not limited to the current process, but propagates to all its decendents as well. For example, if `setaudproc()` is called with the **AUD_PROC** flag, all subsequent audited system calls in the current process *and its decendents* are audited until `setaudproc()` is called with the **AUD_CLEAR** flag.

Further, `setaudproc()` performs its action regardless of whether the user executing the process has been selected to be audited or not. For example, if `setaudproc()` is called with the **AUD_PROC** (or the **AUD_CLEAR**) flag, all subsequent audited system calls will be audited (or not audited), regardless of whether the user executing the process has been selected for auditing or not.

Due to these features, `setaudproc()` should not be used in most self-auditing applications. `audswitch()` should be used (see *audswitch*(2)) when the objective is to suspend auditing within a process without affecting its decendents or overriding the user selection aspect of the auditing system.

RETURN VALUE

Upon successful completion, `setaudproc()` returns 0; otherwise, it returns -1 and sets `errno` to indicate the error.

AUTHOR

`setaudproc()` was developed by HP.

SEE ALSO

getaudproc(2), audswitch(2), audusr(1M), audevent(1M), audit(5).

NAME

setevent - set current events and system calls which are to be audited

SYNOPSIS

```
#include <sys/audit.h>

int setevent(
    const struct aud_type a_syscall[],
    const struct aud_event_tbl a_event[]
);
```

DESCRIPTION

`setevent()` sets the events and system calls to be audited. The event and system call settings in the tables pointed to by *a_syscall* and *a_event* become the current settings. This call is restricted to the super-user.

RETURN VALUE

Upon successful completion, `setevent()` returns 0; otherwise, it returns -1 and sets `errno` to indicate the error.

ERRORS

`setevent()` fails if the following condition is encountered:

[EPERM] The caller is not super-user.

AUTHOR

`setevent()` was developed by HP.

SEE ALSO

getevent(2), audevent(1M).

NAME

setgroups - set group access list

SYNOPSIS

```
#include <unistd.h>

int setgroups(int ngroups, const gid_t *gidset);
```

DESCRIPTION

`setgroups()` sets the group access list of the current user process according to the array *gidset*. The parameter *ngroups* indicates the number of entries in the array and must be no more than NGROUPS, as defined in `<sys/param.h>`.

Only super-user can set new groups by adding to the group access list of the current user process; any user can delete groups from it.

RETURN VALUE

Upon successful completion, `setgroups()` returns 0; otherwise it returns -1 and sets `errno` to indicate the error.

ERRORS

`setgroups()` fails if any of the following conditions are encountered:

[EPERM]	The caller is not super-user and has attempted to set new groups.
[EFAULT]	The address specified for *gidset* is outside the process address space. The reliable detection of this error is implementation dependent.
[EINVAL]	*ngroups* is greater than NGROUPS or not positive.
[EINVAL]	An entry in *gidset* is not a valid group ID.

AUTHOR

`setgroups()` was developed by the University of California, Berkeley.

SEE ALSO

getgroups(2), initgroups(3C)

STANDARDS CONFORMANCE

`setgroups()`: AES

NAME

sethostname - set name of host cpu

SYNOPSIS

```
#include <unistd.h>

int sethostname(const char *name, size_t namelen);
```

DESCRIPTION

sethostname() sets the name of the host processor to *name*, which has a length of *namelen* characters. **sethostname()** is normally executed by **hostname** (see *hostname*(1)) in the **/etc/rc** script at system boot time. Host names are limited to **MAXHOSTNAMELEN** characters, as defined in **<sys/param.h>**.

RETURN VALUE

Upon successful completion, **sethostname()** returns 0; otherwise it returns -1 and sets **errno** to indicate the error.

ERRORS

sethostname() fails if any of the following conditions are encountered:

[EPERM]	It is not executed by a user having appropriate privileges.
[EFAULT]	*name* points to an illegal address. The reliable detection of this error is implementation dependent.

AUTHOR

sethostname() was developed by the University of California, Berkeley.

SEE ALSO

hostname(1), uname(1), gethostname(2), uname(2), privilege(5).

NAME

setpgid, setpgrp2 - set process group ID for job control

SYNOPSIS

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);

int setpgrp2(pid_t pid, pid_t pgid);
```

DESCRIPTION

`setpgid()` or `setpgrp2()` causes the process specified by *pid* to join an existing process group or create a new process group within the session of the calling process. The process group ID of the process whose process ID is *pid* is set to *pgid*. If *pid* is zero, the process ID of the calling process is used. If *pgid* is zero, the process ID of the indicated process is used. The process group ID of a session leader does not change.

`setpgrp2()` is provided for backward compatibility only.

RETURN VALUE

Upon successful completion, `setpgid()` and `setpgrp2()` return zero; otherwise, they return -1 and set *errno* to indicate the error.

ERRORS

`setpgid()` and `setpgrp2()` fail and no change occurs if any of the following conditions are encountered:

[EACCES] The value of *pid* matches the process ID of a child process of the calling process and the child process has successfully executed one of the *exec*(2) functions.

[EINVAL] The value of *pgid* is less than zero or is outside the range of valid process group ID values.

[EPERM] The process indicated by *pid* is a session leader.

[EPERM] The value of *pid* is valid but matches the process ID of a child process of the calling process, and the child process is not in the same session as the calling process.

[EPERM] The value of *pgid* does not match the process ID of the process indicated by *pid* and there is no process with a process group ID that matches the value of *pgid* in the same session as the calling process.

[ESRCH] The value of *pid* does not match the process ID of the calling process or of a child process of the calling process.

AUTHOR

`setpgid()` and `setpgrp2()` were developed by HP and the University of California, Berkeley.

SEE ALSO

bsdproc(2), exec(2), exit(2), fork(2), getpid(2), kill(2), setsid(2), signal(2), termio(7).

STANDARDS CONFORMANCE

`setpgid()`: AES, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

setresuid, setresgid - set real, effective, and saved user and group IDs

SYNOPSIS

```
#include <unistd.h>

int setresuid(uid_t ruid, uid_t euid, uid_t suid);

int setresgid(gid_t rgid, gid_t egid, gid_t sgid);
```

DESCRIPTION

setresuid() sets the real, effective and/or saved user ID of the calling process.

If the current real, effective or saved user ID is equal to that of a user with having appropriate privileges, **setresuid()** sets the real, effective and saved user IDs to *ruid*, *euid*, and *suid*, respectively. Otherwise, **setresuid()** only sets the real, effective, and saved user IDs if *ruid*, *euid*, and *suid* each match at least one of the current real, effective, or saved user IDs.

If *ruid*, *euid*, or *suid* is **-1**, **setresuid()** leaves the current real, effective or saved user ID unchanged.

setresgid() sets the real, effective and/or saved group ID of the calling process.

If the current real, effective or saved user ID is equal to that of a user having appropriate privileges, **setresgid()** sets the real, effective, and saved group IDs to *rgid*, *egid*, and *sgid*, respectively. Otherwise, **setresgid()** only sets the real, effective and saved group IDs if *rgid*, *egid*, and *sgid* each match at least one of the current real, effective or saved group IDs.

If *rgid*, *egid*, or *sgid* is **-1**, **setresgid()** leaves the current real, effective or saved group ID unchanged.

RETURN VALUE

Upon successful completion, **setresuid()** and **setresgid()** return 0; otherwise, they return -1 and set **errno** to indicate the error.

ERRORS

setresuid() and **setresgid()** fail if any of the following conditions are encountered:

[EINVAL] *ruid*, *euid*, or *suid* (*rgid*, *egid*, or *sgid*) is not a valid user (group) ID.

[EPERM] None of the conditions above are met.

AUTHOR

setresuid() and **setresgid()** were developed by HP.

SEE ALSO

exec(2), getuid(2), setuid(2).

NAME

setsid, setpgrp - create session and set process group ID

SYNOPSIS

```
#include <unistd.h>

pid_t setsid(void);

pid_t setpgrp(void);
```

DESCRIPTION

If the calling process is not a process group leader, `setsid()` or `setprgp()` creates a new session. The calling process becomes the session leader of this new session, becomes the process group leader of a new process group, and has no controlling terminal. The process group ID of the calling process is set equal to the process ID of the calling process. The calling process is the only process in the new process group, and the only process in the new session.

`setprgp()` is provided for backward compatibility only.

RETURN VALUE

`setprgp()` returns the value of the process group ID of the calling process.

Upon successful completion, `setsid()` returns the value of the new process group ID of the calling process. Otherwise, a value of -1 is returned, and `errno` is set to indicate the error.

ERRORS

No change occurs if any of the following conditions are encountered. In addition, `setsid()` fails when any of the following conditions occur:

[EPERM] The calling process is already a process group leader.

[EPERM] The process group ID of a process other than the calling process matches the process ID of the calling process.

AUTHOR

`setprgp()` and `setsid()` were developed by HP and AT&T.

SEE ALSO

exec(2), exit(2), fork(2), getpid(2), kill(2), setpgid(2), signal(2), termio(7).

STANDARDS CONFORMANCE

`setsid()`: AES, XPG3, XPG4, FIPS 151-2, POSIX.1

`setpgrp()`: SVID2, XPG2

NAME

setuid, setgid - set user and group IDs

SYNOPSIS

```
#include <unistd.h>

int setuid(uid_t uid);

int setgid(gid_t gid);
```

DESCRIPTION

`setuid()` sets the real-user-ID (*ruid*),*effective-user*-ID (*euid*), and/or saved-user-ID (*suid*) of the calling process. The super-user's *euid* is zero. The following conditions govern *setuid*'s behavior:

- If the *euid* is zero, `setuid()` sets the *ruid*, *euid*, and *suid* to *uid*.
- If the *euid* is not zero, but the argument *uid* is equal to the *ruid* or the *suid*, `setuid()` sets the *euid* to *uid*; the *ruid* and *suid* remain unchanged. (If a set-user-ID program is not running as super-user, it can change its *euid* to match its *ruid* and reset itself to the previous *euid* value.)
- If *euid* is not zero, but the argument *uid* is equal to the *euid*, and the calling process is a member of a group that has the `PRIV_SETRUGID` privilege (see *privgrp*(4)), `setuid()` sets the *ruid* to *uid*; the *euid* and *suid* remain unchanged.

`setgid()` sets the real-group-ID (*rgid*), effective-group-ID (*egid*), and/or saved-group-ID (*sgid*) of the calling process. The following conditions govern `setgid()`'s behavior:

- If *euid* is zero, `setgid()` sets the *rgid* and *egid* to *gid*.
- If *euid* is not zero, but the argument *gid* is equal to the *rgid* or the *sgid*, `setgid()` sets the *egid* to *gid*; the *rgid* and *sgid* remain unchanged.
- If *euid* is not zero, but the argument *gid* is equal to the *egid*, and the calling process is a member of a group that has the `PRIV_SETRUGID` privilege (see *privgrp*(4)), `setgid()` sets the *rgid* to *gid*; the *egid* and *sgid* remain unchanged.

RETURN VALUE

Upon successful completion, `setuid()` and `setgid()` returned 0; otherwise, they return -1 and set `errno` to indicate the error.

ERRORS

`setuid()` and `setgid()` fail and return `-1` if any of the following conditions are encountered:

[EPERM]	None of the conditions above are met.
[EINVAL]	*uid* (*gid*) is not a valid user (group) ID.

WARNINGS

It is recommended that the `PRIV_SETRUGID` capability be avoided, as it is provided for backward compatibility. This feature may be modified or dropped from future HP-UX releases. When changing the real user ID and real group ID, use of `setresuid()` and `setresgid()` (see *setresuid*(2)) are recommended instead.

AUTHOR

`setuid()` was developed by AT&T, the University of California, Berkeley, and HP.

`setgid()` was developed by AT&T.

SEE ALSO

exec(2), getprivgrp(2), getuid(2), setresuid(2) privgrp(4).

STANDARDS CONFORMANCE

`setuid()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

`setgid()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

shmctl - shared memory control operations

SYNOPSIS

```
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

DESCRIPTION

shmctl() provides a variety of shared memory control operations as specified by *cmd*. The following *cmd*s are available:

IPC_STAT Place the current value of each member of the data structure associated with *shmid* into the structure pointed to by *buf*. The contents of this structure are defined in the *glossary*.

IPC_SET Set the value of the following members of the data structure associated with *shmid* to the corresponding value found in the structure pointed to by *buf*:

```
shm_perm.uid
shm_perm.gid
shm_perm.mode    /* only low 9 bits */
```

This *cmd* can only be executed by a process that has an effective user ID equal to either that of a user having appropriate privileges or to the value of either **shm_perm.uid** or **shm_perm.cuid** in the data structure associated with *shmid*.

IPC_RMID
Remove the shared memory identifier specified by *shmid* from the system and destroy the shared memory segment and data structure associated with it. If the segment is attached to one or more processes, then the segment key is changed to **IPC_PRIVATE** and the segment is marked removed. The segment disappears when the last attached process detaches it. This *cmd* can only be executed by a process that has an effective user ID equal to either that of a user with appropriate privileges or to the value of either **shm_perm.uid** or **shm_perm.cuid** in the data structure associated with *shmid*.

SHM_LOCK
Lock the shared memory segment specified by *shmid* in memory. This *cmd* can only be executed by a process that either has an effective user ID equal to that of a user having appropriate privileges or has an effective user ID equal to the value of either **shm_perm.uid** or **shm_perm.cuid** in the data structure associated with *shmid* and has **PRIV_MLOCK** privilege (see **setprivgrp()** description, *getprivgrp*(2)).

SHM_UNLOCK
Unlock the shared memory segment specified by *shmid*. This *cmd* can only be executed by a process that either has an effective user ID equal to a user having appropriate privileges or has an effective user ID equal to the value of either **shm_perm.uid** or **shm_perm.cuid** in the data structure associated with *shmid* and has **PRIV_MLOCK** privilege (see **setprivgrp()** description, *getprivgrp*(2)).

RETURN VALUE

shmctl() returns a value of 0 upon successful completion; otherwise, a value of -1 is returned and **errno** is set to indicate the error.

ERRORS

shmctl() fails if any of the following conditions are encountered (see DEPENDENCIES):

[EINVAL] *shmid* is not a valid shared memory identifier.

[EINVAL] *cmd* is not a valid command.

[EACCES] *cmd* is equal to **IPC_STAT** and Read operation permission is denied to the calling process (see *shared memory operation permissions* in *glossary*(9)).

[EPERM] *cmd* is equal to **IPC_RMID**, **IPC_SET**, **SHM_LOCK**, or **SHM_UNLOCK** and the effective user ID of the calling process is not equal to that of a user having appropriate privileges and it is not equal to the value of either **shm_perm.uid** or **shm_perm.cuid** in the data structure associated with *shmid*.

[EPERM] *cmd* is equal to **SHM_LOCK** or **SHM_UNLOCK** and the effective user ID of the calling process is not equal to that of a user having appropriate privileges and the calling process does not have **PRIV_MLOCK** privilege (see **setprivgrp()** description, *getprivgrp*(2)).

[EINVAL] *cmd* is equal to **SHM_UNLOCK** and the shared-memory segment specified by *shmid* is not locked in memory.

[EFAULT] *buf* points to an illegal address. The reliable detection of this error is implementation dependent.

[ENOMEM] *cmd* is equal to **SHM_LOCK** and there is not sufficient lockable memory to fill the request.

EXAMPLES

The following call to **shmctl()** locks in memory the shared memory segment represented by *myshmid*. This example assumes the process has a valid shmid, which can be obtained by calling *shmget*(2).

```
shmctl (myshmid, SHM_LOCK, 0);
```

The following call to **shmctl()** removes the shared memory segment represented by *myshmid*. This example assumes the process has a valid shmid, which can be obtained by calling **shmget()** (see *shmget*(2).

```
shmctl (myshmid, IPC_RMID, 0);
```

DEPENDENCIES

Series 300/400

An additional error condition can occur on Series 300/400 systems:

[EACCES] *shmid* is the id of a shared memory segment currently being used by the system to implement other features (see *graphics*(7) and *iomap*(7)).

AUTHOR

shmctl() was developed by AT&T and HP.

SEE ALSO

ipcrm(1), ipcs(1), shmget(2), shmop(2), stdipc(3C).

STANDARDS CONFORMANCE

shmctl(): SVID2, XPG2, XPG3, XPG4

NAME

shmget - get shared memory segment

SYNOPSIS

```
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);
```

DESCRIPTION

shmget() returns the shared memory identifier associated with *key*.

A shared memory identifier and associated data structure and shared memory segment of size *size* bytes (see *glossary*(9)) are created for *key* if one of the following is true:

- *key* is equal to **IPC_PRIVATE**. This call creates a new identifier, subject to available resources. The identifier will never be returned by another call to **shmget()** until it has been released by a call to **shmctl()**. The identifier should be used among the calling process and its descendents; however, it is not a requirement. The resource can be accessed by any process having the proper permissions.
- *key* does not already have a shared memory identifier associated with it, and (*shmflg* & **IPC_CREAT**) is "true".

Upon creation, the data structure associated with the new shared memory identifier is initialized as follows:

- **shm_perm.cuid**, **shm_perm.uid**, **shm_perm.cgid**, and **shm_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.
- **shm_perm.cuid**, The low-order 9 bits of **shm_perm.mode** are set equal to the low-order 9 bits of *shmflg*. **shm_segsz** is set equal to the value of *size*.
- **shm_lpid**, **shm_nattch**, **shm_atime**, and **shm_dtime** are set equal to 0.
- **shm_ctime** is set equal to the current time.

EXAMPLES

The following call to **shmget()** returns a unique shmid for the newly created shared memory segment of 4096 bytes:

```
int myshmid;

myshmid = shmget (IPC_PRIVATE, 4096, 0600);
```

RETURN VALUE

Upon successful completion, a non-negative integer, namely a shared memory identifier is returned. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

ERRORS

shmget() fails if any of the following conditions are encountered:

[EINVAL]	*size* is less than the system-imposed minimum or greater than the system-imposed maximum.
[EACCES]	A shared memory identifier exists for *key* but operation permission (see *glossary*(9)) as specified by the low-order 9 bits of *shmflg* would not be granted.
[EINVAL]	A shared memory identifier exists for *key* but the size of the segment associated with it is less than *size* and *size* is not equal to zero.
[ENOENT]	A shared memory identifier does not exist for *key* and (*shmflg* & **IPC_CREAT**) is "false".
[ENOSPC]	A shared memory identifier is to be created but the system-imposed limit on the maximum number of allowed shared memory identifiers system wide would be exceeded.
[ENOMEM]	A shared memory identifier and associated shared memory segment are to be created, but the amount of available physical memory is not sufficient to fill the request.
[EEXIST]	A shared memory identifier exists for *key* but ((*shmflg* & **IPC_CREAT**) && (*shmflg* & **IPC_EXCL**)) is "true".

SEE ALSO
ipcrm(1), ipcs(1), shmctl(2), shmop(2), stdipc(3C).

STANDARDS CONFORMANCE
`shmget()`: SVID2, XPG2, XPG3, XPG4

NAME

shmat, shmdt - shared memory operations

SYNOPSIS

```
#include <sys/shm.h>

char *shmat(int shmid, void *shmaddr, int shmflg);

int shmdt(void *shmaddr);
```

DESCRIPTION

`shmat()` attaches the shared memory segment associated with the shared memory identifier specified by *shmid* to the data segment of the calling process.

Series 700/800 Systems

If the shared memory segment is not already attached, *shmaddr* must be specified as zero and the segment is attached at a location selected by the operating system. That location is identical in all processes accessing that shared memory object.

If the shared memory segment is already attached, a non-zero value of *shmaddr* is accepted, provided the specified address is identical to the current attach address of the segment.

Series 300/400 Systems

shmaddr can be specified as a non-zero value as a machine-dependent extension (see DEPENDENCIES below). However, those systems do not necessarily guarantee that a given shared memory object appears at the same address in all processes that access it, unless the user specifies an address.

The segment is attached for reading if (*shmflg* & `SHM_RDONLY`) is "true"; otherwise it is attached for reading and writing. It is not possible to attach a segment for write only.

`shmdt()` detaches from the calling process's data segment the shared memory segment located at the address specified by *shmaddr*.

RETURN VALUE

Upon successful completion, the return value is as follows:

`shmat()` returns the data segment start address of the attached shared memory segment.

`shmdt()` returns a value of 0; otherwise, a value of -1 is returned and `errno` is set to indicate the error.

ERRORS

`shmat()` fails and does not attach the shared memory segment if any of the following conditions are encountered (see DEPENDENCIES):

[EINVAL]	*shmid* is not a valid shared memory identifier.
[EACCES]	Operation permission is denied to the calling process.
[ENOMEM]	The available data space is not large enough to accommodate the shared memory segment.
[EINVAL]	*shmaddr* is not zero and the machine does not permit non-zero values or *shmaddr* is not equal to the current attach location for the shared memory segment.
[EMFILE]	The number of shared memory segments attached to the calling process exceed the system-imposed limit.

`shmdt()` fails and returns `-1` if the following condition is encountered:

[EINVAL]	*shmaddr* is not the data segment start address of a shared memory segment.

EXAMPLES

The following call to *shmat* attaches the shared memory segment to the process. This example assumes the process has a valid *shmid*, which can be obtained by calling *shmget*(2).

```
char *shmptr, *shmat();
shmptr = shmat(myshmid, (char *)0, 0);
```

The following call to `shmdt()` then detaches the shared memory segment.

```
shmdt (shmptr);
```

DEPENDENCIES

Series 300/400

shmaddr can be non-zero, and if it is, the segment is attached at the address specified by one of the following criteria:

If *shmaddr* is equal to zero, the segment is attached at the first available address as selected by the system. The selected value varies for each process accessing that shared memory object.

If *shmaddr* is not equal to zero and (*shmflg* & **SHM_RND**) is "true", the segment is attached at the address given by (*shmaddr* - (*shmaddr* % **SHMLBA**)). The character % is the C language modulus operator.

If *shmaddr* is not equal to zero and (*shmflg* & **SHM_RND**) is "false", the segment is attached at the address given by *shmaddr*.

This form of **shmat()** fails and does not attach the shared memory segment if any of the following conditions are encountered:

[EACCES] *shmid* is the ID of a shared memory segment currently being used by the system to implement other features (see *graphics*(7) and *iomap*(7)).

[EINVAL] *shmaddr* is not equal to zero, and the value of (*shmaddr* - (*shmaddr* % **SHMLBA**)) is an illegal address.

[EINVAL] *shmaddr* is not equal to zero, (*shmflg* & **SHM_RND**) is "false", and the value of *shmaddr* is an illegal address.

[ENOMEM] The calling process is locked (see *plock*(2)) and there is not sufficient lockable memory to support the process-related data structure overhead.

Series 700/800

shmat() fails and returns -1 if the following is encountered:

[EINVAL] The calling process is already attached to *shmid*.

SEE ALSO

ipcs(1), exec(2), exit(2), fork(2), shmctl(2), shmget(2), stdipc(3C).

STANDARDS CONFORMANCE

shmat(): SVID2 [Series 300/400 only], XPG2, XPG3, XPG4

shmdt(): SVID2, XPG2, XPG3, XPG4

NAME

shutdown - shut down a socket

SYNOPSIS

```
int shutdown(int s, int how);
```

DESCRIPTION

The **shutdown()** system call is used to shut down a socket. In the case of a full-duplex connection, **shutdown()** can be used to either partially or fully shut down the socket, depending upon the value of *how*:

how	**Interpretation**
0	Further receives are disallowed
1	Further sends are disallowed
2	Further sends and receives are disallowed

The *s* parameter is a socket descriptor for the socket to be shut down.

Once the socket has been shut down for receives, all further **recv()** calls return an end-of-file condition. A socket that has been shut down for sending causes further **send()** calls to return an EPIPE error and send the **SIGPIPE** signal. After a socket has been fully shut down, operations other than **recv()** and **send()** return appropriate errors, and the only other thing that can be done to the socket is a **close()**.

Multiple shutdowns on a connected socket and shutdowns on a socket that is not connected might not return errors.

A **shutdown()** on a connectionless socket, such as **SOCK_DGRAM** , only marks the socket as unable to do further **send()** or **recv()** calls, depending upon *how*. Once this type of socket has been disabled for both sending and receiving data, it becomes fully shut down. For **SOCK_STREAM** sockets, if *how* is **1 or 2**, the connection begins to be closed gracefully in addition to the normal actions. However, the **shutdown()** call does not wait for the completion of the graceful disconnection. The disconnection is complete when both sides of the connection have done a **shutdown()** with *how* equal to **1 or 2**. Once the connection has been completely terminated, the socket becomes fully shut down. The **SO_LINGER** option (see *socket*(2)) does not have any meaning for the **shutdown()** call, but does for the **close()** call. For more information on how the **close()** call interacts with sockets, see *socket*(2).

If a **shutdown()** is performed on a **SOCK_STREAM** socket that has a **listen()** pending on it, that socket becomes fully shut down when *how* = 1.

AF_CCITT only:

The *how* parameter behaves differently if the socket is of the the **AF_CCITT** address family. If *how* is set to **0** the specified socket can no longer receive data. The SVC is not cleared and remains intact. However, if data is subsequently received on the SVC, it is cleared. The connection is not completely down until either side executes a **close()** or **shutdown()** with *how* set to **1 or 2**.

If *how* is set to **1** or **2**, the SVC can no longer send or receive data and the SVC is cleared. The socket's resources are maintained so that data arriving prior to the **shutdown()** call can still be read.

RETURN VALUE

Upon successful completion, **shutdown()** returns 0; otherwise it returns -1 and **errno** is set to indicate the error.

ERRORS

shutdown() fails if any of the following conditions are encountered:

[EBADF]	*s* is not a valid descriptor.
[ENOTSOCK]	*s* is a file, not a socket.
[EINVAL]	The specified socket is not connected.

AUTHOR

shutdown() was developed by the University of California, Berkeley.

SEE ALSO

close(2), connect(2), socket(2).

NAME

sigaction - examine and change signal action

SYNOPSIS

```
#include <signal.h>

int sigaction (
    int sig,
    const struct sigaction *act,
    struct sigaction *oact
);
```

DESCRIPTION

`sigaction()` allows the calling process to examine and specify the action to be taken on delivery of a specific signal. The argument *sig* specifies the signal; acceptable values are defined in `<signal.h>`. More details on the semantics of specific signals can be found in the *signal*(5) manual entry.

The `sigaction` structure and type `sigset_t` are defined in `<signal.h>`.

act and *oact* are pointers to `sigaction` structures that include the following elements:

```
void        (*sa_handler)();
sigset_t    sa_mask;
int         sa_flags;
```

Unless it is a null pointer, the argument *act* points to a structure specifying the action to be taken when delivering the specified signal. If the argument *oact* is not a null pointer, the action previously associated with the signal is stored in the location pointed to by *oact*. If the argument *act* is a null pointer, signal handling is unchanged; thus `sigaction()` can be used to inquire about the current handling of a given signal.

The *sa_handler* member of the `sigaction` structure is assigned one of three values: `SIG_DFL`, `SIG_IGN`, or a *function address*. The actions prescribed by these values are as follows:

`SIG_DFL` Execute default action for signal.
Upon receipt of the signal *sig*, the default action (specified on *signal*(5)) is performed. The default action for most signals is to terminate the process.

A pending signal is discarded (whether or not it is blocked) if `sigaction()` is set to `SIG_DFL` for a pending signal whose default action is to ignore the signal (as in the case of `SIGCHLD`).

`SIG_IGN` Ignore the signal.
Setting a signal action to `SIG_IGN` causes a pending signal to be discarded, whether or not it is blocked.

The `SIGKILL` and `SIGSTOP` signals cannot be ignored.

function address Catch the signal.
Upon receipt of the signal *sig*, the receiving process executes the signal-catching function pointed to by *sa_handler*. The signal-catching function is entered as a C-language function call. Details on the arguments passed to this function can be found in the *signal*(5) manual entry.

The signals `SIGKILL` and `SIGSTOP` cannot be caught.

When a signal is caught by a signal-catching function installed by *sigaction*, a new mask is calculated and installed for the duration of the signal-catching function, or until a call is made to `sigprocmask()` or `sigsuspend()` (see *sigprocmask*(2) and *sigsuspend*(2)). This mask is formed by taking the union of the current signal mask, the signal to be delivered, and unless the `SA_RESETHAND` flag is set (see below), the signal mask specified in the *sa_mask* field of the `sigaction` structure associated with the signal being delivered. If and when the signal-catching function returns normally, the original signal mask is restored.

Once an action is installed for a specific signal, it remains installed until another action is explicitly requested, or until one of the *exec*(2) functions is called.

If the previous action for *sig* was established by *signal*(2), the values of the fields returned in the structure pointed to by *oact* are unspecified; in particular, *oact->sa_handler* is not necessarily the same value passed to **signal()**. However, if a pointer to the same structure or a copy thereof is passed to a subsequent call to **sigaction()** via the *act* argument, handling of the signal is reinstated as if the original call to **signal()** were repeated.

The set of signals specified by the *sa_mask* field of the **sigaction** structure pointed to by the *act* argument cannot block the **SIGKILL** or **SIGSTOP** signal. This is enforced by the system without causing an error to be indicated.

The *sa_flags* field in the **sigaction** structure can be used to modify the behavior of the specified signal. The following flag bits, defined in the <**signal.h**> header, can be set in *sa_flags:*

SA_NOCLDSTOP Do not generate **SIGCHLD** when untraced children stop (see *ptrace*(2)).

SA_ONSTACK Use the space reserved by **sigspace()** for signal processing.

SA_RESETHAND Use the semantics of **signal()**. The signal mask specified by the *sa_mask* field is not used when setting up the effective signal mask for the signal handler. If the signal is not one of those marked "not reset when caught" (see *signal*(5)), the default action for the signal is reinstated when the signal is caught, prior to entering the signal-catching function. The "not reset when caught" distinction is insignificant when **sigaction()** is called and **SA_RESETHAND** is not set.

RETURN VALUE

Upon successful completion, **sigaction()** returns 0; otherwise it returns -1 and sets **errno** to indicate the error.

ERRORS

sigaction() fails and no new signal-catching function is installed if any of the following conditions is encountered:

[EINVAL] The value of the *sig* argument is not a valid signal number, or an attempt is made to supply an action other than **SIG_DFL** for the **SIGKILL** or **SIGSTOP** signal.

[EFAULT] *act* or *oact* points to an invalid address. The reliable detection of this error is implementation dependent.

AUTHOR

sigaction() was derived from the IEEE POSIX 1003.1-1988 Standard.

SEE ALSO

ptrace(2), sigprocmask(2), sigpending(2), sigspace(2), sigsuspend(2), sigsetops(3C), signal(5).

STANDARDS CONFORMANCE

sigaction(): AES, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

sigblock - block signals

SYNOPSIS

```
#include <signal.h>

long sigblock(long mask);
```

DESCRIPTION

`sigblock()` causes the signals specified in *mask* to be added to the set of signals currently being blocked from delivery. Signal *i* is blocked if the *i*-th bit in *mask* is `1`, as specified with the macro `sigmask(`*i*`)`.

It is not possible to block signals that cannot be ignored, as documented in *signal*(5); this restriction is silently imposed by the system.

Use `sigsetmask()` to set the mask absolutely (see *sigsetmask*(2)).

RETURN VALUE

`sigblock()` returns the previous set of masked signals.

EXAMPLES

The following call to `sigblock()` adds the `SIGUSR1` and `SIGUSR2` signals to the mask of signals currently blocked for the process:

```
long oldmask;

oldmask = sigblock (sigmask (SIGUSR1) | sigmask (SIGUSR2));
```

WARNINGS

Do not use `sigblock()` in conjunction with the facilities described under *sigset*(2V).

AUTHOR

`sigblock()` was developed by the University of California, Berkeley.

SEE ALSO

kill(2), sigprocmask(2), sigsetmask(2), sigvector(2).

NAME

signal - specify what to do upon receipt of a signal

SYNOPSIS

```
#include <signal.h>
void (*signal(int sig, void (*action)(int)))(int);
```

DESCRIPTION

`signal()` allows the calling process to choose one of three ways to handle the receipt of a specific signal. *sig* specifies the signal and *action* specifies the choice.

Acceptable values for *sig* are defined in `<signal.h>`. The specific signals are described in full in the *signal*(5) manual entry.

The value of the *action* argument specifies what to do upon the receipt of signal *sig*, and should be one of the following:

`SIG_DFL` Execute the default action, which varies depending on the signal. The default action for most signals is to terminate the process (see *signal*(5)).

A pending signal is discarded (whether or not it is blocked) if *action* is set to `SIG_DFL` but the default action of the pending signal is to ignore the signal (as in the case of `SIGCLD`).

`SIG_IGN` Ignore the signal.
When `signal()` is called with *action* set to `SIG_IGN` and an instance of the signal *sig* is pending, the pending signal is discarded, whether or not it is blocked.

`SIGKILL` and `SIGSTOP` signals cannot be ignored.

address Catch the signal.
Upon receipt of signal *sig*, reset the value of *action* for the caught signal to SIG_DFL (except signals marked with "not reset when caught"; see *signal*(5)), call the signal-catching function to which *address* points, and resume executing the receiving process at the point where it was interrupted.

The signal-catching function is called with the following three parameters:

sig The signal number.

code A word of information usually provided by the hardware.

scp A pointer to the machine-dependent structure *sigcontext* defined in `<signal.h>`.

Depending on the value of *sig*, *code* can be zero and/or *scp* can be NULL. The meanings of *code* and *scp* and the conditions determining when they are other than zero or NULL are implementation dependent (see DEPENDENCIES below). It is possible for *code* to always be zero, and *scp* to always be NULL.

The pointer *scp* is valid only during the context of the signal-catching function.

The signals `SIGKILL` and `SIGSTOP` cannot be caught.

RETURN VALUE

Upon successful completion, `signal()` returns the previous value of *action* for the specified signal *sig*. Otherwise, a value of `SIG_ERR` is returned and `errno` is set to indicate the error.

ERRORS

`signal()` fails if the following is true:

[EINVAL] *sig* is an illegal signal number, or is equal to `SIGKILL` or `SIGSTOP`.

EXAMPLES

The following call to `signal()` sets up a signal-catching function for the `SIGINT` signal:

```
void myhandler();
(void) signal(SIGINT, myhandler);
```

WARNINGS

`signal()` should not be used in conjunction with the facilities described under *bsdproc*(2), *sigaction*(2), *sigset*(2V), or *sigvector*(2).

`signal()` does not detect an invalid value for *action*, and if it does not equal `SIG_DFL` or `SIG_IGN`, or point to a valid function address, subsequent receipt of the signal *sig* causes undefined results.

DEPENDENCIES

Series 300/400

The *code* word is always zero for all signals except `SIGILL` and `SIGFPE`. For `SIGILL`, *code* has the following values:

0	illegal instruction;
6	check instruction;
7	TRAPV;
8	privilege violation.

Refer to the MC6800xx processor documentation for more detailed information about the meaning of the `SIGILL` errors.

For `SIGFPE`, *code* has the following values:

0	software floating point exception;
5	integer divide-by-zero.
0x8*xxxxxx*	any value with the high-order bit set indicates an exception while using the HP98248 floating-point accelerator. The value of (*code* &~ 0x8000000) is the value of the HP98248 status register. Refer to the HP98248 documentation for more detailed information.
other	any other value indicates an exception while using the MC68881 or MC68882 floating-point coprocessor. The value of *code* is the value of the MC68881 or MC68882 status register. Refer to the MC68881 documentation for more detailed information.

Series 700/800

The structure pointer *scp* is always defined.

The *code* word is always zero for all signals except `SIGILL` and `SIGFPE`. For `SIGILL`, *code* has the following values:

8	illegal instruction trap;
9	break instruction trap;
10	privileged operation trap;
11	privileged register trap.

For `SIGFPE`, *code* has the following values:

12	overflow trap;
13	conditional trap;
14	assist exception trap;
22	assist emulation trap.

As defined by the IEEE POSIX Standard, HP-UX on Series 700/800 systems does not raise an exception on floating-point divide by zero. The result of floating-point divide by zero is infinity which can be checked by *isinf*(3M).

AUTHOR

`signal()` was developed by HP, AT&T, and the University of California, Berkeley.

SEE ALSO

kill(1), init(1M), exit(2), kill(2), lseek(2), pause(2), sigaction(2), sigvector(2), wait(2), abort(3C), setjmp(3C), signal(5).

STANDARDS CONFORMANCE

`signal()`: AES, SVID2, XPG2, XPG3, XPG4, ANSI C

NAME

sigpause - atomically release blocked signals and wait for interrupt

SYNOPSIS

```
#include <signal.h>

long sigpause(long mask);
```

DESCRIPTION

`sigpause()` blocks signals according to the value of *mask* in the same manner as *sigsetmask*(2), then atomically waits for an unmasked signal to arrive. On return `sigpause()` restores the current signal mask to the value that existed before the `sigpause()` call. When no signals are to be blocked, a value of `0L` is used for *mask*.

In normal usage, a signal is blocked using `sigblock()` (see *sigblock*(2)). To begin a critical section, variables modified on the occurrence of the signal are examined to determine that there is no work to be done, and the process pauses, awaiting work by using `sigpause()` with the mask returned by `sigblock()`.

RETURN VALUE

`sigpause()` terminates when it is interrupted by a signal. When `sigpause()` terminates, it returns -1 and sets `errno` to EINTR.

EXAMPLES

The following call to `sigpause()` waits until the calling process receives a signal:

```
sigpause (0L);
```

The following example blocks the SIGIO signal until `sigpause()` is called. When a signal is received at the `sigpause()` statement, the signal mask is restored to its value before `sigpause()` was called:

```
long savemask;
savemask = sigblock (sigmask (SIGIO));
/* critical section */
sigpause (savemask);
```

WARNINGS

Check all references to *signal*(5) for appropriateness on systems that support *sigvector*(2). *sigvector()* can affect the behavior described on this page.

Do not use `sigpause()` in conjunction with the facilities described under *sigset*(2V).

AUTHOR

`sigpause()` was developed by the University of California, Berkeley.

SEE ALSO

sigblock(2), sigsetmask(2), sigsuspend(2), sigvector(2).

NAME

sigpending - examine pending signals

SYNOPSIS

```
#include <signal.h>

int sigpending(sigset_t *set);
```

DESCRIPTION

`sigpending()` stores sets of signals that are blocked from delivery and are pending to the calling process, at the location pointed to by *set*.

RETURN VALUE

Upon successful completion, `sigpending()` returns a value of 0; otherwise it returns -1 and sets `errno` to indicate the error.

ERRORS

`sigpending()` fails if the following condition is encountered:

[EFAULT] *set* points to an invalid address. The reliable detection of this error is implementation dependent.

AUTHOR

`sigpending()` was derived from the IEEE POSIX 1003.1-1988 Standard.

SEE ALSO

sigaction(2), sigsuspend(2), sigprocmask(2), sigsetops(3C), signal(5).

STANDARDS CONFORMANCE

`sigpending()`: AES, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

sigprocmask - examine and change blocked signals

SYNOPSIS

```
#include <signal.h>

int sigprocmask(
    int how,
    const sigset_t *set,
    sigset_t *oset
);
```

DESCRIPTION

`sigprocmask()` allows the calling process to examine and/or change its signal mask.

Unless it is a null pointer, the argument *set* points to a set of signals to be used to change the currently blocked set.

The argument *how* indicates how the set is changed, and consists of one of the following values (see `<signal.h>`):

`SIG_BLOCK`	The resulting set is the union of the current set and the signal set pointed to by *set*.
`SIG_UNBLOCK`	The resulting set is the intersection of the current set and the complement of the signal set pointed to by *set*.
`SIG_SETMASK`	The resulting set is the signal set pointed to by *set*.

If the argument *oset* is not a null pointer, the previous signal mask is stored in the location pointed to by *oset*. If *set* is a null pointer, the value of the argument *how* is insignificant and the process's signal mask is unchanged; thus the call can be used to inquire about currently blocked signals.

If any pending unblocked signals remain after the call to `sigprocmask()`, at least one of those signals is delivered before the call to `sigprocmask()` returns.

It is impossible to block the `SIGKILL` or `SIGSTOP` signal. This is enforced by the system without causing an error to be indicated.

The process's signal mask is not changed if `sigprocmask()` fails for any reason.

RETURN VALUE

Upon successful completion, `sigprocmask()` returns 0; otherwise it returns -1 and sets `errno` to indicate the error.

ERRORS

`sigprocmask()` fails if any of the following conditions are encountered:

[EINVAL]	The value of the *how* argument is not equal to one of the defined values.
[EFAULT]	*set* or *oset* points to an invalid address. The reliable detection of this error is implementation dependent.

AUTHOR

`sigprocmask()` was derived from the IEEE POSIX 1003.1-1988 Standard.

SEE ALSO

sigaction(2), sigsuspend(2), sigpending(2), sigsetops(3C), signal(5).

STANDARDS CONFORMANCE

`sigprocmask()`: AES, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

sigset, sighold, sigrelse, sigignore, sigpause - signal management

SYNOPSIS

```
#include <signal.h>

void (*sigset(int sig, void (*func)(int)))(int);

int sighold(int sig);

int sigrelse(int sig);

int sigignore(int sig);

int sigpause(int sig);
```

DESCRIPTION

The system defines a set of signals that can be delivered to a process. The set of signals is defined in *signal*(5), along with the meaning and side effects of each signal. An alternate mechanism for handling these signals is defined here. The facilities described here should not be used in conjunction with the other facilities described under *signal*(2), *sigvector*(2), *sigblock*(2), *sigsetmask*(2), *sigpause*(2) and *sigspace*(2).

sigset() allows the calling process to choose one of four ways to handle the receipt of a specific signal. *sig* specifies the signal and *func* specifies the choice.

sig can be any one of the signals described under *signal*(5) except **SIGKILL** or **SIGSTOP**.

func is assigned one of four values: **SIG_DFL**, **SIG_IGN**, **SIG_HOLD**, or a *function address*. The actions prescribed by **SIG_DFL** and **SIG_IGN** are described under *signal*(5). The action prescribed by **SIG_HOLD** and *function address* are described below:

SIG_HOLD Hold signal.
The signal *sig* is held upon receipt. Any pending signal of this signal type remains held. Only one signal of each type is held.

Note: the signals **SIGKILL**, **SIGCONT**, and **SIGSTOP** cannot be held.

function address
Catch signal.
func must be a pointer to a function, the signal-catching handler, that is called when signal *sig* occurs. **sigset()** specifies that the process calls this function upon receipt of signal *sig*. Any pending signal of this type is released. This handler address is retained across calls to the other signal management functions listed here. Upon receipt of signal *sig*, the receiving process executes the signal-catching function pointed to by *func* as described under *signal*(5) with the following differences:

Before calling the signal-catching handler, the system signal action of *sig* is set to SIG_HOLD. During a normal return from the signal-catching handler, the system signal action is restored to *func* and any held signal of this type is released. If a non-local goto (*longjmp*(3C)) is taken, **sigrelse()** must be called to restore the system signal action to *func* and release any held signal of this type.

sighold() holds the signal *sig*. **sigrelse()** restores the system signal action of *sig* to that specified previously by **sigset()**. **sighold()** and **sigrelse()** are used to establish critical regions of code. **sighold()** is analogous to raising the priority level and deferring or holding a signal until the priority is lowered by **sigrelse()**.

sigignore() sets the action for signal *sig* to **SIG_IGN** (see *signal*(5)).

sigpause() suspends the calling process until it receives an unblocked signal. If the signal *sig* is held, it is released before the process pauses. **sigpause()** is useful for testing variables that are changed when a signal occurs. For example, **sighold()** should be used to block the signal first, then test the variables. If they have not changed, call **sigpause()** to wait for the signal.

These functions can be linked into a program by giving the **-lV3** option to the **ld** command (see *ld*(1)).

RETURN VALUE

Upon successful completion, **sigset()** returns the previous value of the system signal action for the specified signal *sig*. Otherwise, a value of **SIG_ERR** is returned and **errno** is set to indicate the error.

SIG_ERR is defined in <**signal.h**>.

For the other functions, a 0 value indicates that the call succeeded. A -1 return value indicates an error occurred and **errno** is set to indicate the reason.

ERRORS

sigset() fails and the system signal action for *sig* is not changed if any of the following occur:

[EFAULT] The *func* argument points to memory that is not a valid part of the process address space. Reliable detection of this error is implementation dependent.

sigset(), **sighold()**, **sigrelse()**, **sigignore()**, and **sigpause()** fail and the system signal action for *sig* is not changed if any of the following occur:

[EINVAL] *sig* is not a valid signal number.

[EINVAL] An attempt is made to ignore, hold, or supply a handler for a signal that cannot be ignored, held, or caught; see *signal*(5).

sigpause returns when the following occurs:

[EINTR] A signal was caught.

WARNINGS

These signal facilities should not be used in conjunction with *bsdproc*(2), *signal*(2), *sigvector*(2), *sigblock*(2), *sigsetmask*(2), *sigpause*(2) and *sigspace*(2).

SEE ALSO

kill(1), kill(2), signal(2), pause(2), wait(2), abort(3C), setjmp(3C), signal(5).

STANDARDS CONFORMANCE

sigset: SVID2

sighold: SVID2

sigignore: SVID2

sigpause: SVID2

sigrelse: SVID2

NAME

sigsetmask - set current signal mask

SYNOPSIS

```
#include <signal.h>

long sigsetmask(long mask);
```

DESCRIPTION

sigsetmask() sets the current signal mask (those signals that are blocked from delivery). Signal *i* is blocked if the *i*-th bit in *mask*, as specified with the macro **sigmask(*i*)**, is a 1.

It is not possible to mask signals that cannot be ignored, as documented in *signal*(5); this restriction is silently imposed by the system.

sigblock() can be used to add elements to the set of blocked signals.

RETURN VALUE

The previous set of masked signals is returned.

EXAMPLES

The following call to **sigsetmask()** causes only the **SIGUSR1** and **SIGUSR2** signals to be blocked:

```
long oldmask;

oldmask = sigsetmask (sigmask (SIGUSR1) | sigmask (SIGUSR2));
```

WARNINGS

Do not use **sigsetmask()** in conjunction with the facilities described under *sigset*(2V).

AUTHOR

sigsetmask() was developed by the University of California, Berkeley.

SEE ALSO

kill(2), sigblock(2), sigpause(2), sigprocmask(2), sigvector(2).

NAME

sigspace - assure sufficient signal stack space

SYNOPSIS

```
#include <signal.h>

size_t sigspace(size_t stacksize);
```

DESCRIPTION

`sigspace()` requests additional stack space that is guaranteed to be available for processing signals received by the calling process.

If the value of *stacksize* is positive, it specifies the size of a space, in bytes, which the system guarantees to be available when processing a signal. If the value of *stacksize* is zero, any guarantee of space is removed. If the value is negative, the guarantee is left unchanged; this can be used to interrogate the current guaranteed value.

When a signal's action indicates that its handler should use the guaranteed space (specified with a `sigaction()`, `sigvector()`, or `sigvec()` call (see *bsdproc*(2)), the system checks to see if the process is currently using that space. If the process is not currently using that space, the system arranges for that space to be available for the duration of the signal handler's execution. If that space has already been made available (due to a previous signal) no change is made. Normal stack discipline is resumed when the signal handler first using the guaranteed space is exited.

The guaranteed space is inherited by child processes resulting from a successful `fork()` system call, but the guarantee of space is removed after any `exec()` system call (see *fork*(2) and *exec*(2)).

The guaranteed space cannot be increased in size automatically, as is done for the normal stack. If the stack overflows the guaranteed space, the resulting behavior of the process is undefined.

Guaranteeing space for a stack can interfere with other memory allocation routines in an implementation-dependent manner.

During normal execution of the program the system checks for possible overflow of the stack. Guaranteeing space might cause the space available for normal execution to be reduced.

Leaving the context of a service routine abnormally, such as by `longjmp()` (see *setjmp*(3C)), removes the guarantee that the ordinary execution of the program will not extend into the guaranteed space. It might also cause the program to lose forever its ability to automatically increase the stack size, causing the program to be limited to the guaranteed space.

RETURN VALUE

Upon successful completion, `sigspace()` returns the size of the former guaranteed space. Otherwise, it returns -1 and sets `errno` to indicate the error.

ERRORS

`sigspace()` fails and the guaranteed amount of space remains unchanged if the following occurs:

[ENOMEM]	The requested space cannot be guaranteed, either because of hardware limitations or because some software-imposed limit would be exceeded.

WARNINGS

The guaranteed space is allocated using *malloc*(3C). This use might interfere with other heap management mechanisms.

Methods for calculating the required size are not well developed.

Do not use `sigspace()` in conjunction with the facilities described under *sigset*(2V).

Do not use `sigspace()` in conjunction with *sigstack*(2).

DEPENDENCIES

Series 300/400

The kernel overhead taken in the reserved space is 608 bytes on Series 300/400 systems. This overhead must be included in the requested amount. These values are subject to change in future releases.

AUTHOR

`sigspace()` was developed by HP.

SEE ALSO
sigaction(2), sigstack(2), sigvector(2), malloc(3C), setjmp(3C).

NAME

sigstack - set and/or get signal stack context

SYNOPSIS

```
#include <signal.h>

int sigstack(
    const struct sigstack *ss,
    struct sigstack *oss
);
```

DESCRIPTION

sigstack() allows the calling process to indicate to the system an area of its address space to be used for processing signals received by the process.

The correct use of **sigstack()** is hardware dependent, and therefore is not portable between different HP-UX implementations (see DEPENDENCIES below). **sigspace()** is portable between different HP-UX implementations and should be used when the application does not need to know where the signal stack is located (see *sigspace*(2)). **sigstack()** is provided for compatability with other systems that provide this functionality. Users should note that there is no guarantee that functionality similar to this is even possible on some architectures.

If the value of the *ss* argument is not a null pointer, it is assumed to point to a **struct sigstack** structure, which includes the following members:

int *ss_onstack*;	Non-zero when signal stack is in use.
void ***ss_sp**;	Signal stack pointer.

The value of the *ss_onstack* member indicates whether the process wants the system to use a signal stack when delivering signals; the value of the *ss_sp* member indicates the desired location (see DEPENDENCIES) of the signal stack area in the process's virtual address space.

If the *ss* argument is a null pointer, the current signal stack context is not changed.

If the *oss* argument is not a null pointer, it should point to a variable of type **struct sigstack**; the current signal stack context is returned in that variable. The value stored in the *ss_onstack* member tells whether the process is currently using a signal stack, and if so, the value stored in the *ss_sp* member is the current stack pointer for the stack in use.

If the *oss* argument is a null pointer, the current signal stack context is not returned.

When a signal's action indicates its handler should execute on the signal stack (specified by calling **sigaction()**, **sigvector()**, or **sigvec()** (see *bsdproc*(2))), the system checks to see if the process is currently executing on that stack. If the process is not currently executing on the signal stack, the system arranges a switch to the signal stack for the duration of the signal handler's execution.

The signal stack context is inherited by child processes resulting from a successful **fork()** system call, but the context is removed after an **exec()** system call (see *fork*(2) and *exec*(2)).

RETURN VALUE

Upon successful completion, **sigstack()** returns 0; otherwise, it returns -1 and sets **errno** to indicate the error.

ERRORS

sigstack() fails and the signal stack context remains unchanged if the following is true:

[EFAULT]	Either of *ss* or *oss* is not a null pointer and points outside the allocated address space of the process. The reliable detection of this error is implementation dependent.

WARNINGS

Do not use *sigstack*(2) in conjunction with *sigspace*(2).

User-defined signal stacks do not grow automatically, as does the normal process stack. If a signal stack overflows, the resulting behavior of the process is undefined.

Methods for calculating the required stack size are not well developed.

Leaving the context of a service routine abnormally, such as by `longjmp()` (see *setjmp*(3C)), might remove the guarantee that the ordinary execution of the program does not extend into the guaranteed space. It might also cause the program to lose forever its ability to automatically increase the stack size, causing the program to be limited to the guaranteed space.

DEPENDENCIES

Series 300/400

Stack addresses grow from high addresses to low addresses; therefore the signal stack address provided to *sigstack*(2) should point to the end of the space to be used for the signal stack. This address should be aligned to a four-byte boundary.

Series 700/800

Stack addresses grow from low addresses to high addresses; therefore the signal stack address provided to *sigstack*(2) should point to the beginning of the space to be used for the signal stack. This address should be aligned to an eight-byte boundary.

AUTHOR

`sigstack()` was developed by HP and the University of California, Berkeley.

SEE ALSO

sigspace(2), setjmp(3C).

NAME

sigsuspend - wait for a signal

SYNOPSIS

```
#include <signal.h>

int sigsuspend(const sigset_t *sigmask);
```

DESCRIPTION

sigsuspend() replaces the process's current signal mask with the set of signals pointed to by *sigmask*, then suspends the process until delivery of a signal that either executes a signal handler or terminates the process.

If the signal terminates the process, **sigsuspend()** never returns. If the signal executes a signal handler, **sigsuspend()** returns after the signal handler returns, and restores the signal mask to the set that existed prior to the **sigsuspend()** call.

It is impossible to block the **SIGKILL** or **SIGSTOP** signal. This is enforced by the system without causing an error to be indicated.

RETURN VALUE

Since **sigsuspend()** suspends a process indefinitely, there is no successful completion return value. If a return occurs, a value of -1 is returned and **errno** is set to indicate the error.

ERRORS

sigsuspend() fails if any of the following conditions are encountered:

[EINTR]	**sigsuspend()** was interrupted by receipt of a signal.
[EFAULT]	*sigmask* points to an invalid address. The reliable detection of this error is implementation dependent.

AUTHOR

sigsuspend() was derived from the IEEE POSIX 1003.1-1988 Standard.

SEE ALSO

sigaction(2), sigpending(2), sigprocmask(2), sigsetops(3C), signal(5).

STANDARDS CONFORMANCE

sigsuspend(): AES, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

sigvector - software signal facilities

SYNOPSIS

```
#include <signal.h>

int sigvector(
    int sig,
    const struct sigvec *vec,
    struct sigvec *ovec
);
```

DESCRIPTION

The system defines a set of signals that can be delivered to a process. The set of signals is defined in *signal*(5), along with the meaning and side effects of each signal. This manual entry, along with those for *sigblock*(2), *sigsetmask*(2), *sigpause*(2), and *sigspace*(2), defines an alternate mechanism for handling these signals that ensures the delivery of signals and the integrity of signal handling procedures. The facilities described here should not be used in the same program as *signal*(2).

With the **sigvector()** interface, signal delivery resembles the occurrence of a hardware interrupt: the signal is blocked from further occurrence, the current process context is saved, and a new one is built. A process can specify a handler function to be invoked when a signal is delivered, or specify that a signal should be blocked or ignored. A process can also specify that a default action should be taken by the system when a signal occurs. It is possible to ensure a minimum amount of stack space for processing signals using **sigspace()** (see *sigspace*(2)).

All signals have the same priority. Signal routines execute with the signal that causes their invocation to be blocked, although other signals can yet occur. A global signal mask defines the set of signals currently blocked from delivery to a process. The signal mask for a process is initialized from that of its parent (normally 0). It can be changed with a **sigblock()**, **sigsetmask()**, or **sigpause()** call, or when a signal is delivered to the process.

A signal mask is represented as a **long**, with one bit representing each signal being blocked. The following macro defined in **<signal.h>** is used to convert a signal number to its corresponding bit in the mask:

```
#define   sigmask(signo)   (1L << (signo-1))
```

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently blocked by the process, it is delivered to the process. When a signal is delivered, the current state of the process is saved, a new signal mask is calculated (as described below), and the signal handler is invoked. The call to the handler is arranged so that if the signal handling routine returns normally, the process resumes execution in the same context as before the signal's delivery. If the process wishes to resume in a different context, it must arrange to restore the previous context itself.

When a signal is delivered to a process, a new signal mask is installed for the duration of the process' signal handler (or until a **sigblock()** or **sigsetmask()** call is made). This mask is formed by taking the current signal mask, computing the bit-wise inclusive OR with the value of *vec.sv_mask* (see below) from the most recent call to **sigvector()** for the signal to be delivered, and, unless the **SV_RESETHAND** flag is set (see below), setting the bit corresponding to the signal being delivered. When the user's signal handler returns normally, the original mask is restored.

sigvector() assigns a handler for the signal specified by *sig*. *vec* and *ovec* are pointers to *sigvec* structures that include the following elements:

```
void   (*sv_handler)();
long   sv_mask;
long   sv_flags;
```

If *vec* is non-zero, it specifies a handler routine (*sv_handler*), a mask (*sv_mask*) that the system should use when delivering the specified signal, and a set of flags (*sv_flags*) that modify the delivery of the signal. If *ovec* is non-zero, the previous handling information for the signal is returned to the user. If *vec* is zero, signal handling is unchanged. Thus, the call can be used to enquire about the current handling of a given signal. If *vec* and *ovec* point to the same structure, the value of *vec* is read prior to being overwritten.

The *sv_flags* field can be used to modify the receipt of signals. The following flag bits are defined:

SV_ONSTACK	Use the `sigspace()` allocated space.
SV_BSDSIG	Use the Berkeley signal semantics.
SV_RESETHAND	Use the semantics of *signal*(2).

If SV_ONSTACK is set, the system uses or permits the use of the space reserved for signal processing in the `sigspace()` system call.

If SV_BSDSIG is set, the signal is given the Berkeley semantics. The following signal is affected by this flag:

SIGCLD — In addition to being sent when a child process dies, the signal is also sent when any child's status changes from running to stopped. This would normally be used by a program such as `csh` (see *csh*(1)) when maintaining process groups under Berkeley job control.

If SV_RESETHAND is set, the signal handler is installed with the same semantics as a handler installed with *signal*(2). This affects signal mask set-up during the signal handler (see above) and whether the handler is reset after a signal is caught (see below).

If SV_RESETHAND is not set, once a signal handler is installed, it remains installed until another `sigvector()` call is made or an `exec()` system call is performed (see *exec*(2)). If SV_RESETHAND is set and the signal is not one of those marked "not reset when caught" under *signal*(5), the default action is reinstated when the signal is caught, prior to entering the signal-catching function. The "not reset when caught" distinction is not significant when `sigvector()` is called and SV_RESETHAND is not set.

The default action for a signal can be reinstated by setting *sv_handler* to SIG_DFL; this default usually results in termination of the process. If *sv_handler* is SIG_IGN the signal is usually subsequently ignored, and pending instances of the signal are discarded. The exact meaning of SIG_DFL and SIG_IGN for each signal is discussed in *signal*(5).

Certain system calls can be interrupted by a signal; all other system calls complete before the signal is serviced. The *scp* pointer described in *signal*(5) is never null if `sigvector()` is supported. *scp* points to a machine-dependent *sigcontext* structure. All implementations of this structure include the fields:

```
int     sc_syscall;
char    sc_syscall_action;
```

The value SYS_NOTSYSCALL for the *sc_syscall* field indicates that the signal is not interrupting a system call; any other value indicates which system call it is interrupting.

If a signal that is being caught occurs during a system call that can be interrupted, the signal handler is immediately invoked. If the signal handler exits normally, the value of the *sc_syscall_action* field is inspected; if the value is SIG_RETURN, the system call is aborted and the interrupted program continues past the call. The result of the interrupted call is -1 and `errno` is set to EINTR. If the value of the *sc_syscall_action* field is SIG_RESTART, the call is restarted. A call is restarted if, in the case of a `read()` or `write()` system call (see *read*(2) or *write*(2)), it had transferred no data. If some data had been transferred, the operation is considered to have completed with a partial transfer, and the *sc_syscall* value is SYS_NOTSYSCALL. Other values are undefined and reserved for future use.

Exiting the handler abnormally (such as with `longjmp()` — see *setjmp*(3C)) aborts the call, leaving the user responsible for the context of further execution. The value of *scp->sc_syscall_action* is ignored when the value of *scp->sc_syscall* is SYS_NOTSYSCALL. *scp->sc_syscall_action* is always initialized to SIG_RETURN before invocation of a signal handler. When an system call that can be interrupted is interrupted by multiple signals, if any signal handler returns a value of SIG_RETURN in *scp->sc_syscall_action*, all subsequent signal handlers are passed a value of SYS_NOTSYSCALL in *scp->sc_syscall*.

Note that calls to `read()`, `write()`, or `ioctl()` on fast devices (such as disks) cannot be interrupted, but I/O to a slow device (such as a printer) can be interrupted. Other system calls, such as those used for networking, also can be interrupted on some implementations. In these cases additional values can be specified for Programs that look at the values of *scp->sc_syscall* always should compare them to these symbolic constants; the numerical values represented by these constants might vary among implementations. System calls that can be interrupted and their corresponding values for *scp->sc_syscall* are listed below:

Call	sc_syscall value
read (slow devices)	SYS_READ
readv (slow devices)	SYS_READV
write (slow devices)	SYS_WRITE
writev (slow devices)	SYS_WRITEV
open (slow devices)	SYS_OPEN
ioctl (slow requests)	SYS_IOCTL
close (slow requests)	SYS_CLOSE
wait	SYS_WAIT
select	SYS_SELECT
pause	SYS_PAUSE
sigpause	SYS_SIGPAUSE
semop	SYS_SEMOP
msgsnd	SYS_MSGSND
msgrcv	SYS_MSGRCV

These system calls are not defined if the preprocessor macro _XPG2 is defined when <signal.h> is included. This is because the *X/Open Portability Guide, Issue 2* specifies a different meaning for the symbol SYS_OPEN (see *limits*(5)).

After a fork() or vfork() system call, the child inherits all signals, the signal mask, and the reserved signal stack space.

exec(2) resets all caught signals to the default action; ignored signals remain ignored, the signal mask remains unchanged, and the reserved signal stack space is released.

The mask specified in *vec* is not allowed to block signals that cannot be ignored, as defined in *signal*(5). This is enforced silently by the system.

If sigvector() is called to catch SIGCLD in a process that currently has terminated (zombie) children, a SIGCLD signal is delivered to the calling process immediately, or as soon as SIGCLD is unblocked if it is currently blocked. Thus, in a process that spawns multiple children and catches SIGCLD, it is sometimes advisable to reinstall the handler for SIGCLD after each invocation in case there are multiple zombies present. This is true even though the handling of the signal is not reset by the system, as with *signal*(2), because deaths of multiple processes while SIGCLD is blocked in the handler result in delivery of only a single signal. Note that the function must reinstall itself after it has called wait() or wait3(). Otherwise the presence of the child that caused the original signal always causes another signal to be delivered.

RETURN VALUE

Upon successful completion, sigvector() returns 0; otherwise, it returns -1 and sets errno to indicate the reason.

ERRORS

sigvector() fails and no new signal handler is installed if any of the following conditions are encountered:

[EFAULT]	Either *vec* or *ovec* points to memory that is not a valid part of the process address space. Reliable detection of this error is implementation dependent.
[EINVAL]	*sig* is not a valid signal number.
[EINVAL]	An attempt was made to ignore or supply a handler for a signal that cannot be caught or ignored; see *signal*(5).

WARNINGS

Restarting a *select*(2) call can sometimes cause unexpected results. If the select() call has a timeout specified, the timeout is restarted with the call, ignoring any portion that had elapsed prior to interruption by the signal. Normally this simply extends the timeout and is not a problem. However, if a handler repeatedly catches signals, and the timeout specified to select() is longer than the time between those signals, restarting the select() call effectively renders the timeout infinite.

sigvector() should not be used in conjunction with the facilities described under *sigset*(2V).

AUTHOR

sigvector() was developed by HP and the University of California, Berkeley.

SEE ALSO

kill(1), kill(2), ptrace(2), sigblock(2), signal(2), sigpause(2), sigsetmask(2), sigspace(2), setjmp(3C), signal(5), termio(7).

NAME

socket - create an endpoint for communication

SYNOPSIS

```
#include <sys/socket.h>
```

AF_CCITT only:

```
#include <x25/x25ccittproto.h>
```

```
int socket(int af, int type, int protocol);
```

DESCRIPTION

`socket()` creates an endpoint for communication and returns a descriptor. The socket descriptor returned is used in all subsequent socket-related system calls.

The *af* parameter specifies an address family to be used to interpret addresses in later operations that specify the socket. These address families are defined in the include files `<sys/socket.h>` and `<x25/ccittproto.h>`. The only currently-supported address families are:

AF_INET	(DARPA Internet addresses)
AF_UNIX	(path names on a local node)
AF_CCITT	(CCITT X.25 addresses)

The *type* specifies the semantics of communication for the socket. Currently defined types are:

SOCK_STREAM
SOCK_DGRAM (for AF_INET only)

A **SOCK_STREAM** type provides sequenced, reliable, two-way-connection-based byte streams. A **SOCK_DGRAM** socket supports datagrams (connectionless, unreliable messages of a fixed, typically small, maximum length).

protocol specifies a particular protocol to be used with the socket. Normally, only a single protocol exists to support a particular socket type using a given address family. However, many protocols may exist, in which case a particular protocol must be specified. The protocol number to use depends on the **communication domain** in which communication is to take place (see *services*(4) and *protocols*(4)). *protocol* can be supplied as zero, in which case the system chooses a protocol type to use.

Sockets of type **SOCK_STREAM** are byte streams similar to pipes except that they are full-duplex instead of half-duplex. A stream socket must be in a *connected* state before any data can be sent or received on it. A connection to another socket is created with a `connect()` or `accept()` call. Once connected, data can be transferred using some variant of the `send()` and `recv()` or the `read()` and `write()` calls. When a session has been completed, a `close()` can be performed.

TCP, the communications protocol used to implement **SOCK_STREAM** for AF_INET sockets, ensures that data is not lost or duplicated. If a peer has buffer space for data and the data cannot be successfully transmitted within a reasonable length of time, the connection is considered broken and the next `recv()` call indicates an error with `errno` set to ETIMEDOUT. If **SO_KEEPALIVE** is set and the connection has been idle for two hours, the TCP protocol sends "keepalive" packets every 75 seconds to determine whether the connection is active. These transmissions are not visible to users, and cannot be read by a `recv()` call. If the remote system does not repond within 10 minutes (i.e., after 8 "keepalive" packets have been sent), the next socket call (e.g., `recv()`) returns an error sets `errno` to ETIMEDOUT. A **SIGPIPE** signal is raised if a process sends on a broken stream; this causes naive processes that do not handle the signal to exit. An end-of-file condition (zero bytes read) is returned if a process tries to read on a broken stream.

SOCK_DGRAM sockets allow sending of messages to correspondents named in `send()` calls. It is also possible to receive messages at such a socket with `recv()`.

The operation of sockets is controlled by socket level options set by the `setsockopt()` system call described by the *getsockopt*(2) manual entry. These options are defined in the file `<sys/socket.h>` and explained in the *getsockopt*(2) manual entry.

X.25 only:

Socket endpoints for communication over an X.25/9000 link can be in either address family **AF_INET** or **AF_CCITT.** If the socket is in the **AF_INET** family, the connection will behave as described above. TCP is used if the socket type is **SOCK_STREAM**; **UDP** is used if the socket type is **SOCK_DGRAM**. In both cases,

Internet Protocol (IP) and the X.25-to-IP interface module are used. If the socket is in the **AF_CCITT** address family, only the **SOCK_STREAM** socket type is supported. Refer to the topic *Comparing X.25 Level 3 Access to IP* in the *X.25 Programmer's Guide* for more details on the difference between programmatic access to X.25 via IP and X.25 Level 3.

If the socket is of the **AF_CCITT** family, the connection and all other operations pass data directly from the application to the X.25 Packet Level (level 3) without passing through a TCP or UDP protocol. Connections of the **AF_CCITT** family cannot use most of the socket level options described in the *getsockopt*(2) manual entry. However, AF_CCITT connections can use many X.25-specific **ioctl()** calls, described by *socketx25*(7).

DEPENDENCIES

AF_CCITT

Only the **SOCK_STREAM** *type* is supported.

RETURN VALUE

Upon successful completion, **socket()** returns a valid file descriptor referencing the socket. Otherwise, it returns -1 and sets **errno** to indicate the error.

ERRORS

socket() fails if any of the following conditions are encountered:

[EHOSTDOWN]	The networking subsystem has not been started up.
[EAFNOSUPPORT]	The specified address family is not supported in this version of the system.
[ESOCKTNOSUPPORT]	The specified socket type is not supported in this address family.
[EPROTONOSUPPORT]	The specified protocol is not supported.
[EMFILE]	The per-process descriptor table is full.
[ENOBUFS]	No buffer space is available. The socket cannot be created.
[ENFILE]	The system's table of open files is temporarily full and no more **socket()** calls can be accepted.
[EPROTOTYPE]	The type of socket and protocol do not match.
[ETIMEDOUT]	Connection timed-out.
[EINVAL]	SOCK_DGRAM sockets currently not supported for AF_UNIX address family.

AUTHOR

socket() was developed by the University of California, Berkeley.

SEE ALSO

accept(2), bind(2), connect(2), getsockname(2), getsockopt(2), ioctl(2), listen(2), recv(2), select(2), send(2), shutdown(2), af_ccitt(7F), socket(7), socketx25(7), tcp(7P), udp(7P), unix(7P).

NAME

socketpair - create a pair of connected sockets

SYNOPSIS

```
#include <sys/socket.h>

int socketpair(int af, int type, int protocol, int sv[2]);
```

DESCRIPTION

socketpair() creates an unnamed pair of connected sockets and returns two file descriptors in *sv*[0] and *sv*[1]. The two sockets are indistinguishable. *af* specifies the address family. See *socket*(2). *type* specifies the semantics of communication for the socket. *protocol* specifies a particular protocol to be used. *protocol* can be supplied as zero, in which case the system chooses a protocol type to use.

RETURN VALUES

Upon successful completion, **socketpair()** returns 0; otherwise, it returns -1 and sets **errno** to indicate the error.

ERRORS

socketpair() fails if any of the following conditions are encountered:

[EMFILE]	The per-process file descriptor table is full.
[ENFILE]	The system file table is temporarily full.
[EAFNOSUPPORT]	The specified address family is not supported in this version of the system.
[EPROTONOSUPPORT]	The specified protocol is not supported in this version of the system.
[EOPNOSUPPORT]	The specified protocol does not support creation of socket pairs.
[EFAULT]	The *sv* parameter is not valid.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.

DEPENDENCIES

This call is supported only for **AF_UNIX**.

AUTHOR

socketpair() was developed by the University of California, Berkeley.

SEE ALSO

read(2), write(2), socket(2).

NAME

stat, lstat, fstat - get file status

SYNOPSIS

```
#include <sys/stat.h>

int stat(const char *path, struct stat *buf);

int lstat(const char *path, struct stat *buf);

int fstat(int fildes, struct stat *buf);
```

DESCRIPTION

`stat()` obtains information about the named file.

path points to a path name naming a file. Read, write, or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable.

Similarly, `fstat()` obtains information about an open file known by the file descriptor *fildes*, obtained from a successful `open()`, `creat()`, `dup()`, `fcntl()`, or `pipe()` system call (see *open*(2), *creat*(2), *dup*(2), *fcntl*(2), or *pipe*(2)).

`lstat()` is similar to `stat()` except when the named file is a symbolic link, in which case `lstat()` returns the information about the link, while `stat()` returns information about the file to which the link points.

buf is a pointer to a `stat()` structure into which information is placed concerning the file.

The contents of structure `stat()` pointed to by *buf* include the following members. Note that there is no necessary correlation between the placement in this list and the order in the structure.

```
dev_t     st_dev;        /* ID of device containing a */
                         /* directory entry for this file */
ino_t     st_ino;        /* Inode number */
ushort     st_fstype;    /* Type of filesystem this file  */
                         /* is in; see vfsmount(2) */
ushort     st_mode;      /* File type, attributes, and */
                         /* access control summary */
ushort     st_basemode   /* Permission bits (see chmod(1)) */
ushort     st_nlink;     /* Number of links */
uid_t     st_uid;        /* User ID of file owner */
gid_t     st_gid;        /* Group ID of file group */
dev_t     st_rdev;       /* Device ID; this entry defined */
                         /* only for char or blk spec files */
off_t     st_size;       /* File size (bytes) */
time_t     st_atime;     /* Time of last access */
time_t     st_mtime;     /* Last modification time */
time_t     st_ctime;     /* Last file status change time */
                         /* Measured in secs since */
                         /* 00:00:00 GMT, Jan 1, 1970 */
uint     st_acl:1;       /* Set if the file has optional */
                         /* access control list entries */
```

Field contents are as follows:

`st_atime` Time when file data was last accessed. Changed by the following system calls: `creat()`, `mknod()`, `pipe()`, `read()`, `readv()` (see *read*(2)), and `utime()`. If a file is mapped into virtual memory, accesses of file data through the mapping may also modify `st_mtime`. See *mmap(2)*.

`st_mtime` Time when data was last modified. Changed by the following system calls: `creat()`, `truncate()`, `ftruncate()`, (see *truncate*(2)), `mknod()`, `pipe()`, `prealloc()`, `utime()`, `write()`, and `writev()` (see *write*(2)). Also changed by `close()` when the reference count reaches zero on a named pipe (FIFO special) file that contains data. If a file is mapped into virtual memory, updates of file data through the mapping may also modify `st_mtime`. See *mmap(2)*.

st_ctime Time when file status was last changed. Changed by the following system calls: **chmod()**, **chown()**, **creat()**, **fchmod()**, **fchown()**, **truncate()**, **ftruncate()**, (see *truncate*(2)), **link()**, **mknod()**, **pipe()**, **prealloc()**, **rename()**, **setacl()**, **unlink()**, **utime()**, **write()**, and **writev()** (see *write*(2)).

The **touch** command (see *touch*(1) can be used to explicitly control the times of a file.

st_mode The value returned in this field is the bit-wise inclusive OR of a value indicating the file's type, attribute bits, and a value summarizing its access permission. See *mknod*(2).

For ordinary users, the least significant nine bits consist of the file's permission bits modified to reflect the access granted or denied to the caller by optional entries in the file's access control list.

For users with appropriate privileges
the least significant nine bits are the file's access permission bits. In addition, the **S_IXUSR** (execute by owner) mode bit is set if the following conditions are met:

- The file is a regular file,
- No permission execute bits are set, and
- An execute bit is set in one or more of the file's optional access control list entries.

The write bit is not cleared for a file on a read-only file system or a shared-text program file that is being executed. However, **getaccess()** clears this bit under these conditions (see *getaccess*(2).

RETURN VALUE

Upon successful completion, 0 is returned. Otherwise, -1 is returned and **errno** is set to indicate the error.

ERRORS

stat() and **lstat()** fail if any of the following conditions are encountered:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist (for example, *path* is null or a component of *path* does not exist).
[EACCES]	Search permission is denied for a component of the path prefix.
[EFAULT]	*buf* or *path* points to an invalid address. The reliable detection of this error is implementation dependent.
[ELOOP]	Too many symbolic links were encountered in translating the path name.
[ENAMETOOLONG]	The length of the specified path name exceeds **PATH_MAX** bytes, or the length of a component of the path name exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect.

fstat() fails if any of the following conditions are encountered:

[EBADF]	*fildes* is not a valid open file descriptor.
[EFAULT]	*buf* points to an invalid address. The reliable detection of this error is implementation dependent.

DEPENDENCIES

HP Clustered Environment

The contents of the **stat()** structure include the following additional members:

```
cnode_t     st_cnode;     /* cnode ID of machine */
                          /* where the inode lives */
cnode_t     st_rcnode     /* cnode ID where this */
                          /* device file can be used */
dev_t       st_realdev;   /* Real device number of device */
                          /* containing the inode for this file */
```

st_dev The ID number for the volume on which the inode exists. This number may or may not be the device number for the device containing the volume. Device numbers are not

unique throughout a cluster, but the value of **st_dev** is guaranteed to be unique among all volumes currently mounted in the file system. The device number for the volume can always be found in the field **st_realdev**, which, together with **st_cnode**, fully specifies the device containing the volume.

CD-ROM
The **st_uid** and **st_gid** fields are set to -1 if they are not specified on the disk for a given file.

NFS
The **st_basemode** and **st_acl** fields are zero on files accessed remotely.

WARNINGS

Access Control Lists

Access control list descriptions in this entry apply only to standard HP-UX operating systems. If HP-UX BLS software has been installed, access control lists are handled differently. Refer to HP-UX BLS documentation for information about access control lists in the HP-UX BLS environment.

AUTHOR

stat() and **fstat()** were developed by AT&T. **lstat()** was developed by the University of California, Berkeley.

SEE ALSO

touch(1), chmod(2), chown(2), creat(2), link(2), mknod(2), pipe(2), read(2), rename(2), setacl(2), time(2), truncate(2), unlink(2), utime(2), write(2), acl(5), stat(5), privilege(5).

STANDARDS CONFORMANCE

stat(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

fstat(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

lstat(): AES

NAME

statfs, fstatfs - get file system statistics

SYNOPSIS

```
#include <sys/vfs.h>

int statfs(const char *path, struct statfs *buf);

int fstatfs(int fildes, struct statfs *buf);
```

DESCRIPTION

`statfs()` returns information about a mounted file system. *path* is the path name of any file within the mounted file system.

buf is a pointer to a `statfs()` structure into which information is placed concerning the file system. The contents of the structure pointed to by *buf* include the following members:

```
long     f_bavail;  /* free blocks available to non-superuser */
long     f_bfree;   /* free blocks */
long     f_blocks;  /* total blocks in file system */
long     f_bsize;   /* fundamental file system block size in bytes */
long     f_ffree;   /* free file nodes in file system */
long     f_files;   /* total file nodes in file system */
long     f_type;    /* type of info, zero for now */
fsid_t   f_fsid     /* file system ID.  f_fsid[1] is MOUNT_UFS,
                       MOUNT_NFS, or MOUNT_CDFS */
```

A *file node* is a structure in the file system hierarchy that describes a file. For mounted HP-UX volumes, *file node* is an HP-UX inode. For other types of mounts, *file node* is defined by the system embodying the file pointed to by *path*.

Fields that are undefined for a particular file system are set to -1.

`fstatfs()` returns similar information about an open file referred to by file descriptor *fildes*.

RETURN VALUE

`statfs()` and `fstatfs()` return 0 upon successful completion; otherwise, they return -1 and set `errno` to indicate the error.

ERRORS

`statfs()` fails if any of the following conditions are encountered:

[EACCES]	Search permission is denied for a component of the path prefix.
[EFAULT]	*buf* or *path* points to an invalid address.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[ELOOP]	Too many symbolic links are encountered in translating the path name.
[ENAMETOOLONG]	A component of *path* exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect, or *path* exceeds **PATH_MAX** bytes.
[ENOENT]	The named file does not exist.
[ENOTDIR]	A component of the path prefix is not a directory.

`fstatfs()` fails if any of the following conditions are encountered:

[EBADF]	*fildes* is not a valid open file descriptor.
[EFAULT]	*buf* points to an invalid address.
[EIO]	An I/O error occurs while reading from or writing to the file system.

AUTHOR

`statfs()` and `fstatfs()` were developed by Sun Microsystems, Inc.

SEE ALSO

df(1M), stat(2), ustat(2).

NAME

stime - set time and date

SYNOPSIS

```
#include <time.h>

int stime(const time_t *tp);
```

DESCRIPTION

`stime()` sets the system's idea of the time and date. *tp* points to the value of time as measured in seconds from 00:00:00 UTC (Coordinated Universal Time) January 1, 1970.

RETURN VALUE

Upon successful completion, `stime()` returns a value of 0; otherwise, a value of -1 is returned and `errno` is set to indicate the error.

ERRORS

[EPERM] `stime()` fails if the effective user ID of the calling process is not super-user.

DEPENDENCIES

HP Clustered Environment

On systems that are members of a cluster, setting the time sets the time and date for all systems in the cluster.

SEE ALSO

date(1), gettimeofday(2), time(2).

STANDARDS CONFORMANCE

`stime()`: SVID2, XPG2

NAME

stty, gtty - control device

SYNOPSIS

```
#include <sgtty.h>

int stty(int fildes, const struct sgttyb *argp);

int gtty(int fildes, struct sgttyb *argp);
```

REMARKS

These system calls are preserved for backward compatibility with Bell Version 6. They provide as close an approximation as possible to the old Version 6 functions. All new code should use the **TCSETA** and **TCGETA** `ioctl()` calls described in *termio*(7).

DESCRIPTION

For certain status settings and status inquiries about terminal devices, the functions `stty()` and `gtty()` are equivalent to

```
ioctl(fildes, TIOCSETP, argp)
```

and

```
ioctl(fildes, TIOCGETP, argp)
```

respectively; see *termio*(7).

RETURN VALUE

`stty()` returns zero if the call was successful or -1 if the file descriptor does not refer to the kind of file for which it was intended.

SEE ALSO

stty(1), exec(2), sttyV6(7), tty(7), termio(7).

NAME

swapon - add swap space for interleaved paging/swapping

SYNOPSIS

```
#include <unistd.h>

int swapon(
    const char *path, ...
    /* [int min,
       int limit,
       int reserve,]
       int priority */
 );
```

DESCRIPTION

If *path* names a block device file:

swapon() makes it available to the system at the specified *priority* for allocation for paging and swapping.

In this form, **swapon()** takes only two arguments: the *path* to the block device file, and the *priority*.

The device associated with *path* can be a device already known to the system, defined at system configuration time, or it can be a previously unspecified device.

If the device was already defined at system configuration time and also has a start and/or size defined for that swap device, these values are used.

Otherwise, if a filesystem exists on the device, swap is added following the filesystem, or if no filesystem exists, the complete device is used for swap.

See the appropriate system administrator's manual for information on how the size of the swap area is calculated.

If *path* names a directory:

swapon() makes the blocks on the file system rooted at *path,* available for paging and swapping.

The *min*, *limit*, and *reserve* arguments are passed and used only if the *path* argument names a directory.

min indicates the number of file system blocks to take from the file system when **swapon()** is called.

limit indicates the maximum number of file system blocks the swap system is allowed to take from the file system.

reserve indicates the number of file system blocks that are saved for file system use only.

priority indicates the order in which the swap space from this device or file system is used. Space is taken from the lower-priority systems first.

swapon() can be used only by users who have appropriate privileges.

ERRORS

swapon() fails if any of the following conditions are encountered:

[EALREADY]	The device or directory associated with *path* already has swap turned on.
[ENXIO]	The device associated with *path* could not be opened.
[EBUSY]	The device associated with *path* is already in use.
[ENODEV]	The device associated with *path* does not exist.
[EPERM]	The effective user ID is not a user with appropriate privileges.
[ELOOP]	Too many symbolic links were encountered in translating the path name.
[ENOTBLK]	The *path* argument is not a block special file or the root directory of a file system.

[ENOENT]	The system-imposed limit on the number of swap file entries has been reached.
[ENOSPC]	There is is not enough available space on the specified file system or device.
[EINVAL]	The node (see *cluster*(1M)) attempting to add swap had no swap configured at boot time.
[ENOSYS]	The device associated with *path* was specified at system configuration time to add swap following the file system, but no file system was found.
[EEXIST]	The device associated with *path* was specified at system configuration time to add swap at a specified location, but that location is within an existing file system on the device.
[EIO]	Unable to read the device associated with *path*.
[EROFS]	The device associated with *path* is read-only.
[EFAULT]	The LIF header on the device associated with *path* contains inconsistent directory data.
[ENAMETOOLONG]	The length of the specified path name exceeds **PATH_MAX** bytes, or the length of a component of the path name exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect.

WARNINGS

No means is available to stop swapping to a device.

The system allocates no less than the amount specified in *min*. However, to make the most efficient use of space, more than the amount requested might be taken from the file system. The actual amount taken will not exceed the number of file system blocks indicated in *reserve*.

Swapping to a file system is usually slower than swapping to a device.

AUTHOR

swapon() was developed by the University of California, Berkeley.

SEE ALSO

swapon(1M), privilege(5).

NAME

symlink - make symbolic link to a file

SYNOPSIS

```
#include <symlink.h>

int symlink(const char *name1, const char *name2);
```

DESCRIPTION

`symlink()` creates a file *name2*, which is a symbolic link to *name1*. Either name can be an arbitrary path name. The files need not be on the same file system.

RETURN VALUE

Upon successful completion, a zero value is returned. If an error occurs, the error code is stored in `errno` and a -1 value is returned.

ERRORS

The symbolic link is made unless one or more of the following is true:

[ENOTDIR]	A component of the *name2* prefix is not a directory.
[ENAMETOOLONG]	A component of either path name exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect, or the entire length of either path name exceeds **PATH_MAX** bytes.
[ENOENT]	A component of the *name2* prefix does not exist.
[EACCES]	A component of the *name2* path prefix denies search permission.
[EDQUOT]	User's disk quota block or inode limit has been reached for this file system.
[ELOOP]	Too many symbolic links were encountered in translating the path name.
[EEXIST]	*name2* already exists.
[EIO]	An I/O error occurred while making the directory entry for *name2*, allocating the inode for *name2*, or writing out the link contents of *name2*.
[EROFS]	The file *name2* resides on a read-only file system.
[ENOSPC]	The directory in which the entry for the new symbolic link is being placed cannot be extended because there is no space left on the file system containing the directory.
[ENOSPC]	The new symbolic link cannot be created because there is no space left on the file system that will contain the symbolic link.
[ENOSPC]	There are no free inodes on the file system on which the symbolic link is being created.
[EIO]	An I/O error occurred while making the directory entry or allocating the inode.
[EFAULT]	*name1* or *name2* points outside the process' allocated address space. The reliable detection of this error is implementation dependent.

AUTHOR

`symlink()` was developed by the University of California, Berkeley.

SEE ALSO

cp(1), link(2), readlink(2), unlink(2), symlink(4).

STANDARDS CONFORMANCE

`symlink()`: AES

NAME

sync, lsync - update super-block

SYNOPSIS

```
#include <unistd.h>

void sync(void);

void lsync(void);
```

DESCRIPTION

`sync()` causes all information in memory that should be on disk to be written out. This includes modified super blocks, modified inodes, and delayed block I/O.

It should be used by commands and programs that examine a file system, such as `fsck`, `df`, etc. It is mandatory before a shutdown.

The writing, although scheduled, is not necessarily complete upon return from *sync*.

In some HP-UX systems, `sync()` may be reduced to a no-op. This is permissible on a system which does not cache buffers, or in a system that in some way ensures that the disks are always in a consistent state.

In the HP Clustered Environment, `sync()` causes updates of all file systems in the cluster to be written out, while `lsync()` performs only a local `sync()`; that is, local buffers are flushed to disk and to remote nodes of the cluster, but remote nodes do not flush their own pages.

AUTHOR

`sync()` was developed by HP and AT&T Bell Laboratories. `lsync()` was developed by HP.

SEE ALSO

sync(1M), fsync(2).

STANDARDS CONFORMANCE

`sync()`: SVID2, XPG2

NAME

sysconf - get configurable system variables

SYNOPSIS

```
#include <unistd.h>

long sysconf(int name);

int CPU_IS_PA_RISC(long cpuvers);

int CPU_IS_HP_MC68K(long cpuvers);
```

DESCRIPTION

`sysconf()` provides a way for applications to determine the current value of a configurable limit or variable.

The *name* argument represents the system variable being queried.

The following table lists the configuration variables whose values can be determined by calling `sysconf()`, and for each variable, the associated value of the *name* argument and the value returned:

Variable	Value of *name*	Value Returned
`AES_OS_VERSION`	`_SC_AES_OS_VERSION`	Version number of OSF/AES OSC supported
`ARG_MAX`	`_SC_ARG_MAX`	Maximum total length of the arguments for `exec()` in bytes, including environment data (see *exec*(2))
`ATEXIT_MAX`	`_SC_ATEXIT_MAX`	Maximum number of functions that can be registered with `atexit()` (see *atexit*(2))
`BC_BASE_MAX`	`_SC_BC_BASE_MAX`	Maximum ibase (input number radix) and obase (output number radix) allowed by `bc` (see *bc*(1))
`BC_DIM_MAX`	`_SC_BC_DIM_MAX`	Maximum number of elements in an array permitted by `bc` (see *bc*(1))
`BC_SCALE_MAX`	`_SC_BC_SCALE_MAX`	Maximum scale factor (number of digits to the right of the decimal point) allowed by `bc` (see *bc*(1))
`BC_STRING_MAX`	`_SC_BC_STRING_MAX`	Maximum length of strings allowed by `bc` (see *bc*(1))
`CHILD_MAX`	`_SC_CHILD_MAX`	Maximum number of simultaneous processes per user ID (see *fork*(2))
`CLK_TCK`	`_SC_CLK_TCK`	Number of clock intervals per second for `times()` (see *times*(2))
`CLOCKS_PER_SEC`	`_SC_CLOCKS_PER_SEC`	Number of clock ticks per second for `clock()` (see *clock*(3C))
`COLL_WEIGHTS_MAX`	`_SC_COLL_WEIGHTS_MAX`	Maximum number of weights that can be assigned to an entry of the `LC_COLLATE order` keyword in a `localedef` input file (see *localedef*(1M))
`CPU_VERSION`	`_SC_CPU_VERSION`	Version of CPU architecture (see below)
`EXPR_NEST_MAX`	`_SC_EXPR_NEST_MAX`	Maximum parenthesis nesting level for `expr` expressions (see *expr*(1))
`IO_TYPE`	`_SC_IO_TYPE`	Type of I/O drivers the kernel supports (see below)
`LINE_MAX`	`_SC_LINE_MAX`	Maximum number of bytes in an input line (including the newline) for POSIX.2 utilities

NGROUPS_MAX	_SC_NGROUPS_MAX	Maximum number of simultaneous supplementary group IDs per process
OPEN_MAX	_SC_OPEN_MAX	Maximum number of files that one process can have open at one time
PAGE_SIZE	_SC_PAGE_SIZE	Kernel memory page size
PASS_MAX	_SC_PASS_MAX	Maximum number of significant bytes in a password
POSIX_JOB_CONTROL	_SC_JOB_CONTROL	Positive if the system supports POSIX job control; –1 otherwise
POSIX_SAVED_IDS	_SC_SAVED_IDS	Positive if each process has a saved set-user-ID and a saved set-group-ID; –1 otherwise
POSIX_VERSION	_SC_VERSION	Approval date of the POSIX.1 Standard (such as 199009 for POSIX.1-1990) to which the system conforms. This value indicates the year (first four digits) and month (next two digits) that the standard was approved by the IEEE Standards Board.
POSIX2_C_BIND	_SC_2_C_BIND	Equal to 1 if the POSIX.2 C Language Bindings Option is available through the `c89` utility; –1 otherwise
POSIX2_C_DEV	_SC_2_C_DEV	Equal to 1 if the POSIX.2 C Language Development Utilities Option is supported; –1 otherwise
POSIX2_C_VERSION	_SC_2_C_VERSION	Current version of the POSIX.2 C Language Binding Option supported (same format as `_POSIX_VERSION`); –1 otherwise.
POSIX2_FORT_DEV	_SC_2_FORT_DEV	Equal to 1 if the POSIX.2 FORTRAN Development Utilities Option is supported; –1 otherwise
POSIX2_FORT_RUN	_SC_2_FORT_RUN	Equal to 1 if the POSIX.2 Fortran Runtime Utilities Option is supported; –1 otherwise
POSIX2_LOCALEDEF	_SC_2_LOCALEDEF	Equal to 1 if locales can be created with the POSIX.2 `localedef` utility; –1 otherwise
POSIX2_SW_DEV	_SC_2_SW_DEV	Equal to 1 if the POSIX.2 Software Development Utilities Option is supported; –1 otherwise
POSIX2_UPE	_SC_2_UPE	Equal to 1 if the POSIX.2 User Portability Utilities Option is supported; –1 otherwise
POSIX2_VERSION	_SC_2_VERSION	Current version of POSIX.2 (same format as `_POSIX_VERSION`)
RE_DUP_MAX	_SC_RE_DUP_MAX	Maximum number of repeated occurrences of a regular expression permitted when using the interval notation \{*m*,*n*\} (see *regcomp*(3C))
SECURITY_CLASS	_SC_SECURITY_CLASS	DoD security level (see below)
STREAM_MAX	_SC_STREAM_MAX	Maximum number of stdio streams that one process can have open at one time
TZNAME_MAX	_SC_TZNAME_MAX	Maximum number of bytes in a timezone name for the `TZ` environment variable

XOPEN_CRYPT	_SC_XOPEN_CRYPT	Equal to 1 if the X/Open Encryption Feature Group is supported; –1 otherwise
XOPEN_ENH_I18N	_SC_XOPEN_ENH_I18N	Equal to 1 if the X/Open Enhanced Internationalization Feature Group is supported; –1 otherwise
XOPEN_SHM	_SC_XOPEN_SHM	Equal to 1 if the X/Open Shared Memory Feature Group is supported; –1 otherwise
XOPEN_VERSION	_SC_XOPEN_VERSION	Issue number of *X/Open Portability Guide* supported

Some of the variables in the table are defined as constants in <limits.h> (see *limits*(5)). The associated values of the *name* argument are defined in <unistd.h>.

The SECURITY_CLASS variable (returned by sysconf(_SC_SECURITY_CLASS)) can have the following possible values with meanings as indicated:

Value	Meaning
SEC_CLASS_NONE	No DoD security level supported
SEC_CLASS_C2	DoD C2 level security
SEC_CLASS_B1	DoD B1 level security

The possible values of the IO_TYPE variable (returned by sysconf(_SC_IO_TYPE)) and their meanings are:

Value	Meaning
IO_TYPE_WSIO	Workstation I/O (used by Series 300/400/700)
IO_TYPE_SIO	Server I/O (used by Series 800)

Since the Series 700 instruction set is compatible with Series 800 but its I/O system differs, IO_TYPE can be used to detect which I/O system is present in a single executable program that can be run on either a Series 700 or a Series 800.

The possible values of the CPU_VERSION variable (returned by sysconf(_SC_CPU_VERSION)) and their meanings are:

Value	Meaning
CPU_PA_RISC1_0	HP Precision Architecture RISC Version 1.0
CPU_PA_RISC1_1	HP Precision Architecture RISC Version 1.1
CPU_HP_MC68020	Motorola MC68020
CPU_HP_MC68030	Motorola MC68030
CPU_HP_MC68040	Motorola MC68040

The CPU_IS_PA_RISC() and CPU_IS_HP_MC68K() functions classify *cpuvers*, a value of the CPU_VERSION variable, as to its processor family.

RETURN VALUE

Upon successful completion, sysconf() returns the value of the named variable. If the value of *name* is not valid, sysconf() returns –1 and sets errno to indicate the error. If the variable corresponding to *name* is not defined, sysconf() returns –1, but does not change errno.

CPU_IS_PA_RISC() returns positive non-zero if *cpuvers* is an HP PA-RISC processor; zero if not.

CPU_IS_HP_MC68K() returns positive non-zero if *cpuvers* is a "Motorola MC680x0" processor; zero if not.

ERRORS

sysconf() fails if:

[EINVAL] The value of *name* is not valid.

EXAMPLES

The following example determines the number of times the system clock ticks each second:

```
#include <unistd.h>
long ticks;
  ...
ticks = sysconf(_SC_CLK_TCK);
```

The following example determines whether the current processor is an HP PA-RISC machine:

```
#include <unistd.h>
if (CPU_IS_PA_RISC(sysconf(_SC_CPU_VERSION)))
  ...
```

WARNINGS

`CPU_IS_PA_RISC()` and `CPU_IS_HP_MC68K()` are implemented as macros.

Normally, the values returned from `sysconf()` do not change during the lifetime of the calling process. However, the value of the symbolic constant `_POSIX_VERSION` and thus the value of `sysconf(_SC_VERSION)` can vary under certain circumstances. If either of the feature test macros `_POSIX1_1988` or `_XPG3` is defined by the programmer prior to including `<unistd.h>`, the value of `_POSIX_VERSION` is defined as `198808`, in conformance with POSIX.1-1988, FIPS 151-1, and XPG3. Otherwise, the value of `_POSIX_VERSION` is defined as `199009`, in conformance with POSIX.1-1990.

Similarly, the value of the symbolic constant `_XOPEN_VERSION` and thus the value of `sysconf(_SC_XOPEN_VERSION)` can vary under certain circumstances. If the feature test macro `_XPG3` is defined by the programmer prior to including `<unistd.h>`, the value of `_XOPEN_VERSION` is defined as `3`, in conformance with XPG3. Otherwise, the value of `_XOPEN_VERSION` is defined as `4`, in conformance with XPG4.

See *stdsyms*(5) for more information about these feature test macros.

AUTHOR

`sysconf()` was developed by HP and POSIX.

`CPU_IS_PA_RISC()` and `CPU_IS_HP_MC68K()` were developed by HP.

SEE ALSO

getconf(1), atexit(2), exec(2), fork(2), getrlimit(2), pathconf(2), times(2), clock(3C), regcomp(3C), limits(5), stdsyms(5), unistd(5), x_open(5).

STANDARDS CONFORMANCE

`sysconf()`: AES, XPG3, XPG4, FIPS 151-2, POSIX.1, POSIX.2

NAME

time - get time

SYNOPSIS

```
#include <time.h>

time_t time(time_t *tloc);
```

DESCRIPTION

`time()` returns the value of time in seconds since the Epoch.

If *tloc* is not a null pointer, the return value is also assigned to the object to which it points.

RETURN VALUE

Upon successful completion, `time()` returns the value of time. Otherwise, a value of (time_t)-1 is returned and `errno` is set to indicate the error.

ERRORS

[EFAULT] `time()` fails if *tloc* points to an illegal address. The reliable detection of this error is implementation dependent.

SEE ALSO

date(1), gettimeofday(2), stime(2), ctime(3C), strftime(3C).

STANDARDS CONFORMANCE

`time()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

times - get process and child process times

SYNOPSIS

```
#include <sys/times.h>

clock_t times(struct tms *buffer);
```

DESCRIPTION

`times()` fills the structure pointed to by *buffer* with time-accounting information. The structure defined in `<sys/times.h>` is as follows:

```
struct tms {
    clock_t     tms_utime;      /* user time */
    clock_t     tms_stime;      /* system time */"
    clock_t     tms_cutime;     /* user time, children */
    clock_t     tms_cstime;     /* system time, children */
};
```

This information comes from the calling process and each of its terminated child processes for which it has executed a `wait()`, `wait3()`, or `waitpid()`. The times are in units of 1/`CLK_TCK` seconds, where `CLK_TCK` is processor dependent The value of `CLK_TCK` can be queried using the `sysconf()` function (see *sysconf*(2)).

`tms_utime` is the CPU time used while executing instructions in the user space of the calling process.

`tms_stime` is the CPU time used by the system on behalf of the calling process.

`tms_cutime` is the sum of the `tms_utimes` and `tms_cutimes` of the child processes.

`tms_cstime` is the sum of the `tms_stimes` and `tms_cstimes` of the child processes.

RETURN VALUE

Upon successful completion, `times()` returns the elapsed real time, in units of1/`CLK_TCK` of a second, since an arbitrary point in the past (such as system start-up time). This point does not change from one invocation of `times()` to another. If `times()` fails, -1 is returned and `errno` is set to indicate the error.

ERRORS

[EFAULT] `times()` fails if *buffer* points to an illegal address. The reliable detection of this error is implementation dependent.

SEE ALSO

time(1), gettimeofday(2), exec(2), fork(2), sysconf(2), time(2), wait(2).

WARNINGS

Not all CPU time expended by system processes on behalf of a user process is counted in the system CPU time for that process.

STANDARDS CONFORMANCE

`times()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

truncate, ftruncate - truncate a file to a specified length

SYNOPSIS

```
#include <unistd.h>

int truncate(const char *path, size_t length);

int ftruncate(int fildes, size_t length);
```

DESCRIPTION

`truncate()` causes the file named by *path* or referenced by *fd* to have a size of *length* bytes. If the file previously was larger than this size, the extra data is lost. If it was previously shorter, bytes between the old and new lengths are read as zeroes. With `ftruncate()`, the file must be open for writing; for `truncate()` the user must have write permission for the file.

RETURN VALUES

`truncate()` returns a value of 0 if successful; otherwise a -1 is returned, and `errno` is set to indicate the error.

ERRORS

`truncate()` fails if any of the following conditions are encountered:

[ENOTDIR]	A component of the path prefix of *path* is not a directory.
[EACCES]	A component of the path prefix denies search permission.
[EACCES]	Write permission is denied on the file.
[EINVAL]	*length* was greater than the maximum file size.
[EISDIR]	The named file is a directory.
[EROFS]	The named file resides on a read-only file system.
[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed.
[EFAULT]	*path* points outside the process's allocated address space. The reliable detection of this error is implementation dependent.
[ELOOP]	Too many symbolic links were encountered in translating the path name.
[ENAMETOOLONG]	The length of the specified path name exceeds **PATH_MAX** bytes, or the length of a component of the path name exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect.
[EDQUOT]	User's disk quota block limit has been reached for this file system.

`ftruncate()` fails if any of the following conditions are encountered:

[EBADF]	*fd* is not a valid file descriptor.
[EINVAL]	*fd* references a file that was opened without write permission.
[EDQUOT]	User's disk quota block limit has been reached for this file system.

AUTHOR

`truncate()` was developed by the University of California, Berkeley.

SEE ALSO

open(2).

STANDARDS CONFORMANCE

`truncate()`: AES

`ftruncate()`: AES

NAME

ulimit - get and set user limits

SYNOPSIS

```
#include <ulimit.h>

long ulimit(int cmd, ...);
```

DESCRIPTION

`ulimit()` provides for control over process limits. Available values for *cmd* are:

`UL_GETFSIZE` Get the file size limit of the process. The limit is in units of 512-byte blocks and is inherited by child processes. Files of any size can be read. The optional second argument is not used.

`UL_SETFSIZE` Set the file size limit of the process to the value of the optional second argument which is taken as a long. Any process can decrease this limit, but only a process with an effective user ID of super-user can increase the limit. Note that the limit must be specified in units of 512-byte blocks.

`UL_GETMAXBRK` Get the maximum possible break value (see *brk*(2)). Depending on system resources such as swap space, this maximum might not be attainable at a given time. The optional second argument is not used.

ERRORS

`ulimit()` fails if one or more of the following conditions is true.

[EINVAL] *cmd* is not in the correct range.

[EPERM] `ulimit()` fails and the limit is unchanged if a process with an effective user ID other than super-user attempts to increase its file size limit.

RETURN VALUE

Upon successful completion, a non-negative value is returned. Errors return a -1, with `errno` set to indicate the error.

SEE ALSO

brk(2), write(2).

STANDARDS CONFORMANCE

`ulimit()`: AES, SVID2, XPG2, XPG3, XPG4

NAME

umask - set and get file creation mask

SYNOPSIS

#include <sys/stat.h>

mode_t umask(mode_t cmask);

DESCRIPTION

umask() sets the process's file mode creation mask to **umask()** and returns the previous value of the mask. Only the file access permission bits of the masks are used.

The bits set in *cmask* specify which permission bits to turn off in the mode of the created file, and should be specified using the symbolic values defined in *stat*(5).

EXAMPLES

The following creates a file named **path** in the current directory with permissions **S_IRWXU|S_IRGRP|S_IXGRP**, so that the file can be written only by its owner, and can be read or executed only by the owner or processes with group permission, even though group write permission and all permissions for others are passed in to **creat()**.

```
#include <sys/types.h>
#include <sys/stat.h>

int fildes;

(void) umask(S_IWGRP|S_IRWXO);
fildes = creat("path", S_IRWXU|S_IRWXG|S_IRWXO);
```

RETURN VALUE

The previous value of the file mode creation mask is returned.

SEE ALSO

mkdir(1), sh(1), mknod(1M), chmod(2), creat(2), mknod(2), open(2).

STANDARDS CONFORMANCE

umask(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

umount - unmount a file system

SYNOPSIS

```
#include <sys/mount.h>

int umount(const char *name);
```

DESCRIPTION

`umount()` requests that a previously mounted file system contained on the block special device identified by *name* be unmounted. *name* is a pointer to a path name. After unmounting the file system, the directory upon which the file system was mounted reverts to its ordinary interpretation.

`umount()` can also request that a file system mounted previously on the directory identified by *name* be unmounted. After unmounting the file system, *name* reverts to its ordinary interpretation.

`umount()` can be invoked only by the user with the appropriate privilege.

NETWORKING FEATURES

NFS

path must indicate a directory name when unmounting an NFS file system.

RETURN VALUE

If successful, `umount()` returns a value of `0`. Otherwise, it returns a value of `-1` and sets `errno` to indicate the error.

ERRORS

`umount()` fails if one or more of the following are true:

[EPERM] The effective user ID of the process is not that of a user with appropriate privileges.

[ENOENT] *name* does not exist.

[ENOTBLK] *name* is not a block special device.

[EINVAL] *name* is not mounted.

[EBUSY] A file on *name* is busy.

[EFAULT] *name* points outside the allocated address space of the process. Reliable detection of this error is implementation dependent.

[ENXIO] The device associated with *name* does not exist.

[ENOTDIR] A component of *name* is not a directory.

[ENOENT] *name* is null.

[ENAMETOOLONG] *name* exceeds **PATH_MAX** bytes, or a component of *name* exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect.

[EACCES] A component of the path prefix of *name* denies search permission.

[ELOOP] Too many symbolic links were encountered in translating the path name.

WARNINGS

If `umount()` is called from the program level (that is, not from the *mount*(1M) level), the table of mounted devices contained in `/etc/mnttab` is not updated automatically. Updating of `/etc/mnttab` is performed by the `mount` and `syncer` commands (see *mount*(1M) and *syncer*(1M) for more information).

DEPENDENCIES

HP Clustered Environment:

When `umount()` is called from a client node and *path* refers to a directory on which is mounted a UFS file system (as opposed to an NFS file system; see *vfsmount*(2)), an EINVAL error is returned. This behavior is subject to change in future releases, and its use in applications is not recommended.

SEE ALSO

mount(1M), syncer(1M), mount(2), vfsmount(2).

STANDARDS CONFORMANCE
umount ( ): SVID2, XPG2

NAME

uname, setuname - get/set name of current HP-UX system

SYNOPSIS

```
#include <sys/utsname.h>

int uname(struct utsname *name);

int setuname(const char *name, size_t namelen);
```

DESCRIPTION

`uname()` stores information identifying the current HP-UX system in the structure pointed to by *name*.

`uname()` uses the structure defined in `<sys/utsname.h>` whose members are:

```
#define UTSLEN 9
#define SNLEN 15

char sysname[UTSLEN];
char nodename[UTSLEN];
char release[UTSLEN];
char version[UTSLEN];
char machine[UTSLEN];
char idnumber[SNLEN];
```

`uname()` returns a null-terminated string in each field. The `sysname` field contains `HP-UX`. Similarly, the `nodename` field contains the name by which the system is known on a communications network. The `release` field contains the release number of the operating system, such as `8.0` or `8.0.1`. The `version` field contains additional information about the operating system. The first character of the `version` field is set to:

Character	Series 700/800	Series 300/400
A	two-user system	two-user system
B	16-user system	unlimited-users system
C	32-user system	
D	64-user system	
E	8-user system	
U	unlimited-users system	

(Note that the contents of the version field might change on future releases as AT&T license agreement restrictions change.) The `machine` field contains a standard name that identifies the hardware on which the HP-UX system is running. The `idnumber` is a unique identification number within that class of hardware, possibly a hardware or software serial number. This field returns the null string to indicate the lack of an identification number.

`setuname()` sets the `nodename` field in the `utsname` structure to *name*, which has a length of *namelen* characters. This is usually executed by `/etc/rc` at system boot time. Names are limited to `UTSLEN` - 1 characters; `UTSLEN` is defined in `<sys/utsname.h>`.

ERRORS

[EPERM] `setuname()` was attempted by a user lacking the appropriate privileges.

[EFAULT] *name* points to an illegal address. The reliable detection of this error is implementation dependent.

RETURN VALUE

Upon successful completion, a non-negative value is returned. Otherwise, -1 is returned and `errno` is set to indicate the error.

AUTHOR

`uname()` was developed by AT&T and HP.

SEE ALSO

hostname(1), uname(1), gethostname(2), sethostname(2), privilege(5).

STANDARDS CONFORMANCE

`uname()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

unlink - remove directory entry; delete file

SYNOPSIS

```
#include <unistd.h>

int unlink(const char *path);
```

DESCRIPTION

`unlink()` removes the directory entry named by the path name pointed to by *path*.

When all links to a file have been removed and no process has the file open, the space occupied by the file is freed and the file ceases to exist. If one or more processes have the file open when the last link is removed, only the directory entry is removed immediately so that processes that do not already have the file open cannot access the file. After all processes close their references to the file, if there are no more links to the file, the space occupied by the file is then freed and the file ceases to exist.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

ERRORS

The named file is unlinked unless one or more of the following are true:

[EACCES] Search permission is denied for a component of the path prefix.

[EACCES] Write permission is denied on the directory containing the link to be removed.

[ENOTDIR] A component of the path prefix is not a directory.

[ENOENT] The named file does not exist (for example, *path* is null or a component of *path* does not exist).

[EPERM] The named file is a directory and the effective user ID of the process is not a user with appropriate privileges.

[EBUSY] The entry to be unlinked is the mount point for a mounted file system.

[ETXTBSY] The entry to be unlinked is the last link to a pure procedure (shared text) file that is being executed.

[EROFS] The directory entry to be unlinked is part of a read-only file system.

[EFAULT] *path* points outside the process's allocated address space. The reliable detection of this error is implementation dependent.

[ENAMETOOLONG] The length of the specified path name exceeds **PATH_MAX** bytes, or the length of a component of the path name exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect.

[ELOOP] Too many symbolic links were encountered in translating the path name.

WARNINGS

If `unlink()` is used on a directory that is *not* empty (contains files other than `.` and `..`), the directory is unlinked, the files become orphans, and the directory link count is left with an inaccurate value unless they are linked by some other directory.

If `unlink()` is used on a directory that *is* empty (contains only the files `.` and `..`), the directory is unlinked, but the parent directory's link count is left with an inaccurate value.

In either of the above cases, the file system should be checked using `fsck` (see *fsck*(1M)). To avoid these types of problems, use `rmdir()` instead (see *rmdir*(2)).

SEE ALSO

rm(1), close(2), link(2), open(2), rmdir(2), privilege(5).

STANDARDS CONFORMANCE

`unlink()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

ustat - get file system statistics

SYNOPSIS

```
#include <ustat.h>

int ustat(dev_t dev, struct ustat *buf);
```

DESCRIPTION

`ustat()` returns information about a mounted file system. *dev* is a device number identifying a device containing a mounted file system. *buf* is a pointer to a `ustat` structure (defined in `<ustat.h>`) that includes the following elements:

```
daddr_t  f_tfree;        /* Total free blocks */
ino_t    f_tinode;       /* Number of free inodes */
char     f_fname[6];     /* Filsys name */
char     f_fpack[6];     /* Filsys pack name */
int      f_blksize;      /* Block size */
```

The values of the `f_tfree` and `f_blksize` fields are reported in fragment size units.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

ERRORS

`ustat()` fails if one or more of the following is true:

[EINVAL] *dev* is not the device number of a device containing a mounted file system.

[EFAULT] *buf* points outside the process's allocated address space. The reliable detection of this error is implementation dependent.

AUTHOR

`ustat()` was developed by AT&T and HP.

SEE ALSO

touch(1), stat(2), fs(4).

STANDARDS CONFORMANCE

`ustat()`: SVID2, XPG2

NAME

utime - set file access and modification times

SYNOPSIS

```
#include <utime.h>

int utime(const char *path, const struct utimbuf *times);
```

DESCRIPTION

`utime()` sets the access and modification times of the file to which the *path* argument refers.

If *times* is a null pointer, the access and modification times of the file are set to the current time. A process must be the owner of the file or have write permission on the file to use `utime()` in this manner.

If *times* is not a null pointer, *times* is interpreted as a pointer to a *utimbuf* structure and the access and modification times are set to the values contained in the designated structure. Only the owner of the file or users having appropriate privileges can use `utime()` this way.

The following times in the `utimbuf` structure defined in <utime.h> are measured in seconds since 00:00:00 UTC (Universal Coordinated Time), Jan. 1, 1970.

```
time_t actime;      /* access time */
time_t modtime;     /* modification time */
```

RETURN VALUE

Upon successful completion, a value of `0` is returned. Otherwise, a value of `-1` is returned and `errno` is set to indicate the error.

ERRORS

`utime()` fails if one or more of the following is true:

[ENOENT] The named file does not exist.

[ENOTDIR] A component of the path prefix is not a directory.

[EACCES] Search permission is denied by a component of the path prefix.

[EPERM] The effective user ID is not a user with appropriate privileges. and not the owner of the file, and *times* is not a null pointer.

[EACCES] The effective user ID is not a user with appropriate privileges, and not the owner of the file, *times* is a null pointer, and write access is denied.

[EROFS] The file system containing the file is mounted read-only.

[EFAULT] *times* is not a null pointer, and points outside the process's allocated address space. The reliable detection of this error is implementation dependent.

[EFAULT] *path* points outside the process's allocated address space. The reliable detection of this error is implementation dependent.

[ENAMETOOLONG] The length of the specified path name exceeds **PATH_MAX** bytes, or the length of a component of the path name exceeds **NAME_MAX** bytes while `_POSIX_NO_TRUNC` is in effect.

DEPENDENCIES

NFS: `utime()` may return EPERM when invoked on a remote file owned by the super-user, even if the invoking user has write permission on the file.

SEE ALSO

touch(1), stat(2).

STANDARDS CONFORMANCE

`utime()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

vfork - spawn new process; share virtual memory

SYNOPSIS

```
#include <unistd.h>

pid_t vfork(void);
```

REMARKS

`vfork()` is a higher performance version of `fork()` that is provided on some systems where a performance advantage can be attained.

`vfork()` differs from `fork()` only in that the child process can share code and data with the calling process (parent process). This speeds cloning activity significantly at a risk to the integrity of the parent process if `vfork()` is misused.

The use of `vfork()` for any purpose except as a prelude to an immediate `exec()` or `exit()` is not supported. Any program that relies upon the differences between `fork()` and `vfork()` is not portable across HP-UX systems.

All HP-UX implementations must provide the entry `vfork()`, but it is permissible for them to treat it identically to `fork`. On some implementations the two are not distinguished because the `fork()` implementation is as efficient as possible. Other versions may do the same to avoid the overhead of supporting two similar calls.

DESCRIPTION

`vfork()` can be used to create new processes without fully copying the address space of the old process. If a forked process is simply going to do an `exec()` (see *exec*(2)), the data space copied from the parent to the child by `fork()` is not used. This is particularly inefficient in a paged environment, making `vfork` is particularly useful. Depending upon the size of the parent's data space, `vfork()` can give a significant performance improvement over `fork()`.

`vfork()` differs from `fork()` in that the child borrows the parent's memory and thread of control until a call to `exec()` or an exit (either by a call to `exit()` or abnormally (see *exec*(2) and *exit*(2)). The parent process is suspended while the child is using its resources.

`vfork()` returns 0 in the child's context and (later) the pid of the child in the parent's context.

`vfork()` can normally be used just like `fork()`. It does not work, however, to return while running in the child's context from the procedure which called `vfork()` since the eventual return from `vfork()` would then return to a no longer existent stack frame. Be careful, also, to call `_exit()` rather than `exit()` if you cannot `exec()`, since `exit()` flushes and closes standard I/O channels, thereby damaging the parent process's standard I/O data structures. (Even with `fork()` it is wrong to call `exit()` since buffered data would then be flushed twice.)

The [`vfork,exec`] window begins at the `vfork()` call and ends when the child completes its `exec()` call.

RETURN VALUE

Upon successful completion, `vfork()` returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of −1 is returned to the parent, no child process is created, and `errno` is set to indicate the error.

ERRORS

`vfork()` fails and no child process is created if any of the following conditions are encountered:

[EAGAIN]	The system-wide limit on the total number of processes under execution would be exceeded.
[EAGAIN]	The system-imposed limit on the total number of processes under execution by a single user would be exceeded.

DEPENDENCIES

Series 800

Process times for the parent and child processes within the [`vfork,exec`] window may be inaccurate.

Parent and child processes share the same stack space within the [`vfork,exec`] window. If the size of the stack has been changed within this window by the child process (return from or call to a

function, for example), it is likely that the parent and child processes will be killed with signal **SIGSEGV** or **SIGBUS**.

In the [**vfork,exec**] window, a call to **signal()** (see *signal*(2) that installs a catching function can affect handling of the signal by the parent. The parent is not affected if the handling is being set to **SIG_DFL** or **SIG_IGN**, or if either **sigaction()** or **sigvector()** is used (see *sigaction*(2) and *sigvector*(2)).

AUTHOR

vfork() was developed by the University of California, Berkeley.

SEE ALSO

exec(2), exit(2), fork(2), wait(2).

NAME

vfsmount - mount a file system

SYNOPSIS

```
#include <sys/mount.h>

int vfsmount(
    int type,
    const char *dir,
    int flags,
    caddr_t data
);
```

DESCRIPTION

vfsmount() attaches a file system to a directory. After a successful return, references to directory *dir* refer to the root directory of the newly mounted file system. *dir* is a pointer to a null-terminated string containing a path name. *dir* must exist already, and must be a directory. *dir* cannot be a context-dependent file (see *cdf*(4)). Its old contents are inaccessible while the file system is mounted. **vfsmount()** differs from **mount()** (see *mount*(2)) in its ability to mount file system types other than just the UFS type.

type indicates the type of the file system. It must be one of the types described below. **vfsmount()** does not check that the file system is actually of type *type*; if *type* is incorrect, **vfsmount()** may cause the process to hang. To prevent such problems, **statfsdev()** (see *statfsdev*(3c)) should be called before **vfsmount()** to check the file system type, which **statfsdev()** places in the **f_fsid[1]** field of the **statfs** structure it returns.

The *flags* argument determines whether the file system can be written to (functionally identical to the **rwflag** argument in *mount*(2) in this regard). It also controls whether programs from the mounted file system are allowed to have set-uid execution. Physically write-protected and magnetic tape file systems must be mounted read-only. Failure to do so results in a return of -1 by **vfsmount()** and a value of EIO in **errno**. The following values for the *flags* argument are defined in **<sys/mount.h>**:

M_RDONLY Mount done as read-only.

M_NOSUID Execution of set-uid programs not permitted.

data is a pointer to a structure containing arguments specific to the value contained in *type*. The following values for *types* are defined in **<sys/mount.h>**:

MOUNT_UFS Mount a local HFS file system. *data* points to a structure of the following format:

```
struct ufs_args {
    char    *fspec;
};
```

fspec points to the name of the block special file that is to be mounted. This is identical in use and function to the first argument for *mount*(2).

MOUNT_CDFS Mount a local CD-ROM file system. *data* points to a structure of the following format:

```
struct cdfs_args {
    char    *fspec;
};
```

fspec points to the name of the block special file that is to be mounted.

NETWORKING FEATURES

NFS

An additional value for the *type* argument is supported.

MOUNT_NFS Mount an NFS file system. *data* points to a structure of the following format:

```
#include <nfs/nfs.h>
#include <netinet/in.h>
```

```
struct nfs_args {
    struct sockaddr_in  *addr;
    fhandle_t           *fh;
    int         flags;
    int         wsize;
    int         rsize;
    int         timeo;
    int         retrans;
    char        *hostname;
    int         acregmin;
    int         acregmax;
    int         acdirmin;
    int         acdirmax;
};
```

Elements in the structure as as follows:

addr Points to a local socket address structure (see *inet*(7)), which is used by the system to communicate with the remote file server.

fh Points to a structure containing a **file handle**, an abstract data type that is used by the remote file server when serving an NFS request.

flags Bit map that sets options and indicates which of the following fields contain valid information. The following values of the bits are defined in <**nfs/nfs.h**>:

NFSMNT_SOFT Specify whether the mount is a soft mount or a hard mount. If set, the mount is soft and will cause requests to be retried **retrans** number of times. Otherwise, the mount is hard and requests will be tried forever.

NFSMNT_WSIZE Set the write size.

NFSMNT_RSIZE Set the read size.

NFSMNT_TIMEO Set the initial timeout value.

NFSMNT_RETRANS Set the number of request retries.

NFSMNT_HOSTNAME Set a hostname.

NFSMNT_INT Set the option to have interruptible I/O to the mounted file system.

NFSMNT_NODEVS Set the option to deny access to local devices via NFS device files. By default, access to local devices via NFS device files is allowed.

NFSMNT_IGNORE Mark the file system type as **ignore** in **/etc/mnttab**.

NFSMNT_NOAC Turn off attribute caching. By default NFS caches attributes of files and directories to speed up operations on NFS files by not always getting the attributes from the server. Names are also cached to speed up path name lookup. However it does allow modifications to files on the server to not be immediately detectable on the clients. Setting **NFSMNT_NOAC** turns off attribute caching and name lookup caching. NFS caches attributes for a length of time proportional to how much time has elapsed since the last modification. The time length is subject to **acregmin**, **acregmax**, **acdirmin**, and **acdirmax** described below.

NFSMNT_NOCTO Cached attributes are flushed when a NFS file is opened unless this option is specified. This option is useful where it is known that the files will not be changing as is the case for a CD-ROM drive.

NFSMNT_ACREGMIN
Use the **acregmin** value. See **acregmin** below.

NFSMNT_ACDIRMIN
Use the **acdirmin** value. See **acdirmin** below.

NFSMNT_ACREGMAX
Use the **acregmax** value. See **acregmax** below.

NFSMNT_ACDIRMAX
Use the **acdirmax** value. See **acdirmax** below.

wsize
Can be used to advise the system about the maximum number of data bytes to use for a single outgoing protocol (such as UDP) message. This value must be greater than 0. Default **wsize** is **8192**.

rsize
Can be used to advise the system about the maximum number of data bytes to use for a single incoming protocol (such as UDP) message. This value must be greater than 0. Default **rsize** is **8192**.

timeo
Can be used to advise the system on the time to wait between NFS request retries. This is in units of 0.1 seconds. This value must be greater than 0. Default **timeo** is **7**.

retrans
Can be used to advise the system about the number of times the system will resend a request. This value must be 0 or greater. Default **retrans** is **4**.

hostname
A name for the file server that can be used when any messages are given concerning the server. The string can be of length from 0 to 32 characters.

acregmin
can be used to advise the system the minimum number of seconds to cache attributes for a non-directory file. If this number is less than 0, it means to use the system defined maximum of 3600 seconds. The number specified can not be 0. If the number is greater than 3600, 3600 will be used. Default **acregmin** is **3**. is ignored if **NFSMNT_NOAC** is specified.

acdirmin
can be used to advise the system the minimum number of seconds to cache attributes for a directory. If this number is less than 0, it means to use the system defined maximum of 3600 seconds. The number specified can not be 0. If the number is greater than 3600, 3600 will be used. Default **acdirmin** is **30**. **acdirmin** is ignored if **NFSMNT_NOAC** is specified.

acregmax
can be used to advise the system the maximum number of seconds to cache attributes for a non-directory file. If this number is less than 0, it means to use the system defined maximum of 36000 seconds. The number specified cannot be 0. If the number is greater than 36 000, 36 000 is used. Default **acregmax** is **60**. **acregmax** is ignored if **NFSMNT_NOAC** is specified.

acdirmax can be used to advise the system the maximum number of seconds to cache attributes for a directory. If this number is less than 0, it means to use the system defined maximum of 36 000 seconds. The number specified cannot be 0. If the number is greater than 36 000, 36 000 will be used. Default **acdirmax** is **60**. **acdirmax** is ignored if **NFSMNT_NOAC** is specified.

RETURN VALUE

Upon successful completion, **vfsmount()** returns a value of **0**. Otherwise, no file system is mounted, a value of **-1** is returned, and **errno** is set to indicate the error.

ERRORS

vfsmount() fails when one of the following occurs:

[EBUSY] *dir* is not a directory, or another process currently holds a reference to it.

[EBUSY] No space remains in the mount table.

[EBUSY] The super block for the file system had a bad magic number or an out-of-range block size.

[EBUSY] Not enough memory was available to read the cylinder group information for the file system.

[EFAULT] *data* or *dir* points outside the allocated address space of the process.

[EINVAL] *dir* is a context-dependent file (see *cdf*(4).

[EIO] An I/O error occurred while reading from or writing to the file system.

[EIO] An attempt was made to mount a physically write protected or magnetic tape file system as read-write.

[ELOOP] Too many symbolic links were encountered while translating the path name of file system referred to by *data* or *dir*.

[ENAMETOOLONG] The path name of the file system referred to by *data* or *dir* is longer than **PATH_MAX** bytes, or the length of a component of the path name exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect.

[ENOENT] The file system referred to by *data* or *dir* does not exist.

[ENOENT] The file system referred to by *data* does not exist.

[ENOTBLK] The file system referred to by *data* is not a block device. This message can occur only during a local mount.

[ENOTDIR] A component of the path prefix in *dir* is not a directory.

[ENOTDIR] A component of the path prefix of the file system referred to by *data* or *dir* is not a directory.

[ENXIO] The major device number of the file system referred to by *data* is out of range (indicating that no device driver exists for the associated hardware).

[EOPNOTSUPP] **vfsmount()** of a remote device was attempted.

[EPERM] The caller does not have appropriate privileges.

DEPENDENCIES

NFS: **vfsmount()** fails when one of the following occurs, and returns the error indicated:

[EFAULT] A pointer in the *data* structure points outside the process's allocated address space.

[EINVAL] A value in a field of *data* is out of proper range.

[EREMOTE] An attempt was made to remotely mount a file system that was already mounted from another remote node.

See *getfh*(2), *inet*(7), and *mountd*(1M) for more information.

HP Clustered Environment:

vfsmount() of a local CDFS file system (**MOUNT_CDFS**) is not supported from a cluster client. Such a call returns an EINVAL error.

WARNINGS

Use of *mount*(1M) is preferred over **vfsmount()** because *mount*(1M) supports all mounting options that are available from **vfsmount()** directly, plus *mount*(1M) also maintains the **/etc/mnttab** file which lists what file systems are mounted.

In the HP Clustered environment, the *spec* and *dir* arguments should always be fully expanded pathnames.

AUTHOR

vfsmount() was developed by HP and Sun Microsystems, Inc.

SEE ALSO

mount(2), umount(2), mount(1M).

NAME

wait, waitpid, wait3 - wait for child or traced process to stop or terminate

SYNOPSIS

```
#include <sys/wait.h>

pid_t wait(int *stat_loc);

pid_t waitpid(pid_t pid, int *stat_loc, int options);

pid_t wait3(int *stat_loc, int options, int *reserved);
```

DESCRIPTION

wait() suspends the calling process until one of the immediate children terminates or until a process being traced stops because that traced process has hit a break point. A process being traced can be either a child or a process attached by the **ptrace()** request **PT_ATTACH** (see *ptrace*(2)). The **wait()** system call returns prematurely if a signal is received. If a child or traced process stops or terminates prior to the call on **wait**, return is immediate.

If *stat_loc* is not a null pointer, status information is stored in the location pointed to by *stat_loc*. The status can be used to differentiate between stopped and terminated processes. If the process terminates, the status identifies the cause of termination and passes useful information to the calling process. This is accomplished using the following macros defined in **<sys/wait.h>**, with the status value stored at **stat_loc* as an argument:

WIFEXITED(*stat_val*) If the process terminated because of an **exit()** or **_exit()** system call, this macro evaluates to a non-zero value.

WEXITSTATUS(*stat_val*) If the value of **WIFEXITED**(*stat_val*) is non-zero, this macro evaluates to the low-order 8 bits of the argument that the process passed to **exit()** or **_exit()** (see *exit*(2)).

WIFSIGNALED(*stat_val*) If the process terminated due to the default action of a signal (see *signal*(5)), this macro evaluates to a non-zero value.

WTERMSIG(*stat_val*) If the value of **WIFSIGNALED**(*stat_val*) is non-zero, this macro evaluates to the number of the signal that caused the termination.

WCOREDUMP(*stat_val*) If the value of **WIFSIGNALED**(*stat_val*) is non-zero, this macro evaluates to a non-zero value if a "core image" was produced (see *signal*(5)).

WIFSTOPPED(*stat_val*) If the process is stopped, this macro evaluates to a non-zero value.

WSTOPSIG(*stat_val*) If the value of **WIFSTOPPED**(*stat_val*) is non-zero, this macro evaluates to the number of the signal that caused the process to stop.

As a single special case, the value stored in **stat_loc* is zero if and only if status is being returned from a terminated process that called **exit()** or **_exit()** with a value of zero.

If the information stored at the location pointed to by *stat_loc* was stored there by a call to one of the **wait()** functions, exactly one of the macros **WIFEXITED**(**stat_loc*), **WIFSIGNALED**(**stat_loc*), or **WIFSTOPPED**(**stat_loc*) evaluates to a non-zero value.

The **waitpid()** function behaves identically to **wait()** if *pid* has a value of **-1** and *options* has a value of zero. Otherwise its behavior is modified by the values of the *pid* and *options* arguments.

The *pid* argument specifies the set of processes for which status is requested. *waitpid* returns only the status of a child process from this set.

- If *pid* is equal to **-1**, status is requested for any child process or attached process. In this respect, **waitpid()** is then equivalent to **wait()**.
- If *pid* is greater than zero, it specifies the process ID of a single child or attached process for which status is requested.
- If *pid* is equal to zero, status is requested for any child or attached process whose process group ID is equal to that of the calling process.

- If *pid* is less than `-1`, status is requested for any child or attached process whose process group ID is equal to the absolute value of *pid*.

The *options* argument is constructed from the bit-wise inclusive OR of zero or more of the following flags:

`WNOHANG` If this flag is set, `waitpid()` or `wait3()` is prevented from suspending the calling process. A value of zero is returned indicating that no child or traced processes have stopped or died.

`WUNTRACED` If and only if this flag is set, `waitpid()` or `wait3()` returns information on child or attached processes that are stopped but not traced (with *ptrace*(2)) because they received a `SIGTTIN`, `SIGTTOU`, `SIGTSTP`, or `SIGSTOP` signal, and whose status has not yet been reported. Regardless of this flag, status is returned for child or attached processes that have terminated or are stopped and traced and whose status has not yet been reported.

Calling `wait3()` is equivalent to calling `waitpid()` with the value of *pid* equal to zero. The third parameter to `wait3()` is currently unused and must always be a null pointer.

If a parent process terminates without waiting for its child processes to terminate, the parent process ID of each child process is set to 1. This means the initialization process inherits the child processes.

Notes

Earlier HP-UX versions documented the bit encodings of the status returned by `wait()` rather than the macros `WIFEXITED`, `WEXITSTATUS`, `WIFSIGNALED`, `WTERMSIG`, `WCOREDUMP`, `WIFSTOPPED`, and `WSTOPSIG`. Applications using those bit encodings will continue to work correctly. However, new applications should use the macros for maximum portability.

In earlier HP-UX versions, the macros `WIFSTOPPED`, `WIFSIGNALED`, and `WIFEXITED` have the same definitions as the correspondingly named macros in the BSD 4.3 and earlier systems. Existing applications that depend on these definitions will continue to work correctly. However, if the application is recompiled, the feature test macro `_BSD` must be turned on for the compilation so that the old definitions of these macros are obtained. New definitions of these macros are in effect by default. The only difference between the old and new definitions is the type of the argument. Type `union wait` is used in the BSD definitions while type `int` is used in the default definitions.

ERRORS

`wait()` fails if one or more of the following is true:

[ECHILD] The calling process to `wait()` or `wait3()` has no existing child or traced processes, or the calling process to `waitpid()` has no existing unwaited-for child or traced processes that match the *pid* argument.

[ECHILD] For `waitpid()`, the process or process group specified by *pid* does not exist or is not a child of the calling process.

[EFAULT] *stat_loc* points to an illegal address. The reliable detection of this error is implementation dependent.

[EINVAL] The *options* argument to `waitpid()` or `wait3()` is invalid.

[EINVAL] `wait3()` was passed a non-null pointer value for its third argument.

[EINTR] The function was interrupted by a signal. The value of the location pointed to by *stat_loc* is undefined.

RETURN VALUE

If `wait()` returns due to the receipt of a signal, a value of `-1` is returned to the calling process and `errno` is set to EINTR. If `wait()` returns due to a stopped or terminated child or traced process, the process ID of that process is returned to the calling process. If *waitpid()* or *wait3()* is called, the WNOHANG option is used, and there are no stopped or terminated child or traced processes (as specified by *pid* in the case of *waitpid()*), a value of zero is returned. Otherwise, a value of `-1` is returned and `errno` is set to indicate the error.

WARNINGS

The behavior of `wait()`, `waitpid()`, and `wait3()` is affected by setting the `SIGCLD` signal to `SIG_IGN`. See WARNINGS section of *signal*(5). Signal handlers that cause system calls to be restarted can

affect the EINTR condition described above (see *sigaction*(2), *sigvector*(2), and *bsdproc*(2)).

AUTHOR

`wait()`, `waitpid()`, and `wait3()` were developed by HP, AT&T, and the University of California, Berkeley.

SEE ALSO

Exit conditions (`$?`) in sh(1), exec(2), exit(2), fork(2), pause(2), ptrace(2), signal(5).

STANDARDS CONFORMANCE

`wait()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

`waitpid()`: AES, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

write, writev - write on a file

SYNOPSIS

```
#include <unistd.h>

ssize_t write(int fildes, const void *buf, size_t nbyte);

#include <sys/uio.h>

ssize_t writev(
    int fildes,
    const struct iovec *iov,
    size_t iovcnt
);
```

DESCRIPTION

`write()` attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the file descriptor *fildes*. `writev()` performs the same action, but gathers the output data from the `iovlen` buffers specified by the elements of the `iovec` array: `iov`[0], `iov`[1], ..., `iov`[*iovcnt*–1].

The `iovec` structure for `writev()` is defined as follows:

```
struct iovec {
     caddr_t  iov_base;
     int      iov_len;
};
```

Each `iovec` entry specifies the base address and length of an area in memory from which data should be copied. The `iovec` array can be at most `MAXIOV` long.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file offset. Upon return from `write()`, the file offset is incremented by the number of bytes actually written.

On devices incapable of seeking, writing always takes place starting at the device's current position. The value of a file offset associated with such a device is undefined.

If the `O_APPEND` file status flag is set, the file offset is set to the end of the file prior to each write.

For ordinary files, if the `O_SYNC` flag of the file status flags is set, the write does not return until both the file data and the file status are physically updated. For block special files, if `O_SYNC` is set, the write does not return until the data is physically updated. How the data reaches the physical media is implementation- and hardware-dependent.

If the number of bytes requested by `write()` exceeds the allotted capacity (see *ulimit*(2)) or the physical end of a medium, only the allotted number of bytes are actually written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512 bytes returns 20. The next write of a non-zero number of bytes fails (except as noted below).

A write to an ordinary file is prevented if enforcement-mode file and record locking is set, and another process owns a lock on the segment of the file being written:

If `O_NDELAY` or `O_NONBLOCK` is set, the write returns –1 and sets `errno` to EAGAIN.

If `O_NDELAY` and `O_NONBLOCK` are clear, the write does not complete until the blocking record lock is removed.

If the file being written is a pipe (or FIFO), the system-dependent maximum number of bytes that it can store is given by `PIPSIZ` (defined in `<sys/inode.h>`). The minimum value of `PIPSIZ` on any HP-UX system is 8192. When writing a pipe, the following conditions apply:

If the `O_NDELAY` or `O_NONBLOCK` file status flag is set:

If *nbyte* is less than or equal to `PIPSIZ` and sufficient room exists in the pipe or `FIFO`, the `write()` succeeds and returns the number of bytes written;

If *nbyte* is less than or equal to `PIPSIZ` but insufficient room exists in the pipe or `FIFO`, the `write()` returns having written nothing. If `O_NONBLOCK` is set, –1 is returned and `errno`

is set to EAGAIN. If O_NDELAY is set, 0 is returned.

If *nbyte* is greater than PIPSIZ and the pipe or FIFO is full, the write returns having written nothing. If O_NONBLOCK is set, –1 is returned and errno is set to EAGAIN. If O_NDELAY is set, 0 is returned.

If *nbyte* is greater than PIPSIZ, and some room exists in the pipe or FIFO, as much data as fits in the pipe or FIFO is written, and write() returns the number of bytes actually written, an amount less than the number of bytes requested.

If the O_NDELAY and O_NONBLOCK file status flags are clear:

The write() always executes correctly (blocking as necessary), and returns the number of bytes written.

If write() is interrupted by a signal after it successfully writes some data, it returns the number of bytes written before the interrupt occurred. If write() is interrupted before any bytes are written, write() returns –1 and sets errno to EINTR.

write() clears the SUID, SGID, and sticky bits on all non-directory type files if the write is performed by any user other than the owner or a user who has appropriate privileges. For directories, write() does not clear the SUID, SGID, and sticky bits.

RETURN VALUE

Upon successful completion, the number of bytes actually written is returned. Otherwise, –1 is returned and errno is set to indicate the error.

ERRORS

write() fails and the file offset remains unchanged if any of the following conditions is true:

[EBADF] *fildes* is not a valid file descriptor open for writing.

[EPIPE and SIGPIPE signal]
An attempt is made to write to a pipe that is not open for reading by any process.

[EINTR] A signal was caught before any data was transferred (see *sigvector*(2).

[EDEADLK] A resource deadlock would occur as a result of this operation (see *lockf*(2) and *fcntl*(2)).

[EDQUOT] User's disk quota block limit has been reached for this file system.

[EAGAIN] Enforcement-mode file and record locking was set, O_NDELAY was set, and there was a blocking record lock.

[ENOLCK] The system record lock table is full, preventing the write from sleeping until the blocking record lock is removed.

[EIO] The process is in a background process group and is attempting to write to its controlling terminal, TOSTOP is set, the process is neither ignoring or blocking the SIGTTOU signal, and the process group of the process is orphaned.

[EIO] An I/O error occurred while writing to the device corresponding to *fildes*.

[ENOSPC] Not enough space on the file system.

In addition, writev() might return one of the following errors:

[EFAULT] iov_base or iov points outside of the allocated address space. The reliable detection of this error is implementation dependent.

[EINVAL] iovcnt is less than or equal to 0, or greater than MAXIOV.

[EINVAL] One of the iov_len values in the iov array was negative.

[EINVAL] The sum of iov_len values in the iov array overflowed a 32-bit integer.

write() or writev() fails, the file offset is updated to reflect the amount of data transferred, and errno is set accordingly if one of the following conditions is true:

[EFBIG] An attempt was made to write a file that exceeds the process's file size limit or the maximum file size. See *ulimit*(2).

[EFAULT] *buf* points outside the process's allocated address space. The reliable detection of this error is implementation dependent.

EXAMPLES

Assuming a process opened a file for writing, the following call to *write*(2) attempts to write *mybufsize* bytes to the file from the buffer to which *mybuf* points.

```
#include <string.h>

int mybufsize, nbytes, fildes;
char *mybuf = "aeiou and sometimes y";
mybufsize = strlen (mybuf);
nbytes = write (fildes, mybuf, mybufsize);
```

WARNINGS

Check all references to *signal*(5) for appropriateness on systems that support *sigvector*(2). *sigvector*(2) can affect the behavior described on this page.

Character special devices, and raw disks in particular, apply constraints on how `write()` can be used. See specific Section (7) manual entries for details on particular devices.

AUTHOR

`write()` was developed by HP, AT&T, and the University of California, Berkeley.

SEE ALSO

creat(2), dup(2), lockf(2), lseek(2), open(2), pipe(2), ulimit(2), ustat(2).

STANDARDS CONFORMANCE

`write()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

Section 3:
Library Functions

NAME

intro - introduction to subroutines and libraries

SYNOPSIS

```
#include <stdio.h>

#include <math.h>
```

DESCRIPTION

This section describes functions found in various libraries, other than those functions that directly invoke HP-UX system primitives, which are described in Section (2) of this volume. Certain major collections are identified by a letter after the section identifier (3):

(3C) These functions, together with the Operating System Calls and those marked (3S), constitute the Standard C Library which is automatically loaded by the C compiler, *cc*(1). The link editor *ld*(1) searches this library if the **-lc** option is specified. Declarations for some of these functions can be obtained from **#include** files indicated in the appropriate entries.

(3G) These functions constitute the graphics library, and are documented in separate manuals.

(3I) These functions constitute the instrument support (Device I/O) library.

(3M) These functions constitute the Math Libraries, **libm.a** and **libM.a**. All of the functions are in both libraries except for **matherr** (see *matherr*(3M) for more details). The HP-UX operating system provides two different libraries due to to conflicts between Issue 2 of the SVID specification and the ANSI C standard. If behavior conforming to SVID Issue 2 is desired, **libm.a** should be used. If behavior conforming to the ANSI C standard is desired, **libM.a** should be used. The **libm.a** library is automatically linked as needed by the FORTRAN compiler (see *f77*(1)). Neither is automatically loaded by the C compiler (see *cc*(1)); however, the link editor searches this library if the **-lm** (for **libm.a**) or **-lM** (for **libM.a**) option is specified. Declarations for these functions are available in the header file **<math.h>**. Several generally useful mathematical constants are also defined there (see *math*(5)).

(3N) These functions are applicable to the Internet network, and are part of the standard C library, **libc.a**.

(3S) These functions constitute the "standard I/O package" (see *stdio*(3S)). These functions are in the library **libc**, already mentioned. Declarations for these functions can be obtained from the **#include** file **<stdio.h>**.

(3X) Various specialized libraries. The files in which these libraries are found are specified in the appropriate entries.

Definitions

The word **character** is used to refer to a bit representation that fits in a byte and represents a single graphic character or control function. The **null character** is a character with value 0, represented in the C language as **\0**. A **character array** is a sequence of characters. A **null-terminated character array** is a sequence of characters, the last of which is the **null character**. A **string** is a designation for a **null-terminated character array**. The **null string** is a character array containing only the null character. A null pointer is the value that is obtained by casting **0** into a pointer. The C language guarantees that two null pointers always compare equal, and a null pointer always compares unequal to a pointer to any object or function. Consquently, many functions that return pointers return a null pointer to indicate an error. The macro **NULL** expands to a null pointer constant and is defined in **<stddef.h>** and certain other headers.

Many groups of FORTRAN intrinsic functions have generic function names that do not require explicit or implicit type declaration. The type of the function is determined by the type of its argument or arguments. For example, the generic function **max** returns an integer value if given integer arguments (**max0**), a real value if given real arguments (**amax1**), or a double-precision value if given double-precision arguments (**dmax1**).

DIAGNOSTICS

Functions in the C and Math Libraries, (3C) and (3M), may return the conventional values **0** or **±HUGE_VAL** (the largest-magnitude double-precision floating-point numbers; **HUGE_VAL** is defined in the

<math.h> header file) when the function is undefined for the given arguments or when the value is not representable. Functions in the Math Libraries may also return ±INFINITY or NaN. In these cases, the external variable errno (see *errno*(2)) is set to the value EDOM or ERANGE. As many of the FORTRAN intrinsic functions use the routines found in the Math Library, the same conventions apply.

WARNINGS

Library routines in libc.a and libm.a often call other routines in these libraries. Prior to HP-UX release 7.0, a user could define a function having the same name as one of these library routines, and this function would be linked in instead of the library version. In this way, a user could effectively replace a library routine with his own (see *matherr*(3M) for a supported example of this). More often, this type of linkage would occur unintentionally, causing unexpected behavior which was difficult to debug.

Starting at Release 7.0, object names in libraries have been modified such that they are much less likely to collide with user names. Therefore, calls to library routines from within other library routines are much more likely to call the actual library routine. (*matherr*(3M) is the only exception to this.)

In spite of these changes, it is still remotely possible for name conflicts to occur. The *lint*(1) program checker reports name conflicts of this kind as "multiple declarations" of the names in question. Definitions for Sections (2), (3C), and (3S) are checked automatically. Other definitions can be included by using the -l option (for example, -lm includes definitions for the Math Library, libm.a. Use of *lint*(1) is highly recommended.

FILES

/lib/libc.a	Standard I/O, operating system calls, and general purpose routines archive library.
/lib/libc.sl	Standard I/O, operating system calls, and general purpose routines shared library.
/lib/libcurses.sl	CRT screen handling shared library.
/lib/libm.a	SVID2 compliant math archive library.
/lib/libm.sl	SVID2 compliant math shared library.
/lib/libM.a	XPG3, POSIX.1, ANSI-C compliant math archive library.
/lib/libM.sl	XPG3, POSIX.1, ANSI-C compliant math shared library.
/usr/lib/libF77.a	General FORTRAN 77 routines archive library.
/usr/lib/libF77.sl	General FORTRAN 77 routines shared library.

SEE ALSO

intro(2), stdio(3S), math(5), hier(5), ar(1), cc(1), f77(1), ld(1), lint(1), nm(1).

The introduction to this manual.

Device I/O Library, tutorial in *Device I/O Users Guide*.

NAME

a64l(), l64a() - convert between long integer and base-64 ASCII string

SYNOPSIS

```
#include <stdlib.h>

long int a64l(const char *s);

char *l64a(long int l);
```

DESCRIPTION

These functions are used to maintain numbers stored in *base-64* ASCII characters. This is a notation by which long integers can be represented by up to six characters; each character represents a "digit" in a radix-64 notation.

The characters used to represent "digits" are `.` for 0, `/` for 1, `0` through `9` for 2-11, `A` through `Z` for 12-37, and `a` through `z` for 38-63.

The leftmost character is the least significant digit. For example,

$$a0 = (38 \times 64^0) + (2 \times 64^1) = 166$$

`a64l()` takes a pointer to a null-terminated base-64 representation and returns a corresponding `long` value. If the string pointed to by *s* contains more than six characters, `a64l()` uses the first six.

`l64a()` takes a `long` argument and returns a pointer to the corresponding base-64 representation. If the argument is 0, `l64a()` returns a pointer to a null string.

WARNINGS

The value returned by `l64a()` is a pointer into a static buffer, the contents of which are overwritten by each call.

STANDARDS CONFORMANCE

`a64l()`: SVID2

`l64a()`: SVID2

NAME

AAudioString - get name of audio controller (string) passed to AOpenAudio()

SYNOPSIS

```
#include <audio/Alib.h>

char *AAudioString (Audio *audio);
```

DESCRIPTION

AAudioString() returns the *audio_name* (string) that was passed to **AOpenAudio()**. If *audio_name* was NULL, the value of the AUDIO variable was used, and that is the value returned.

audio specifies the **Audio** structure associated with this connection.

RETURN VALUE

Upon successful completion, **AAudioString()** returns the *audio_name* (string) that was passed to **AOpenAudio()**. If *audio_name* is NULL, the value of the AUDIO variable is used, and that is the value returned.

ERRORS

AAudioString() does not return an error status.

EXAMPLES

The following call to **AAudioString** gets the name of the audio controller (string) that was passed to **AOpenAudio()**.

```
char *ac_name; /* name of audio
Audio *audio; /* audio connection */"
      .
      .
      .
/* get audio controller name */
ac_name = AAudioString(audio);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

AAudioString() was developed by HP.

SEE ALSO

Using the Audio Application Program Interface.

NAME

ABestAudioAttributes - get best audio attribute setting for specified controller

SYNOPSIS

```
#include <audio/Alib.h>

AudioAttributes *ABestAudioAttributes (Audio *audio);
```

DESCRIPTION

ABestAudioAttributes() returns a pointer to an **AudioAttributes** structure containing the optimal attributes for the audio controller associated with the *audio* connection. The application can use the returned attributes pointer directly in subsequent audio operation calls.

Changes should not be made to the **AudioAttributes** structure; rather, the application should copy the structure and make changes in the copy, as shown in the example below.

audio specifies the **Audio** structure associated with this connection.

RETURN VALUE

Upon successful completion, **ABestAudioAttributes()** returns a pointer to an **AudioAttributes** structure.

ERRORS

ABestAudioAttributes() does not return an error status.

EXAMPLES

The following example shows a call to **ABestAudioAttributes()** to get the pointer to the best audio attributes.

```
Audio *audio; /* audio connection */"
AudioAttributes *bestAttr; /* best attributes */"
      .
      .
      .
/* get best audio attributes */
bestAttr = ABestAudioAttributes (audio);
```

This example shows how to get a *copy* of the best attributes and make a change to a field in the copy. The program assigns the contents at the returned pointer (the audio attributes) to *myAttr* and then sets the value of the *sampled_attr* field in *myAttr* to **ASAFBitPerSample**.

```
Audio *audio; /* audio connection */
AudioAttributes myAttr;     /* my copy of best attributes */
      .
      .
      .
/* get copy of audio attributes;  change the copy */
myAttr = *ABestAudioAttributes (audio);
myAttr.attr.sampled_attr.data_format = ADFALaw
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

ABestAudioAttributes() was developed by HP.

SEE ALSO

Using the Audio Application Program Interface.

NAME

abort( ) - generate a software abort fault

SYNOPSIS

```
#include <stdlib.h>

void abort(void);
```

DESCRIPTION

`abort( )` first closes all open files, streams, directory streams, and message catalogue descriptors, if possible, then causes the signal `SIGABRT` to be sent to the calling process. This may cause a core dump to be generated (see *signal*(2)).

If the signal `SIGABRT` is caught, the handling function is executed. If the handling function returns, the action for `SIGABRT` is then reset to `IG_DFL`, and the signal `SIGABRT` is sent again to the process to ensure that it terminates.

RETURN VALUE

`abort( )` does not return.

ERRORS

No errors are defined.

APPLICATION USAGE

`SIGABRT` is not intended to be caught.

DIAGNOSTICS

If `SIGABRT` is neither caught nor ignored, and the current directory is writable, a core dump is produced and the message `abort - core dumped` is written by the shell.

SEE ALSO

adb(1), exit(2), kill(2), signal(2). signal(5).

STANDARDS CONFORMANCE

`abort( )`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

abs(), labs() - return integer absolute value

SYNOPSIS

```
#include <stdlib.h>

int abs(int i);

long int labs(long int i);
```

DESCRIPTION

`abs()` returns the absolute value of its integer operand.

`labs()` is similar to `abs()`, except that the argument and the returned value each have type long int.

The largest negative integer returns itself.

WARNINGS

In two's-complement representation, the absolute value of the negative integer with largest magnitude is undefined. Some implementations trap this error, but others simply ignore it.

SEE ALSO

floor(3M).

STANDARDS CONFORMANCE

`abs()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`labs()`: AES, XPG4, ANSI C

NAME

ACalculateLength - return the size in bytes of converted data

SYNOPSIS

```
#include <audio/Alib.h>

long ACalculateLength (
    Audio              *audio,
    long               buffer1_size,
    AudioAttributes    *buffer1_attributes,
    AudioAttributes    *buffer2_attributes,
    long               *status_return ) ;
```

DESCRIPTION

ACalculateLength() returns the size in bytes of the data in buffer 1 after it is converted to the attributes of *buffer2_attributes*.

audio	specifies the **Audio** structure associated with this connection.
buffer1_size	specifies the length in bytes of the data in buffer 1.
buffer1_attributes	specifies the attributes of the data in buffer 1.
buffer2_attributes	specifes the attributes of the data in buffer 2.
status_return	receives the returned status of the operation, unless it is set to NULL.

RETURN VALUE

Upon successful completion, **ACalculateLength()** returns the size in bytes of the data which will be produced by converting a source buffer whose size in bytes is specified in *buffer1_size* and whose attributes are specified in *buffer1_attributes* to the attributes specifed in *buffer2_attributes.*

ERRORS

If *status_return* is not set to NULL, the following is returned in *status_return*:

0 AENoError

EXAMPLE

For an example, see **/usr/audio/examples/splayer.c**

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware installed. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

ACalculate Length() was developed by HP.

SEE ALSO

Using the Audio Application Program Interface.

NAME

ACheckEvent - get first event found in audio event queue

SYNOPSIS

```
#include <audio/Alib.h>

Boolean
ACheckEvent (Audio *audio, AEvent *event_return, long
```

DESCRIPTION

ACheckEvent() dequeues and returns the first event in the queue and returns TRUE. If the queue is empty, the function returns FALSE immediately. This behavior contrasts with **APeekEvent()** which finds but does not dequeue the first event on the queue, and which blocks, if the queue is empty, until an event is received.

audio specifies the **Audio** structure associated with this connection.

event_return is the first event found in the queue.

status_return receives the returned status of the operation, unless it is set to NULL.

RETURN VALUE

Upon successful completion, **ACheckEvent()** returns TRUE or FALSE.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return:*

0 AENoError
2 AEBadAudio

EXAMPLES

The following example sets up *event* to receive event data and *status_return* to receive status data.

```
    Boolean first_event;      /* first event on queue */
Audio *audio;             /* audio connection */
AEvent event_return;      /* event_return */
long status;              /* error status */
          .
          .
          .
    /* check event queue */
    first_event = ACheckEvent(audio, &event_return, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

ACheckEvent() was developed by HP.

SEE ALSO

ACheckMaskEvent(3X), AEventsQueued(3X), AMaskEvent(3X), ANextEvent(3X), APeekEvent(3X), APutBackEvent(3X), AQLength(3X), ASelectInput(3X).

Using the Audio Application Program Interface.

NAME

ACheckMaskEvent - get first event in audio event queue that matches mask

SYNOPSIS

```
#include <audio/Alib.h>

Boolean
ACheckMaskEvent (
    Audio *audio,
    AEventMask event_mask,
    AEvent *event_return,
    long *status_return
);
```

DESCRIPTION

ACheckMaskEvent() dequeues and returns the first event in the queue that matches the mask and returns TRUE. If no match is found, the function returns FALSE immediately. Unlike **AMaskEvent()**, it does not block if no match is found.

audio is the **Audio** structure associated with this connection.

event_mask is the mask specifying what type(s) of event to look for.

event_return is the first event found in the queue.

status_return receives the returned status of the operation, unless it is set to NULL.

RETURN VALUE

Upon successful completion, **ACheckMaskEvent()** returns TRUE or FALSE.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return:*

0 AENoError
2 AEBadAudio
3 AEBadValue

EXAMPLES

The following example sets the event mask to check for errors and transaction started events, and sets up *event_return* to receive event data and *status_return* to receive status data.

```
Boolean first_match;    /* match found*/
Audio *audio;           /* audio connection */
AEventMask emask;       /* event mask */
AEvent event_return;    /* event return*/
long status;            /* error status */
      .
      .
      .
/* check event queue for mask match */
emask = (AErrorMask | ATransStartedMask);
first_match = ACheckEventMask(audio, emask, &event_return, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

ACheckMaskEvent() was developed by HP.

SEE ALSO

ACheckEvent(3X), AEventsQueued(3X), AMaskEvent(3X), ANextEvent(3X), APeekEvent(3X), APutBackEvent(3X), AQlength(3X), ASelectInput(3X).

Using the Audio Application Program Interface.

NAME

AChooseAFileAttributes - select attributes to use when creating a new file

SYNOPSIS

```
#include <audio/Alib.h>

void
AChooseAFileAttributes (
    Audio               *audio,
    AudioAttributes     *src_attributes,
    AFileFormat         file_format,
    AudioAttrMask       user_mask,
    AudioAttributes     *attributes,
    AByteOrder          *byte_order,
    long                *status_return
);
```

DESCRIPTION

AChooseAFileAttributes() selects attributes to use when creating a new file.

audio	specifies the **Audio** structure associated with this connection.
src_attributes	specifies the audio attributes of the source stream.
file_format	specifes the target file format.
user_mask	specifies which of the audio attributes in the *attributes* structure have been supplied by the user (mask bit set to 1). These attributes are checked for validity, but are not changed.
attributes	contains user-supplied attributes (if any) as indicated by *user_mask*. **AChooseAFileAttributes()** writes appropriate values to those attributes not supplied by the user.
byte_order	receives byte ordering for the audio file.
status_return	receives the returned status of the operation, unless it is set to NULL.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return*:

0 AENoError
6 AEBadDataFormat
7 AEBadFileFormat

EXAMPLES

The following example chooses attributes for use when creating a new Sun/NeXT file.

```
Audio           * audio;  /* audio connection */
AudioAttributes
                * src_attribs;  /* source stream attributes */
AFileFormat     dest_file_format;  /* file format */
AudioAttrMask   dest_mask;  /* attributes set by user */
AudioAttributes
                dest_attribs;  /* returned attributes */
AByteOrder      dest_byte_order;  /* returned byte order */
long            status;  /* status */

dest_file_format = AFFSun;
                        .
                        .
                        .
/* Get the attribute structure for the target file. */
/* Specify MuLaw data format and 8k samples/second */
dest_attribs.type = ATSampled;
dest_attribs.attr.sampled_attr.data_format = ADFMuLaw;
dest_attribs.attr.sampled_attr.sampling_rate = 8000;
```

```
dest_mask = ASDataFormatMask | ASSampling Rate Mask;
AChooseAFileAttributes(audio, src_attribs, dest_file_format,
    dest_mask, &dest_attribs, &dest_byte_order, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual shipped with your system.

AUTHOR

AChooseAFileAttributes() was developed by HP.

SEE ALSO

Using the Audio Application Program Interface.

NAME

AChoosePlayAttributes - select hardware-supported attributes to use when playing an existing file or a stream

SYNOPSIS

```
#include <audio/Alib.h>

void
AChoosePlayAttributes (
    Audio               *audio,
    AudioAttributes     *src_attributes,
    AudioAttrMask       user_mask,
    AudioAttributes     * attributes,
    AByteOrder          *byte_order,
    long                *status_return
);
```

DESCRIPTION

AChoosePlayAttributes() selects hardware-supported attributes to use when playing an existing file or a stream.

audio	Specifies the **Audio** structure associated with this connection.
src_attributes	Specifies the audio attributes of the source stream.
user_mask	Specifies which of the audio attributes in the *attributes* structure have been supplied by the user (mask bit set to 1). These attributes are checked for validity, but are not changed.
attributes	Contains user-supplied attributes (if any) as indicated by *user_mask*. **AChoosePlayAttributes** Writes appropriate values to those attributes not supplied by the user.
byte_order	Receives the byte ordering for hardware.
status_return	Receives the returned status of the operation, unless it is set to NULL.

ERRORS

If *status_return* is not set to NULL, the following is returned in *status_return*:

0 AENoError

EXAMPLE

For an example, see **/usr/audio/examples/splayer.c**

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

AChoosePlayAttributes() was developed by HP.

SEE ALSO

Using the Audio Application Program Interface.

NAME

AChooseSourceAttributes - select attributes to associate with an existing file or a stream

SYNOPSIS

```
#include <audio/Alib.h>
        <stdio.h>

AFileFormat
AChooseSourceAttributes (
    Audio              * audio,
    char               * pathname,
    FILE               * audiofile,
    AFileFormat          file_format,
    AudioAttrMask        user_mask,
    AudioAttributes    * attributes,
    long               * offset,
    long               * data_length,
    AByteOrder         * byte_order,
    long               * status_return );
```

DESCRIPTION

AChooseSourceAttributes( ) selects attributes to associate with an existing file or a stream.

audio	specifies the **Audio** structure associated with this connection.
pathname	specifies the pathname of the audio file. Ignored if *audiofile* is NULL.
audiofile	specifies the file pointer if the source is a file. If the source is a stream, such as stdin, set *audiofile* to NULL.
file_format	specifies the format of the file. May be **AFFUnknown.**
user_mask	specifies which of the audio attributes in the *attributes* structure have been supplied by the user (mask bit set to 1). These attributes will be checked for validity, but will not be changed.
attributes	contains user-supplied attributes (if any) as indicated by *user_mask*. **AChooseSourceAttributes** will write appropriate values to those attributes not supplied by the user.
offset	receives the location, in bytes, where the audio data begins.
data_length	receives the length, in bytes, of the audio data.
byte_order	receives the byte ordering of the data in the file.
status_return	receives the returned status of the operation, unless it is set to NULL.

RETURN VALUE

Upon successful completion, **AChooseSourceAttributes( )** returns the format of the file if the source is a file. If the source is a stream or the file format cannot be determined, **AFFUnknown** is returned.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return:*

0	AENoError
2	AEBadAudio
6	AEBadFileFormat
7	AEBadDataFormat
11	AEBadFileHdr
17	AEOutOfMemory

EXAMPLE

For an example, see **/usr/audio/examples/splayer.c**

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has

audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

`AChooseSourceAttributes( )` was developed by HP.

SEE ALSO

Using the Audio Application Program Interface.

NAME

ACloseAudio - close connection to specified audio server

SYNOPSIS

```
#include <audio/Alib.h>

void ACloseAudio (Audio *audio, long *status_return);
```

DESCRIPTION

ACloseAudio() closes the connection to the server specified by *audio* and deallocates the **Audio** structure memory.

ACloseAudio() waits for the audio server to acknowledge that the audio connection is closed before returning. After the connection has been closed, it cannot be resumed or used in any other way.

audio specifies the **Audio** structure associated with this connection.

status_return receives the returned status of the operation unless it is set to NULL.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return.*

0 AENoError
2 AEBadAudio

EXAMPLES

The following example closes the connection to *audio* and sets up *status* to receive status data.

```
Audio *audio;     /* audio connection */
long status;      /* error status */
        .
        .
        .
/* close audio connection */
ACloseAudio(audio, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

ACloseAudio() was developed by HP.

SEE ALSO

AOpenAudio(3X).

Using the Audio Application Program Interface.

NAME

acltostr() - convert access control list (ACL) structure to string form

SYNOPSIS

```
#include <acllib.h>

char *acltostr(int nentries, const struct acl_entry *acl, int form);
```

Remarks:

To ensure continued conformance with emerging industry standards, features described in this manual entry are likely to change in a future release.

DESCRIPTION

`acltostr()` converts an access control list from structure form to string representation. `acltostr()` takes a pointer to the first element of an array of ACL entries (`acl`), containing the indicated number (*nentries*) of valid entries (zero or more), and the output form desired (`FORM_SHORT` or `FORM_LONG`). It returns a pointer to a static string (overwritten by the next call), which is a symbolic representation of the ACL, ending in a null character. The output forms are described in *acl*(5). In long form, the string returned contains newline characters.

A user ID of `ACL_NSUSER` and a group ID of `ACL_NSGROUP` are both represented by `%`. As with the `ls` command (see *ls*(1)), if an entry contains any other user ID or group ID value not listed in `/etc/passwd` or `/etc/group`, `acltostr()` returns a string equivalent of the ID number instead.

Just as in routines that manage the `/etc/passwd` file, `acltostr()` truncates user and group names to eight characters.

Note: `acltostr()` is complementary in function to `strtoacl()`.

RETURN VALUE

If `acltostr()` succeeds, it returns a pointer to a null-terminated string. If *nentries* is zero or less, the string is of zero length. If *nentries* is greater than `NACLENTRIES` (defined in `<sys/acl.h>`), or if *form* is an invalid value, the call returns (char *) NULL.

EXAMPLES

The following code fragment reads the ACL on file `/users/ggd/test` and prints its short-form representation.

```
#include <stdio.h>
#include <acllib.h>

int nentries;
struct acl_entry acl [NACLENTRIES];

if ((nentries = getacl ("/users/ggd/test", NACLENTRIES, acl)) < 0)
     error (...);

fputs (acltostr (nentries, acl, FORM_SHORT), stdout);
```

AUTHOR

`acltostr()` was developed by HP.

FILES

```
/etc/passwd
/etc/group
```

SEE ALSO

getacl(2), setacl(2), cpacl(3C), chownacl(3C), setaclentry(3C), strtoacl(3C), acl(5).

NAME

AConnectionNumber - get connection number for specified audio server connection

SYNOPSIS

```
#include <audio/Alib.h>

long AConnectionNumber (Audio *audio);
```

DESCRIPTION

AConnectionNumber() gets the number for the audio server connection specified by *audio*.

audio specifies the **Audio** structure associated with this connection.

RETURN VALUE

Upon successful completion, **AConnectionNumber()** returns the connection number for the specified audio server connection.

ERRORS

AConnectionNumber does not return an error status.

EXAMPLES

The following example gets the number for the audio connection specified by *audio*.

```
Audio *audio; /* audio connection */"
long conn; /* connection number */"
        .
        .
        .
/* get number of audio connection */
conn = AConnectionNumber(audio);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

AConnectionNumber() was developed by HP.

SEE ALSO

Using the Audio Application Program Interface.

NAME

AConnectRecordSStream - connect socket to TCP socket address; return transaction ID

SYNOPSIS

```
#include <audio/Alib.h>

ATransID AConnectRecordSStream (
    Audio *audio,
    SStream *remote_sstream,
    SSRecordParams *rp,
    long *status_return
);
```

DESCRIPTION

AConnectRecordSStream() is used by an application that is preparing to send a record sound stream to a server on another system. After creating a socket, the application calls **AConnectRecordSStream()** to connect it to the other server at the TCP socket address contained in *remote_stream*, the pointer to which it obtains from the application that controls the other server. The call returns a transaction ID for the operation.

audio specifies the **Audio** structure associated with this application's connection to its own server.

remote_sstream is a structure containing a TCP socket address, audio attributes, and the maximum block size, in bytes, of each transfer of audio data over the connection.

rp is the structure containing the record gain, pause_first toggle, gain matrix, and the mask for event notification.

status_return receives the returned status of the operation unless it is set to NULL.

RETURN VALUE

Upon successful completion, **AConnectRecordSStream()** returns a transaction ID.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return:*

0	AENoError
2	AEBadAudio
10	AEBadGainMatrix
21	AEBadSoundStream

EXAMPLES

The following example connects a socket to another server's TCP socket address and returns a transaction ID for the operation.

```
ATransID xid;              /* transaction ID */
Audio *audio;              /* audio connection */
SStream *r_sstream;        /* remote stream descrip */
SSRecordParams *ss_rp;     /* sstream record params */
long status;               /* error status */
      .
      .
      .
/* connect to TCP socket addr and get transID */
xid = AConnectRecordSStream(audio, r_sstream, ss_rp, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

AConnectRecordSStream() was developed by HP.

SEE ALSO

APlaySStream(3X), ARecordSStream(3X).

Using the Audio Application Program Interface.

NAME

AConvertAFile - convert audio file data format

SYNOPSIS

```
#include <audio/Alib.h>

void AConvertAFile(
    Audio *audio,
    char *src_pathname,
    AFileFormat src_file_format,
    char *dest_pathname,
    AFileFormat dest_file_format,
    AudioAttrMask dest_attr_mask,
    AudioAttributes *dest_attributes,
    long *status_return
);
```

DESCRIPTION

AConvertAFile() converts the data in *src_pathname* according to the format specified in *dest_file_format* and the attributes in *dest_attributes*. The results are written to *dest_pathname*.

Arguments

audio Audio structure associated with this connection.

src_pathname Pathname of the source file.

src_file_format File format of the source file. If this parameter is set to **AFFUnknown**, the conversion utility attempts to determine the file format from the filename extension, if one exists, or from the file contents.

If there is no determinable file format, an error is returned; there is no default.

Valid file type extensions are:

.u	Mulaw
.al	Alaw
.au	Sun (NeXT)
.wav	Riff
.snd	NeXT
.116	Linear16
.18	Linear8
.lo8	Linear8Offset

If you have a "Mac" file, try treating it as a raw data file in Linear8Offset with a sampling rate of 22k or another sampling rate.

dest_pathname Pathname of the destination file.

dest_file_format File format of the destination file.

dest_attr_mask Audio attributes to be used in *dest_attributes*. The mask is a bitwise inclusive OR of the values defined in **AudioAttrMask**. If this mask is set to 0, values are used from the source file wherever they are appropriate for files of type *dest_file_format*.

dest_attributes Attributes that are affected by the mask. If set to NULL, the attribute mask is cleared, and values are used from the source file wherever they are appropriate for files of type *dest_file_format*. For attributes to be valid, type *must* be set, separate from the mask.

status_return Receives the returned status of the operation, unless it is set to NULL.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return*:

0 AENoError
2 AEBadAudio
6 AEBadFileFormat

7 AEBadDataFormat
8 AEFileNotFound
11 AEBadFileHdr
12 AEUnrecognizableFormat
13 AEBadAttribute
16 AECantDetermineFormat

EXAMPLES

The following example converts the data in `/mydir/audfile.wav` to a 30-second Sun (NeXT) format "mono" Mulaw file, sampled at 8000 samples per second, and writes the result in `/mydir/audfile.au`.

```
Audio *audio;                   /* audio connection */
AFileFormat src_fmt;            /* source file format */
AFileFormat dest_fmt;           /* destination file format */
AudioAttrMask a_mask;           /* audio attributes mask */
AudioAttributes dest_attr;      /* destination attributes */
long status;                    /* error status */
        .
        .
        .
/* convert audio file */
static char s_name[] = {"/mydir/aufile.wav"};
src_fmt = AFFUnknown;           /* let AAPI determine file type */
static char d_name[] = {"/mydir/aufile.au"};
dest_fmt = AFFSun;              /* Sun (NeXT) format */
a_mask = 0;
dest_attr.type = ATSampled;     /* must set this */
dest_attr.attr.sampled_attr.data_format = ADFMulaw;
a_mask = a_mask | ASDataFormatMask;
dest_attr.attr.sampled_attr.sampling_rate = 8000;
a_mask = a_mask | ASSamplingRateMask;
dest_attr.attr.sampled_attr.channels = 1;
a_mask = a_mask | ASChannelsMask;
dest_attr.attr.sampled_attr.duration.type = ATTMilliseconds;
dest_attr.attr.sampled_attr.duration.milliseconds = 30000;
a_mask = a_mask | ASDurationMask;
AConvertAFile(audio, s_name, src_fmt, d_name, dest_fmt,
    a_mask, &dest_attr,&status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

`AConvertAFile()` was developed by HP.

SEE ALSO

ALoadAFile(3X), ASaveSBucket(3X).

Using the Audio Application Program Interface.

NAME

AConvertBuffer - convert a buffer of data

SYNOPSIS

```
#include <audio/Alib.h>

void
AConvertBuffer (
    Audio               * audio,
    AConvertParams      * convert_params,
    char                * src_buffer,
    long                src_buffer_size,
    char                * dest_buffer,
    long                dest_buffer_size,
    long                * bytes_read,
    long                * bytes_written,
    long                * status_return );
```

DESCRIPTION

AConvertBuffer( ) converts the data in *src_buffer* according to the attributes specified in *convert_params* and puts the results in *dest_buffer*. Conversion will stop when either all the data in the source buffer has been converted or the destination buffer is full. If the destination buffer fills up before all the source data is converted, *bytes_read* will be less than *src_buffer_size*. If the source buffer empties before the destination buffer is full, *bytes_written* will be less than *dest_buffer_size*

audio	specifies the **Audio** structure associated with this connection.
convert_params	Pointer to a structure describing conversion source and destination; returned by **ASetupConversion .**
src_buffer	data to be converted.
src_buffer_size	size in bytes of the source buffer, *src_buffer*.
dest_buffer	the destination buffer; receives the converted data.
dest_buffer_size	size in bytes of the destination buffer.
bytes_read	receives the number of bytes read.
bytes_written	receives the number of bytes written.
status_return	receives the returned status of the operation, unless it is set to NULL.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return:*

0 AENoError
13 AEBadAttribute
17 AEOutOfMemory

EXAMPLE

For an example, see `/usr/audio/examples/splayer.c`

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

`AConvert Buffer( )` was developed by HP.

SEE ALSO

ASetupConversion(3X), AEndConversion(3X)

Using the Audio Application Program Interface.

NAME

ACreateSBucket - create empty sound bucket and return pointer to it

SYNOPSIS

```
#include <audio/Alib.h>

SBucket *
    ACreateSBucket (
    Audio *audio,
    AudioAttrMask attr_mask,
    AudioAttributes *audio_attributes,
    long *status_return
);
```

DESCRIPTION

ACreateSBucket() creates an empty sound bucket to receive recorded data, associates it with audio attributes, and returns the pointer to it.

audio specifies the **Audio** structure associated with this connection.

attr_mask is the mask used to select attributes

audio_attributes is the structure containing the audio type and attributes. Audio type *must* be set.

status_return receives the returned status of the operation, unless it is set to NULL.

RETURN VALUE

Upon successful completion, **ACreateSBucket()** returns a pointer to a sound bucket.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return:*

0 AENoError
2 AEBadAudio
7 AEBadDataFormat
13 AEBadAttribute
17 AEOutOfMemory
19 AEBadSamplingRate

EXAMPLES

The following example creates sound bucket **sb** and selects Bit Per Sample and Duration attributes:

```
SBucket *sb;            /* sound bucket */
Audio *audio;           /* audio connection */
AudioAttrMask amask;    /* audio attributes mask */
AudioAttributes *attr;  /* audio attributes */
long status;            /* error status */
      .
      .
      .
/* create sound bucket */
amask = (ASAFBitPerSample | ASAFDuration);
sb = ACreateSBucket(audio, amask, attr, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

ACreateSBucket() was developed by HP.

SEE ALSO

ADestroySBucket(3X), AGetSBucketData(3X), ALoadAFile(3X), APlaySBucket(3X), APutSBucketData(3X), ARecordAData(3X), ASaveSBucket(3X).

Using the Audio Application Program Interface.

NAME

ADataFormats - get list of data formats supported by audio controller

SYNOPSIS

```
#include <audio/Alib.h>

ADataFormat *ADataFormats (Audio *audio);
```

DESCRIPTION

ADataFormats() returns a pointer to a list of the data formats that are supported by the audio controller associated with the *audio* connection. The length of the list is returned by **ANumDataFormats()**.

audio specifies the **Audio** structure associated with this connection.

RETURN VALUE

On successful completion, **ADataFormats()** returns a pointer to a list of data formats that are supported by the audio controller.

EXAMPLES

The following example gets a list of data formats that are supported by the audio controller associated with *audio*.

```
ADataFormat *list_fmts;    /* list of data formats */
Audio *audio;              /* audio connection */
      .
      .
      .
/* get list of data formats */
list_fmts = ADataFormats(audio);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

ADataFormats() was developed by HP.

SEE ALSO

ANumDataFormats(3X).

Using the Audio Application Program Interface.

NAME

addopt() - add argument and data to NetIPC option buffer

SYNOPSIS

```
#include <sys/ns_ipc.h>

void addopt(
    short opt[],
    short argnum,
    short optioncode,
    short datalength,
    short data[],
    short *result);
```

DESCRIPTION

`addopt()` adds an argument and its associated data to a NetIPC *opt* buffer. A NetIPC option buffer is a data array structured and used by NetIPC . The size of the data array can be determined by calling `optoverhead()` (see *optoverhead*(3N)). The buffer must be initialized by calling `initopt()` (see *initopt*(3N)).

Parameters

opt (input parameter) The *opt* buffer to which you want to add an argument.

argnum (input parameter) The number of the argument to be added. The first argument is number zero.

optioncode (input parameter) The option code of the argument to be added. These codes are described in each NetIPC system call *opt* parameter description.

datalength (input parameter) The length in bytes of the data to be included. This information is provided in each NetIPC system call *opt* parameter description.

data (input parameter) An array containing the data associated with the argument.

result (output parameter) The result code returned. Refer to "Diagnostics" below for more information.

RETURN VALUE

None. Errors are returned in the *result* parameter.

ERRORS

[NSR_ADDR_OPT] An unknown or illegal option number was specified.

[NSR_NO_ERROR] The call was successful.

[NSR_OPT_DATA_LEN] The length of the *opt* parameter is less than 0.

[NSR_OPT_ENTRY_NUM] Option index is less than 0 or greater than the option buffer for which it was initialized. If an option buffer is initialized for 3 options, number the options as 0, 1, and 2. In this example, the number 3 is illegal.

AUTHOR

`addopt()` was developed by HP.

SEE ALSO

ipcconnect(2), ipccontrol(2), ipccreate(2), ipcdest(2), ipcgetnodename(2), ipclookup(2), ipcname(2), ipcnamerase(2), ipcrecv(2), ipcrecvcn(2), ipcselect(2), ipcsend(2), ipcsetnodename(2), ipcshutdown(2), initopt(3N), ipcerrmsg(3N), optoverhead(3N), readopt(3N).

NAME

ADestroySBucket - destroy specified sound bucket

SYNOPSIS

```
#include <audio/Alib.h>

void ADestroySBucket (
    Audio * audio,
    SBucket *sb,
    long *status_return
);
```

DESCRIPTION

ADestroySBucket() destroys the specified sound bucket and frees the space allocated for the sound bucket and its audio data.

audio specifies the **Audio** structure associated with this connection.

sb specifies the sound bucket to be destroyed.

status_return receives the returned status of the operation, unless it is set to NULL.

Once it has been destroyed, a sound bucket cannot be used to play or record.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return:*

0	AENoError
2	AEBadAudio
20	AEBadSoundBucket

EXAMPLES

The following example destroys the sound bucket *sb* and frees its allocated space.

```
Audio *audio;              /* audio connection */
SBucket *sb;               /* sound bucket */
long status;               /* error status */
      .
      .
      .
/* destroy sound bucket */
ADestroySBucket(audio, sb, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

ADestroySBucket() was developed by HP.

SEE ALSO

ACreateSBucket(3X), AGetSBucketData(3X), ALoadAFile(3X), APlaySBucket(3X), APutSBucketData(3X), ARecordAData(3X), ASaveSBucket(3X).

Using the Audio Application Program Interface.

NAME

AEndConversion - finish stream data conversion

SYNOPSIS

```
#include <audio/Alib.h>

void
AEndConversion (
    Audio              * audio,
    AConvertParams     * convert_params,
    char               * dest_buffer,
    long               dest_buffer_size,
    long               * bytes_written,
    long               * status_return );
```

DESCRIPTION

AEndConversion( ) converts any data remaining in the conversion pipeline to the attributes specified in *convert_params* and puts the results in *dest_buffer*. The amount of data written is returned in *bytes_written*. **AEndConversion** also frees the structure, *convert_params*.

audio	specifies the **Audio** structure associated with this connection.
convert_params	pointer to a structure describing conversion source and destination; returned by **ASetupConversion .**
dest_buffer	the destination buffer; receives the converted data.
dest_buffer_size	size in bytes of the destination buffer.
bytes_written	receives the number of bytes written.
status_return	receives the returned status of the operation, unless it is set to NULL.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return:*

0 AENoError

EXAMPLE

For an example, see `/usr/audio/examples/splayer.c`

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

`AEndConversion( )` was developed by HP.

SEE ALSO

ASetupConversion(3X), AConvertBuffer(3X),

Using the Audio Application Program Interface.

NAME

AEventsQueued - get number of events in queue for specified server connection

SYNOPSIS

```
#include <audio/Alib.h>

long AEventsQueued (
    Audio *audio, AQueueCheckMode mode,
    long *status_return
);
```

DESCRIPTION

AEventsQueued() returns the number of events in the queue for the specified audio server, depending on *mode*.

audio specifies the **Audio** structure associated with this connection.

mode is **AQueuedAlready** or **AQueuedAfterReading**.

If the mode is **AQueuedAlready**, the call returns the number of events in the queue including zero. If the mode is **AQueuedAfterReading** and there are no events on the queue, the function tries to determine whether the server has more events for this connection, and returns the number it finds. If there are none, it returns zero.

status_return receives the returned status of the operation unless it is set to NULL.

RETURN VALUE

Upon successful completion, **AEventsQueued()** returns the number of events in the queue for the specified server connection, depending upon the *mode*.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return:*

0	AENoError
2	AEBadAudio
3	AEBadValue

EXAMPLES

The following example gets the number of events on the queue for the audio server connection specified by *audio* and sets up *status* to receive an error status. The *mode* is set to **AQueuedAlready** so that the call will return zero if there are no events in the queue.

```
long e_num;              /* number of events in q */
Audio *audio;            /* audio connection */
AQueueCheckMode mode;    /* check mode */
long status; /* error status */"
      .
      .
      .
/* check event queue */
mode = AQueuedAlready;
e_num = AEventsQueued(audio, mode, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

AEventsQueued() was developed by HP.

SEE ALSO

ACheckEvent(3X) ACheckMaskEvent(3X), AMaskEvent(3X), ANextEvent(3X), APeekEvent(3X), APutBackEvent(3X), AQLength(3X), ASelectInput(3X).

Using the Audio Application Program Interface.

NAME

AGetAFileAttributes() - get file attributes of specified file

SYNOPSIS

```
#include <audio/Alib.h>

AFileFormat AGetAFileAttributes (
    Audio *audio,
    char *name,
    long *offset,
    long *data_length,
    AByteOrder *file_byte_order,
    AudioAttrMask *mask,
    AudioAttributes *file_attr,
    long *status_return
    );
```

DESCRIPTION

AGetAFileAttributes() returns the file format of the file specified in *name*.

audio	specifies the **Audio** structure associated with this connection.
name	the pathname of the audio data file to be queried.
offset	receives the number of bytes into the file where the audio samples begin.
data_length	receives the length (in bytes) of the audio data.
file_byte_order	receives the byte order (relevant only for 116 data).
mask	receives the information indicating which attributes were determined from file header or file extension (mask bits set to 1). A mask bit set to 0 indicates that the attribute was determined by inference.
file_attr	attribute structure that receives requested attribute information.
status_return	receives the returned status of the operation, unless it is set to NULL.

RETURN VALUE

Upon successful completion, **AGetAFileAttributes()** returns the file type of the file specified in *name*. It also returns the length, the byte order, the attributes, and a mask that indicates how the attribute values were derived. **AFFUnknown** is returned if the format type cannot be determined.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return:*

0	AENoError
2	AEBadAudio
8	AEFileNotFound
11	AEBadFileHdr
13	AEBadAttribute
16	AECantDetermineFormat

EXAMPLES

The following example queries the file attributes of the file **/myhome/a_dir/a_file**.

```
AFileFormat file_fmt;          /* file format */
Audio *audio;                  /* audio connection */
long offset;                   /* offset where data begins */
long data_length;              /* returned data length */
AByteOrder byte_order;         /* returned byte order */
AudioAttrMask attr_mask;       /* attr found in hdr or .ext */
AudioAttributes attribs;       /* returned attributes */
long status;                   /* status */
      .
      .
      .
/* get attributes of /myhome/a_dir/a_file */
charfname[] = /myhome/a_dir/a_file ;
file_fmt = AGetAFileAttributes(audio, fname, &offset, &data_len,
&byte_order, &attr_mask, &attribs, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The

audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

`AGetAFileAttributes()` was developed by HP.

SEE ALSO

Using the Audio Application Program Interface.

NAME

AGetChannelGain - get transaction channel gain

SYNOPSIS

```
#include <audio/Alib.h>

void AGetChannelGain (
    Audio *audio,
    ATransID xid,
    AChType channel,
    AGainDB *gain_return,
    long *status_return
);
```

DESCRIPTION

`AGetChannelGain()` Returns the transaction gain value.

audio the `Audio` structure associated with this connection.

xid the transaction ID.

channel the type of channel: `ACTMono`, `ACTLeft`, or `ACTRight`.

gain_return receives the returned gain value.

status_return receives the returned status of the operation, unless it is set to NULL.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return*.

0 AENoError
2 AEBadAudio
3 AEBadValue

EXAMPLES

The following example returns the transaction left channel gain value.

```
Audio *audio;                   /* audio connection */
AChType *chtype;                /* type of channel */
AGainDB chgain_ret;             /* transaction gain value*/
long status;                    /* error status */
      .
      .
      .
/* get xid left channel gain */
chtype = ACTLeft
AGetChannelGain(audio, xid, chtype, &chgain_ret, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

`AGetChannelGain()` was developed by HP.

SEE ALSO

AGetGain(3X), AGetSystemChannelGain(3X), AGMGainRestricted(3X), AInputChannels(3X), AInputSources(3X), AMaxInputGain(3X), AMaxOutputGain(3X), AMinInputGain(3X), AMinOutputGain(3X), AOutputChannels(3X), AOutputDestinations(3X), ASetChannelGain(3X), ASetGain(3X), ASetSystemChannelGain(3X), ASimplePlayer(3X), ASimpleRecorder(3X).

Using the Audio Application Program Interface.

NAME

AGetDataFormats - get data formats for a specified file format

SYNOPSIS

```
#include <audio/Alib.h>

long
AGetDataFormats (
    AFileFormat         file_format);
```

DESCRIPTION

AGetDataFormats( ) returns a mask of supported data formats for the file format specified in *file_format*. The returned mask is a long, whose bits correspond to the enum "ADataFormat' defined in Alib.h.

The encoding is:

```
/*  *dataFormatNames[ ] =

     Unknown,   0
     MuLaw,     1
     ALaw,      2
     Lin16,     3
     Lin8,      4
     Lin80ffset,
                5
*/
```

file_format specifies the file format of interest.

RETURN VALUE

Upon successful completion, **AGetDataFormats( )** returns a long integer mask of the valid data formats for the given file format. If the file format itself is invalid, **AGetDataFormats** returns zero.

EXAMPLE

The following example gets the data formats for a Sun/NeXT file .

```
long            data_formats_msk;  /* supported data formats */
                .
                .
                .
/* determine valid data formats for a Sun/NeXT file */
data_formats_msk = AGetDataFormats (AFFSun);

/* determine if MuLaw is supported */
if (data_formats_msk & ( 1 << ADFMuLaw ))

    /* Mulaw is a supported data type */
                .
                .
                .
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

AGetDataFormats( ) was developed by HP.

SEE ALSO

Using the Audio Application Program Interface.

NAME

AGetErrorText - copy error description into specified buffer

SYNOPSIS

```
#include <audio/Alib.h>

void AGetErrorText (
    Audio *audio,
    AError error,
    char *buffer_return,
    int buffer_length
);
```

DESCRIPTION

`AGetErrorText()` copies the description for the error specified in *error* to the buffer specified in *buffer_return*. The error description is a null-terminated string.

audio specifies the `Audio` structure associated with this connection.

error specifies the type of error.

buffer_return receives the error description.

buffer_length specifies the size of *buffer_return*.

ERRORS

`AGetErrorText()` does not return an error status.

EXAMPLES

The following example gets the error description for `AEBadOffset` and returns it in *buffer_return*:

```
#define TEXT_LENGTH 256
Audio *audio;                       /* audio connection */
AError err;                         /* type of error */
char buffer_return[TEXT_LENGTH]; /* buffer for description */
      .
      .
      .
/* get error description */
err = AEBadOffset
AGetErrorText(audio, err, &buffer_return, TEXT_LENGTH);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

`AGetErrorText()` was developed by HP.

SEE ALSO

Using the Audio Application Program Interface.

NAME

AGetGain - get play volume or record gain of specified transaction

SYNOPSIS

```
#include <audio/Alib.h>

void AGetGain (
    Audio *audio,
    ATransID xid,
    AGainDB *gain_return
    long *status_return );
```

DESCRIPTION

AGetGain() returns the play volume or record gain of the transaction specified in *xid*.

audio specifies the **Audio** structure associated with this connection.

xid specifies the transaction ID.

gain_return receives the returned gain value.

status_return receives the returned status of the operation unless it is set to NULL.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return*.

0 AENoError
2 AEBadAudio
15 AEBadTransactionID

EXAMPLES

The following example gets the gain for the *xid* transaction and sets up *status* to receive an error status.

```
Audio *audio;           /* audio connection */
TransID xid;            /* transaction ID */
AGainDB gain_return;    /* gain return */
long status;            /* error status */
      .
      .
      .
/* get gain for xid returned by prior call */
AGetGain(audio, xid, &gain_return, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

AGetGain() was developed by HP.

SEE ALSO

AGMGainRestricted(3X), AGetSystemMonitorGain(3X), AGetSystemPlayGain(3X), AGetSystemRecordGain(3X), AInputChannels(3X), AInputSources(3X), AMaxInputGain(3X), AMaxOutputGain(3X), AMinInputGain(3X), AMinOutputGain(3X), AOutputChannels(3X), AOutputDestinations(3X), ASetGain(3X), ASetSystemMonitorGain(3X) ASetSystemPlayGain(3X), ASetSystemRecordGain(3X), ASimplePlayer(3X), ASimpleRecorder(3X).

Using the Audio Application Program Interface.

NAME

AGetSBucketData - copy audio data in sound bucket to buffer; return number of bytes

SYNOPSIS

```
#include <audio/Alib.h>

unsigned long AGetSBucketData (
    Audio *audio,
    SBucket *sb,
    unsigned long start_offset,
    char *buffer,
    unsigned long buf_len,
long *status_return
);
```

DESCRIPTION

AGetSBucketData() copies the audio data in the specified sound bucket to the specified buffer and returns the number of bytes copied.

audio specifies the **Audio** structure associated with this connection.

sb specifies the sound bucket containing the data to be copied.

start_offset specifies the starting point of the copy, given as the byte offset from the beginning of the data.

buffer specifies the buffer to receive the copied data.

buf_len specifies the maximum length of the buffer, in bytes.

status_return receives the returned status of the operation, unless it is set to NULL.

This call is used only when the application needs to manipulate the sound bucket data directly.

RETURN VALUE

Upon successful completion, **AGetSBucket()** returns the byte count of the copied data.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return:*

0 AENoError
2 AEBadAudio
20 AEBadSoundBucket

EXAMPLES

The following example copies the audio data contained in *sb* to the buffer at *bufp* and returns the number of bytes that were copied. In this example, we allocate 80 000 bytes for the buffer, and pass this size value in *buflen*.

```
unsigned long datalen_g; /* copied get_data
Audio *audio; /* audio connection */
SBucket *sb; /* sound bucket*/
unsigned long startoff; /* start offset
char *bufp; /* ptr to buffer
unsigned long buflen; /* length of
long status; /* error status */
      .
      .
      .
/* copy sound bucket data to buffer */
startoff = 0;
bufp = malloc(80000);
buflen = 80000;
datalen_g = AGetSBucketData(audio, sb, startoff, bufp, buflen, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The

audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

`AGetSBucketData()` was developed by HP.

SEE ALSO

ACreateSBucket(3X), ADestroySBucket(3X), ALoadAFile(3X), APlaySBucket(3X), APutSBucketData(3X), ARecordAData(3X), ASaveSBucket(3X).

Using the Audio Application Program Interface.

NAME

AGetSilenceValue - get a silence value

SYNOPSIS

```
#include <audio/Alib.h>

long
AGetSilenceValue (
    Audio               * audio,
    ADataFormat         data_format,
    long                * significant_bytes_return,
    long                * status_return );
```

DESCRIPTION

AGetSilenceValue( ) returns the appropriate "silence" value for the given data format. (Some data formats do not use zero to correspond to silence.) The silence value can be used for clearing or padding an audio file or buffer.

audio specifies the **Audio** structure associated with this connection.

data_format the data format for which a silence value will be returned.

significant_bytes_return
indicates the number of bytes of the returned long that constitute the actual silence value. Currently, all silence values are one byte in length. The application will thus need to cast the silence value to an unsigned char before using it.

status_return receives the returned status of the operation, unless it is set to NULL.

RETURN VALUE

Upon successful completion, **AGetSilenceValue( )** returns a long integer containing the silence value in the least significant bytes.

ERRORS

If *status_return* is not set to NULL, the following is returned in *status_return:*

0 AENoError

EXAMPLE

The following example gets the silence value for MuLaw data.

```
Audio          * audio;  /* audio connection */
ADataFormat    data_format;  /* data format of interest */
long           significant_bytes;  /* number valid bytes in returned
               long */
unsigned char  silence_value;  /* pads audio file or buffer with
               silence */
long           status;  /* status */
               .
               .
               .
/* get silence value for MuLaw data */
data_format = ADFMulaw;
silence_value = (unsigned char)AGetSilenceValue(audio, data_format,
    &significant_bytes, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

`AGetSilenceValue( )` was developed by HP.

SEE ALSO

Using the Audio Application Program Interface.

NAME

AGetSystemChannelGain - get system or monitor audio channel gain

SYNOPSIS

```
#include <audio/Alib.h>

void AGetSystemChannelGain(
    Audio *audio,
    ASystemGainType gain_type,
    AChType channel,
    AGainDB *gain_return,
    long *status_return
);
```

DESCRIPTION

AGetSystemChannelGain() returns the current monitor or system gain.

audio **Audio** structure associated with this connection.

gain_type Type of operation: **ASGTPlay**, **ASGTRecord**, or **ASGTMonitor**. If this field is set to **ASGTMonitor**, the channel specification must be **ACTMono**.

channel Type of channel: **ACTMono**, **ACTLeft**, or BR ACTRight . If *gain_type* is **ASGTMonitor**, this field must be **ACTMono**.

gain_return Receives the returned gain value.

status_return Receives the returned status of the operation unless it is set to NULL.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return*.

0 AENoError
2 AEBadAudio
3 AEBadValue

EXAMPLES

The following example gets the system monitor gain:

```
Audio *audio; /* audio connection */"
ASystemGainType *sgtype; /* type of operation
AChType * chtype; /* type of channel
AGainDB chgain_ret; /* gain value*/"
long status; /* error status */"
      .
      .
      .
/* get monitor gain */
sgtype = ASGTMonitor
chtype = ACTMono
ASetSystemChannelGain(audio, sgtype, chtype, &chgain_ret, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

AGetSystemChannelGain() was developed by HP.

SEE ALSO

AGetChannelGain(3X) AGetGain(3X), AGMGainRestricted(3X), AInputChannels(3X), AInputSources(3X), AMaxInputGain(3X), AMaxOutputGain(3X), AMinInputGain(3X), AMinOutputGain(3X), AOutputChannels(3X), AOutputDestinations(3X), ASetChannelGain(3X), ASetGain(3X), ASetSystemChannelGain(3X), ASimplePlayer(3X), ASimpleRecorder(3X).

Using the Audio Application Program Interface.

NAME

AGetTransStatus - get status of specified transaction

SYNOPSIS

```
#include <audio/Alib.h>

void AGetTransStatus (
    Audio *audio,
    ATransID xid,
    ATransStatus *trans_status_return,
    long *status_return );
```

DESCRIPTION

`AGetTransStatus()` gets the status of the transaction specified in *xid*.

audio specifies the `Audio` structure associated with this connection.

xid specifies the ID of the transaction.

trans_status_return receives the returned status value.

status_return receives the returned status of the operation unless it is set to NULL. If set to NULL, the transaction status is returned as an `AETTransStatus` event which can be read by `ANextEvent()`, `ACheckEvent()`, or `ACheckMaskEvent()`

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return*.

0 AENoError
2 AEBadAudio
15 AEBadTransactionID

EXAMPLES

The following example gets the status for the *xid* transaction and sets up *trans_stat* to receive the transaction status and *status* to receive an error status.

```
Audio *audio; /* audio connection */
ATransStatus trans_stat; /* transaction status return */
long status; /* error status */
      .
      .
      .
/* get status for xid returned from prior call */
AGetTransStatus(audio, xid, &trans_stat, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

`AGetTransStatus()` was developed by HP.

SEE ALSO

Using the Audio Application Program Interface.

NAME

AGMGainRestricted - find out if audio controller restricts gain entries

SYNOPSIS

```
#include <audio/Alib.h>

Boolean AGBGainRestricted (Audio *audio);
```

DESCRIPTION

AGBGainRestricted() returns TRUE if gain is restricted to **AUnityGain** or **AZeroGain**. It returns FALSE if other values can be used for gain entries.

audio specifies the **Audio** structure associated with this connection.

RETURN VALUE

Upon succesful completion, **AGBGainRestricted()** returns TRUE if the audio controller restricts gain entries to **AUnityGain** or **AZeroGain**. It returns FALSE if other values can be used for gain entries.

ERRORS

AGBGainRestricted() does not return an error status.

EXAMPLES

The following example queries the audio controller to see if gain entries are restricted:

```
Boolean restricted; /* gain restricted */
Audio *audio; /* audio connection */
      .
      .
      .
/* find out if gain values are restricted */
restricted=AGMGainRestricted(audio);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

AGMGainRestricted() was developed by HP.

SEE ALSO

AGetChannelGain(3X) AGetGain(3X), AGetSystemChannelGain(3X), AInputChannels(3X), AInputSources(3X), AMaxInputGain(3X), AMaxOutputGain(3X), AMinInputGain(3X), AMinOutputGain(3X), AOutputChannels(3X), AOutputDestinations(3X), ASetChannelGain(3X), ASetGain(3X), ASetSystemChannelGain(3X), ASimplePlayer(3X), ASimpleRecorder(3X).

Using the Audio Application Program Interface.

NAME

AGrabServer - acquire exclusive use of audio server

SYNOPSIS

```
#include <audio/Alib.h>

Boolean AGrabServer (Audio *audio, long *status_return);
```

DESCRIPTION

AGrabServer() acquires exclusive use of the audio server for this connection and returns TRUE unless the server has already been grabbed, in which case it returns FALSE. When the server is grabbed, all requests from other connections are interrupted; they are resumed when the server is released. To release (ungrab) the server, call **AUngrabServer()**.

audio specifies the **Audio** structure associated with this connection.

status_return receives the returned status of the operation, unless it is set to NULL.

RETURN VALUE

Upon successful completion, **AGrabServer()** returns TRUE; if the server is already grabbed, the return is FALSE.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return.*

0 AENoError
2 AEBadAudio

EXAMPLES

The following example grabs the server for the connection associated with *audio* and sets up *status* to receive an error status.

```
Boolean grab;          /* server acquired */
Audio *audio;          /* audio connection */
long status;           /* error status */
      .
      .
      .
/* grab server for audio connection */
grab = AGrabServer(audio, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

AGrabServer() was developed by HP.

SEE ALSO

AUngrabServer(3X).

Using the Audio Application Program Interface.

NAME

AInputChannels - get list of A/D input channels on current hardware

SYNOPSIS

```
#include <audio/Alib.h>

AInputChMask AInputChannels (Audio *audio);
```

DESCRIPTION

AInputChannels() returns a mask showing the Analog and/or Digital input channels that exist on the current hardware. Each bit in the returned **AInputChMask** correponds to one input channel.

audio specifies the **Audio** structure associated with this connection.

RETURN VALUE

Upon successful completion, **AInputChannels()** returns a mask showing the input channels that exist on the current hardware: mono, left, or right input. Each bit in the returned **AInputChMask** correponds to one type of input channel.

ERRORS

AInputChannels() does not return an error status.

EXAMPLES

The following example gets the types of input channels that exist on the current hardware.

```
AInputChMask in_channels; /* mask showing existing input channels */
Audio *audio;             /* audio connection */
      .
      .
      .
/* get input channels */
in_channels = AInputChannels(audio);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

AInputChannels() was developed by HP.

SEE ALSO

AGetChannelGain(3X) AGetGain(3X), AGetSystemChannelGain(3X), AGMGainRestricted(3X), AInputSources(3X), AMaxInputGain(3X), AMaxOutputGain(3X), AMinInputGain(3X), AMinOutputGain(3X), AOutputChannels(3X), AOutputDestinations(3X), ASetChannelGain(3X), ASetGain(3X), ASetSystemChannelGain(3X), ASimplePlayer(3X), ASimpleRecorder(3X).

Using the Audio Application Program Interface.

NAME

AInputSources - get types of input sources existing on current hardware

SYNOPSIS

```
#include <audio/Alib.h>

AInputSrcMask AInputSources (Audio *audio);
```

DESCRIPTION

AInputSources() returns a mask showing the types of input sources that exist on the current hardware. Each bit in the returned **AInputSrcMask** correponds to one type of input source.

audio specifies the **Audio** structure associated with this connection.

RETURN VALUE

Upon successful completion, **AInputSources()** returns a mask showing the types of input sources that exist on the current hardware: mono, left, or right microphone input jacks, and mono, left, or right auxiliary input jacks. Each bit in the returned **AInputSrcMask** correponds to one type of input source.

ERRORS

AInputSources() does not return an error status.

EXAMPLES

The following example gets the types of input source that exist on the current hardware.

```
AInputSrcMask sources;  /* input source mask */
Audio *audio;           /* audio connection */
      .
      .
      .
/* get input sources */
sources = AInputSources(audio);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

AInputSources() was developed by HP.

SEE ALSO

AGetChannelGain(3X) AGetGain(3X), AGetSystemChannelGain(3X), AGMGainRestricted(3X), AInputChannels(3X), AMaxInputGain(3X), AMaxOutputGain(3X), AMinInputGain(3X), AMinOutputGain(3X), AOutputChannels(3X), AOutputDestinations(3X), ASetChannelGain(3X), ASetGain(3X), ASetSystemChannelGain(3X), ASimplePlayer(3X), ASimpleRecorder(3X).

Using the Audio Application Program Interface.

NAME

almanac() - return numeric date information in MPE format

SYNOPSIS

```
void almanac(
    unsigned short int date,
    unsigned short int err[2],
    short int *pyear,
    short int *pmonth,
    short int *pday,
    short int *pweekday
);
```

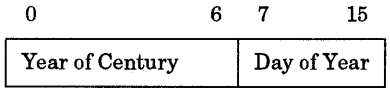
DESCRIPTION

`almanac()` returns numeric date information for a date in the packed date format returned by the `calendar()` routine (see *calendar*(3X)). The returned information is:

year of the century
month of the year
day of the month
day of the week

The arguments to `almanac()` are used as follows:

date An unsigned short containing the date about which information is to be returned. The year of the century is packed into bits 0 through 6, and the day of the year is packed into bits 7 through 15. The packed date format is:



err The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
1	No parameters are present in which to return values: pday, pmonth, pyear, and pweek all point to zero.
2	Day of the year is out of range.
3	Year of the century is out of range.

pyear A pointer to a short in which the year of the century is returned.

pmonth A pointer to a short in which the month of the year is returned (for example, January is represented by `1` and December is represented by `12`).

pday A pointer to a short in which the day of the month is returned.

pweekday A pointer to a short in which the weekday is returned. Note that `1` is returned for Sunday and `7` for Saturday.

WARNINGS

This routine is provided for compatibility with MPE, a proprietary HP operating system. See *portnls*(5) for more information on the use of this routine. Refer to *hpnls*(5) for information about Native Language Support routines used in C programs in the HP-UX NLS environment.

AUTHOR

`almanac()` was developed by HP.

SEE ALSO

calendar(3X), nlfmtdate(3X), ctime(3C), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

ALoadAFile - copy audio file into new sound bucket with data conversion

SYNOPSIS

```
#include <audio/Alib.h>

SBucket * ALoadAFile(
    Audio *audio,
    char *pathname,
    AFileFormat file_format,
    AudioAttrMask attr_mask,
    AudioAttributes *sb_attributes,
    long *status_return
);
```

DESCRIPTION

ALoadAFile() copies the audio data in *pathname* into a new sound bucket and returns the pointer to the sound bucket. The data is converted according to the specified attributes. The HP-UX kernel configuration sets a data size restriction. If the audio data file exceeds this size, the function returns a **OutOfMemory** error.

When the sound bucket is no longer needed, call **ADestroySBucket()** to deallocate the space.

audio is the audio structure associated with this connection.

pathname specifies the file containing the audio data.

file_format must be set to a valid enumerated value, or else an error is returned.

If this parameter is set to **AFFUnknown**, the conversion utility checks for an extension on *pathname*. Extensions can be appended to the filename as follows:

name.sampling_rate.file_type.

Valid sampling rate extensions are **.***n* and **.***n***k** where **.***n***k** is typically **8k** to **22k**.

Valid file type extensions are:

.u	Mulaw
.al	Alaw
.au	Sun (NeXT)
.wav	Riff
.snd	NeXT
.116	Linear16
.18	Linear8
.lo8	Linear8Offset

If no recognizable extension exists, the utility checks the header on the *pathname* file. If file format is not valid or is not determinable, an error is returned.

If you have a "Mac" file, try treating it as a raw data file in Linear8Offset with a sampling rate of 22k or another sampling rate.

attr_mask specifies the audio attributes to associate with the new sound bucket. The mask is a bitwise inclusive OR of values defined in **AudioAttrMask**.

If this value is set to 0 or if *sb_attributes* is set to NULL, the *pathname* attributes are used if the controller supports them. If there is an unsupported attribute, the attribute returned by **ABestAudioAttributes()** is used.

If the mask is set, the new attributes are used *without checking for controller support*. This allows **ALoadAFile()** to be used purely for conversion purposes.

NOTE: If **ASDurationMask** is set, the *pathname* audio data is truncated or padded with zeros to match the length specified in **audio_attributes.sampled_attr.duration**.

sb_attributes specifies the attributes that are affected by the mask. Audio type *must* be set, separate from the mask. If the attribute is different from the one used by *pathname*, the data is

converted.

status_return receives the returned status of the operation unless it is set to NULL.

RETURN VALUE

Upon successful completion, **ALoadAFile()** returns a pointer to the new sound bucket.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return:*

0 AENoError
2 AEBadAudio
6 AEBadFileFormat
7 AEBadDataFormat
8 AEFileNotFound
11 AEBadFileHdr
16 AECantDetermineFormat
17 AEOutOfMemory

EXAMPLES

The following example copies the file **/myhome/a_dir/a_file** into the new sound bucket and specifies **AFFRawALaw** for the file format. Specifying zero for *a_mask* and NULL for *myAttr* means that the *pathname* attributes will be used if the controller supports them; if there is an unsupported attribute, the attribute returned by **ABestAudioAttributes()** will be used:

```
SBucket *sb; /* sound bucket*/ Audio *audio; /* audio connection */
char a_name[30]; /* file name */ AFileFormat file_fmt; /* file format
*/ AudioAttrMask a_mask; /* audio attributes mask*/ AudioAttributes
*myAttr; /* audio attributes */ long status; /* error status */
     .        .        . /* load file into new sound bucket */ a_name
= "/myhome/a_dir/a_file"; file_fmt = AFFRawALaw; a_mask = 0; myAttr =
""; sb = ALoadAFile(audio, a_name, file_fmt, a_mask, myAttr, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

ALoadAFile() was developed by HP.

SEE ALSO

ACreateSBucket(3X), ADestroySBucket(3X), AGetSBucketData(3X), APlaySBucket(3X), APutSBucketData(3X), ARecordAData(3X), ASaveSBucket(3X).

Using the Audio Application Program Interface.

NAME

AMaskEvent - get first matching event in audio event queue

SYNOPSIS

```
#include <audio/Alib.h>

void AMaskEvent(
    Audio *audio,
    AEventMask event_mask,
    AEvent *event_return,
    long *status_return );
```

DESCRIPTION

AMaskEvent() dequeues and returns the first event in the queue that matches the mask. If no match is found, **AMaskEvent()** blocks until a matching event is received. This behavior is unlike **ACheckMaskEvent()** which does not block and returns FALSE immediately if no match is found.

audio	is the **Audio** structure associated with this connection.
event_mask	is the mask specifying what type(s) of event to look for.
event_return	is the first event found in the queue.
status_return	receives the returned status of the operation, unless it is set to NULL.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return:*

0	AENoError
2	AEBadAudio
3	AEBadValue

EXAMPLES

The following example sets the event mask to select errors and transaction started events, and sets up *event_return* to receive event data and *status_return* to receive status data.

```
Audio *audio; /* audio connection */
AEventMask emask; /* event mask */
AEvent event_return; /* event return */
long status; /* error status */
      .
      .
      .
/* check event queue for mask match */
emask = (AErrorMask|ATransStartedMask);
AMaskEvent(audio, emask, &event_return, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

AMaskEvent() was developed by HP.

SEE ALSO

ACheckEvent(3X), ACheckMaskEvent(3X), AEventsQueued(3X), ANextEvent(3X), APeekEvent(3X), APutBackEvent(3X), AQlength(3X), ASelectInput(3X).

Using the Audio Application Program Interface.

NAME

AMaxInputGain - get maximum input gain supported by audio controller

SYNOPSIS

```
#include <audio/Alib.h>

AGainDB AMaxInputGain(Audio *audio);
```

DESCRIPTION

AMaxInputGain() gets the maximum input gain, in decibels, supported by the audio controller associated with the *audio* connection. If the application specifies a gain higher than this, the maximum supported value is used.

audio specifies the **Audio** structure associated with this connection.

RETURN VALUE

Upon successful completion, **AMaxInputGain()** returns the maximum input gain, in decibels, that the audio controller supports.

ERRORS

AMaxInputGain does not return an error status.

EXAMPLES

The following example gets the maximum input gain supported by the audio controller:

```
AGainDB max_in; /* max input gain */
Audio *audio; /* audio connection */
      .
      .
      .
/* get max input gain */
max_in = AMaxInputGain(audio);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

AMaxInputGain() was developed by HP.

SEE ALSO

AGetChannelGain(3X) AGetGain(3X), AGetSystemChannelGain(3X), AGMGainRestricted(3X), AInputChannels(3X), AInputSources(3X), AMaxOutputGain(3X), AMinInputGain(3X), AMinOutputGain(3X), AOutputChannels(3X), AOutputDestinations(3X), ASetChannelGain(3X), ASetGain(3X), ASetSystemChannelGain(3X), ASimplePlayer(3X), ASimpleRecorder(3X).

Using the Audio Application Program Interface.

NAME

AMaxOutputGain - get maximum output gain supported by audio controller

SYNOPSIS

```
#include <audio/Alib.h>

AGainDB AMaxOutputGain(Audio *audio);
```

DESCRIPTION

AMaxOutputGain() returns the maximum output gain, in decibels, supported by the audio controller associated with the *audio* connection. If the application specifies a gain higher than this, the supported maximum value is used.

audio specifies the **Audio** structure associated with this connection.

RETURN VALUE

Upon successful completion, **AMaxOutputGain()** returns the maximum output gain, in decibels, that the audio controller supports.

ERRORS

AMaxOutputGain does not return an error status.

EXAMPLES

The following example gets the maximum output gain supported by the audio controller:

```
AGainDB max_out; /* max output gain */
Audio *audio; /* audio connection */
      .
      .
      .
/* get max output gain */
max_out = AMaxOutputGain(audio);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

AMaxOutputGain() was developed by HP.

SEE ALSO

AGetChannelGain(3X) AGetGain(3X), AGetSystemChannelGain(3X), AGMGainRestricted(3X), AInputChannels(3X), AInputSources(3X), AMaxInputGain(3X), AMinInputGain(3X), AMinOutputGain(3X), AOutputChannels(3X), AOutputDestinations(3X), ASetChannelGain(3X), ASetGain(3X), ASetSystemChannelGain(3X), ASimplePlayer(3X), ASimpleRecorder(3X).

Using the Audio Application Program Interface.

NAME

AMinInputGain - get minimum input gain supported by audio controller

SYNOPSIS

```
#include <audio/Alib.h>

AGainDB AMinInputGain(Audio *audio);
```

DESCRIPTION

AMinInputGain() returns the minimum input gain supported by the audio controller associated with the *audio* connection. If the application specifies a gain lower than this, the gain is set to **AZeroGain**, which results in no sound.

audio specifies the **Audio** structure associated with this connection.

RETURN VALUE

Upon successful completion, **AMinInputGain()** returns the minimum input gain, in decibels, that the audio controller supports.

ERRORS

AMinInputGain() does not return an error status.

EXAMPLES

The following example gets the minimum input gain supported by the audio controller:

```
AGainDB min_in; /* min input gain */
Audio *audio; /* audio connection */
      .
      .
      .
/* get min input gain */
min_in = AMinInputGain(audio);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

AMinInputGain() was developed by HP.

SEE ALSO

AGetChannelGain(3X) AGetGain(3X), AGetSystemChannelGain(3X), AGMGainRestricted(3X), AInputChannels(3X), AInputSources(3X), AMaxInputGain(3X), AMaxOutputGain(3X), AMinOutputGain(3X), AOutputChannels(3X), AOutputDestinations(3X), ASetChannelGain(3X), ASetGain(3X), ASetSystemChannelGain(3X), ASimplePlayer(3X), ASimpleRecorder(3X).

Using the Audio Application Program Interface.

NAME

AMinOutputGain - get minimum output gain supported by audio controller

SYNOPSIS

```
#include <audio/Alib.h>

AGainDB AMinOutputGain(Audio *audio);
```

DESCRIPTION

AMinOutputGain() returns the minimum output gain, in decibels, supported by the audio controller associated with the *audio* connection. If the application specifies a gain lower than this, the gain is set to **AZeroGain**, which results in no sound.

audio specifies the **Audio** structure associated with this connection.

RETURN VALUE

Upon successful completion, **AMinOutputGain()** returns the minimum output gain, in decibels, that the audio controller supports.

ERRORS

AMinOutputGain() does not return an error status.

EXAMPLES

The following example gets the minimum output gain supported by the audio controller:

```
Audio *audio; /* audio connection */
AGainDB min_out; /* min output gain */
      .
      .
      .
/* get min output gain */
min_out = AMinOutputGain(audio);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

AMinOutputGain() was developed by HP.

SEE ALSO

AGetChannelGain(3X) AGetGain(3X), AGetSystemChannelGain(3X), AGMGainRestricted(3X), AInputChannels(3X), AInputSources(3X), AMaxInputGain(3X), AMaxOutputGain(3X), AMinInputGain(3X), AOutputChannels(3X), AOutputDestinations(3X), ASetChannelGain(3X), ASetGain(3X), ASetSystemChannelGain(3X), ASimplePlayer(3X), ASimpleRecorder(3X).

Using the Audio Application Program Interface.

NAME

ANextEvent - dequeue and return first event in audio event queue

SYNOPSIS

```
#include <audio/Alib.h>

void ANextEvent(
    Audio *audio,
    AEvent *event_return,
    long *status_return
);
```

DESCRIPTION

ANextEvent() dequeues and returns the first event in the audio event queue. If no match is found, the function blocks until an event is received. (This behavior is unlike **ACheckEvent()** and **ACheckMaskEvent()** which do not block if there is no event or matching event, respectively.)

audio specifies the **Audio** structure associated with this connection.

event_return is the first event found in the queue.

status_return receives the returned status of the operation, unless it is set to NULL.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return:*

0 AENoError
2 AEBadAudio

EXAMPLES

The following example sets up *event_return* to receive event data and *status_return* to receive an error status:

```
Audio *audio; /* audio connection */
AEvent event_return; /* event return */
long status; /* error status */
        .
        .
        .
/* check event queue */
ANextEvent(audio, &event_return, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

ANextEvent() was developed by HP.

SEE ALSO

ACheckEvent(3X), ACheckMaskEvent(3X), AEventsQueued(3X), AMaskEvent(3X), APeekEvent(3X), APutBackEvent(3X), AQlength(3X), ASelectInput(3X).

Using the Audio Application Program Interface.

NAME

ANumDataFormats - return number of data formats supported by audio controller

SYNOPSIS

```
#include <audio/Alib.h>

long ANumDataFormats (Audio *audio);
```

DESCRIPTION

ANumDataFormats() returns the number of data formats supported by the audio controller associated with the connection specified by *audio*.

A list of the data formats is obtained using the function **ADataFormats()**.

audio specifies the **Audio** structure associated with this connection.

ERRORS

ANumDataFormats() does not return an error status.

EXAMPLES

The following example gets the number of data formats supported by the audio controller associated with *audio*.

```
long dfnum;             /* number of supported data formats */
Audio *audio;           /* audio connection */
      .
      .
      .
/* get number of data formats supported by controller */
dfnum = ANumDataFormats(audio);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

ANumDataFormats() was developed by HP.

SEE ALSO

ADataFormats(3X).

Using the Audio Application Program Interface.

NAME

ANumSamplingRates - return number of sampling rates supported by audio controller

SYNOPSIS

```
#include <audio/Alib.h>

long ANumSamplingRates (Audio *audio);
```

DESCRIPTION

ANumSamplingRates() returns the number of sampling rates supported by the audio controller associated with the connection specified by *audio*. Zero is returned if sampled data is not supported by the controller.

A list of the supported sampling rates is obtained using the function **ASamplingRates()**.

audio specifies the **Audio** structure associated with this connection.

RETURN VALUE

Upon successful completion, **ANumSamplingRates()** returns the number of sampling rates supported by the audio controller associated with the connection specified by *audio*.

ERRORS

ANumSamplingRates() does not return an error status.

EXAMPLES

The following example gets the number of sampling rates supported by the audio controller associated with *audio*.

```
long srnum;             /* number of supported sampling rates */
Audio *audio;           /* audio connection */
      .
      .
      .
/* get number of sampling rates supported by controller */
srnum = ANumSamplingRates(audio);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

ANumSamplingRates() was developed by HP.

SEE ALSO

ASamplingRates(3X).

Using the Audio Application Program Interface.

NAME

AOpenAudio - open connection to specified audio server

SYNOPSIS

```
#include <audio/Alib.h>

Audio *AOpenAudio(char *audio_name, long *status_return);
```

DESCRIPTION

AOpenAudio() opens a connection to the server for the specified audio controller and returns a pointer to an **Audio** structure. The audio library allocates the **Audio** structure to hold information that supports the controller. The structure acts as the information connection between the application and the server; the application passes the **Audio** pointer to subsequent audio library function calls to identify which connection the call should affect.

NOTE: If the audio server is not active, this function returns a NULL and sets *status_return* to **AEOpenFailed**. For this reason, the application should use *status_return* (not set it to NULL) and should check it before proceeding.

audio_name specifies the audio controller name as a string. If *audio_name* is specified NULL, the value of the **AUDIO** environment variable is used.

The string format is *hostname*:*number*

where:

hostname specifies the name of the host machine on which the audio controller is physically installed.

number specifies the audio server number on that host machine. Each audio server services one audio controller. More than one audio controller can be installed in a machine. The audio servers are numbered starting with 0.

status_return receives the returned status of the operation unless it is set to NULL.

One successful call to **AOpenAudio()** must precede all other audio operation function calls pertaining to a connection.

To close the connection, use **ACloseAudio()**.

EXTERNAL INFLUENCES

If *audio_name* is specified NULL, the value of the **AUDIO** environment variable is used.

RETURN VALUE

Upon successful completion, **AOpenAudio()** returns a pointer to an **Audio** structure. Otherwise, it returns a NULL.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return:*

0	AENoError
1	AESystemCall
4	AEHostNotFound
5	AENoSuchAudioNumber
17	AEOutOfMemory
18	AEOpenFailed
23	AEllbdNotStarted

AOpenAudio() does not generate error events.

EXAMPLES

The following example sets the audio name argument, *a_name*, to NULL, causing the value of the AUDIO environment variable to be used; *status* is set up to receive an error status.

```
Audio *audio;      /* audio connection */ char a_name;      /* audio
name */ long status;      /* error status */      .      .      .
/* open audio connection */ a_name = ""; audio = AOpenAudio(a_name,
&status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

`AOpenAudio()` was developed by HP.

SEE ALSO

ACloseAudio(3X).

Using the Audio Application Program Interface.

NAME

AOutputChannels - get D/A output channels existing on current hardware

SYNOPSIS

```
#include <audio/Alib.h>

AOutputChMask AOutputChannels(Audio *audio);
```

DESCRIPTION

AOutputChannels() returns a mask showing the Digital and/or Analog output channels that exist on the current hardware. Each bit in the returned **AOutputChMask** correponds to one output channel.

audio specifies the **Audio** structure associated with this connection.

RETURN VALUE

Upon successful completion, **AOutputChannels()** returns a mask showing the output channels that exist on the current hardware: mono, left, or right output. Each bit in the returned **AOutputChMask** correponds to one type of output channel.

ERRORS

AOutputChannels() does not return an error status.

EXAMPLES

The following example gets the types of output channels that exist on the current hardware.

```
AOutputChMask out_channels;    /* output channel mask */
Audio *audio;                  /* audio connection */
      .
      .
      .
/* get output sources */
out_channels = AOutputChannels(audio);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

AOutputChannels() was developed by HP.

SEE ALSO

AGetChannelGain(3X) AGetGain(3X), AGetSystemChannelGain(3X), AGMGainRestricted(3X), AInputChannels(3X), AInputSources(3X), AMaxInputGain(3X), AMaxOutputGain(3X), AMinInputGain(3X), AMinOutputGain(3X), AOutputDestinations(3X), ASetChannelGain(3X), ASetGain(3X), ASetSystemChannelGain(3X), ASimplePlayer(3X), ASimpleRecorder(3X).

Using the Audio Application Program Interface.

NAME

AOutputDestinations - get types of output destinations existing on current hardware

SYNOPSIS

```
#include <audio/Alib.h>

AOutputDstMask AOutputDestinations(Audio *audio);
```

DESCRIPTION

AOutputDestinations() returns a mask showing the types of output destinations that exist on the current hardware. Each bit in the returned **AOutputDstMask** correponds to one type of output destination.

audio specifies the **Audio** structure associated with this connection.

RETURN VALUE

Upon successful completion, **AOutputDestinations()** returns a mask showing the output destinations that exist on the current hardware: mono, left, or right headphone jacks, and mono, left, or right internal speakers. Each bit in the returned **AOutputDstMask** correponds to one type of output destination.

ERRORS

AOutputDestinations() does not return an error status.

EXAMPLES

The following example gets the types of output destination that exist on the current hardware.

```
AOutputDstMask dests;     /* output destination mask */
Audio *audio;             /* audio connection */
      .
      .
      .
/* get output destinations */
dests = AOutputDestinations(audio);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

AOutputDestinations() was developed by HP.

SEE ALSO

AGetChannelGain(3X) AGetGain(3X), AGetSystemChannelGain(3X), AGMGainRestricted(3X), AInputChannels(3X), AInputSources(3X), AMaxInputGain(3X), AMaxOutputGain(3X), AMinInputGain(3X), AMinOutputGain(3X), AOutputChannels(3X), ASetChannelGain(3X), ASetGain(3X), ASetSystemChannelGain(3X), ASimplePlayer(3X), ASimpleRecorder(3X).

Using the Audio Application Program Interface.

NAME

APauseAudio - pause the specified audio transaction

SYNOPSIS

```
#include <audio/Alib.h>

void APauseAudio (
    Audio *audio,
    ATransID xid,
    ATransStatus *trans_status_return,
    long *status_return
);
```

DESCRIPTION

APauseAudio() pauses the transaction specified in *xid*. To continue with the operation, call **AResumeAudio()**.

While one transaction is paused, another transaction can play or record.

To stop the transaction so that it cannot be resumed, call **AStopAudio()**.

audio specifies the **Audio** structure associated with this connection.

xid specifies the transaction ID.

To use **APauseAudio()** on a series of linked transactions, specify the first transaction in the linked list. The pause affects the current transaction. A call to **AResumeAudio()** resumes the transaction and continues through the linked list.

trans_status_return receives the returned status value. Setting this argument to NULL prevents the data from being collected and returned, which may enhance performance.

status_return receives the returned status of the operation, unless it is set to NULL.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return*.

0 AENoError
2 AEBadAudio
15 AEBadTransactionID

EXAMPLES

The following example pauses the transaction identified by *xid*, sets *trans_stat* to 0, and sets up *status* to receive an error status.

```
Audio *audio; /* audio connection */"
ATransID xid; /* transaction ID */"
ATransStatus trans_stat_return; /* transaction status return
long status; /* error status */"
      .
      .
      .
/* pause transaction - xid returned by prior call */
trans_stat = 0;
APauseAudio(audio, xid, &trans_stat_return, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

APauseAudio() was developed by HP.

SEE ALSO

AResumeAudio(3X), AStopAudio(3X).

Using the Audio Application Program Interface.

NAME

APeekEvent - return but do not dequeue first event in audio event queue

SYNOPSIS

```
#include <audio/Alib.h>

void APeekEvent (
    Audio *audio,
    AEvent *event_return,
    long *status_return
);
```

DESCRIPTION

APeekEvent() returns, but does not dequeue, the first event in the audio event queue. If no match is found, this function blocks until a matching event is received. This behavior is unlike **ACheckEvent()**, **ACheckMaskEvent()**, and **ANextEvent()**, which dequeue an event from the queue when they return it, and **ACheckEvent()**, and **ACheckMaskEvent()**, which do not block if there is no event or matching event, respectively.

audio specifies the **Audio** structure associated with this connection.

event_return is the first event in the audio event queue.

status_return receives the returned status of the operation, unless it is set to NULL.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return:*

0 AENoError
2 AEBadAudio

EXAMPLES

The following example sets up *event* to receive the event copy and *status_return* to receive an error status.

```
Audio *audio;           /* audio connection */
AEvent event_return;    /* event_return */
long status;            /* error status */
      .
      .
      .
/* copy first event on queue */
APeekEvent(audio, &event_return, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

APeekEvent() was developed by HP.

SEE ALSO

ACheckEvent(3X), ACheckMaskEvent(3X), AEventsQueued(3X), AMaskEvent(3X), ANextEvent(3X), APutBackEvent(3X), AQlength(3X) ASelectInput(3X).

Using the Audio Application Program Interface.

NAME

APlaySBucket - play specified sound bucket and return transaction ID

SYNOPSIS

```
#include <audio/Alib.h>

ATransID APlaySBucket (
    Audio *audio,
    SBucket *sb,
    SBPlayParams *pp,
    long *status_return
);
```

DESCRIPTION

APlaySBucket() plays the audio data in the specified sound bucket on the specified server connection and returns a transaction ID.

audio specifies the **Audio** structure associated with this connection.

sb specifies the sound bucket to be played.

pp specifies the play parameters associated with the play operation.

status_return receives the returned status of the operation, unless it is set to NULL.

RETURN VALUE

Upon successful completion, **APlaySBucket()** returns the transaction ID.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return:*

0 AENoError
2 AEBadAudio
10 AEBadGainMatrix
20 AEBadSoundBucket

EXAMPLES

The following example plays the audio data contained in the sound bucket specified by *sb* and returns a transaction ID.

```
ATransID xid;              /* transaction ID */
Audio *audio;              /* audio connection */
SBucket *sb;               /* sound bucket*/
SBPlayParams *pparams;     /* play parameters */
long status;               /* error status */
      .
      .
      .
/* play sound bucket */
xid = APlaySBucket(audio, sb, pparams, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

APlaySBucket() was developed by HP.

SEE ALSO

ACreateSBucket(3X), ADestroySBucket(3X), AGetSBucketData(3X), ALoadAFile(3X), APutSBucketData(3X), ARecordAData(3X), ASaveSBucket(3X).

Using the Audio Application Program Interface.

NAME

APlaySStream - initiate transaction and return transaction ID and SStream structure

SYNOPSIS

```
#include <audio/Alib.h>

ATransID APlaySStream(
    Audio *audio,
    AudioAttrMask attr_mask,
    AudioAttributes *audio_attributes,
    SSPlayParams *pp,
    SStream *sstream_return,
    long *status_return
);
```

DESCRIPTION

APlaySStream() initiates a play sound stream transaction and returns a transaction ID and an **SStreams** structure that contains a TCP socket address.

The application connects the socket it has created to the TCP address. The play operation begins as soon as there is data on the sound stream. The play stream transaction can be controlled using **APauseAudio()**, **AResumeAudio()**, and **AStopAudio()**.

audio specifies the **Audio** structure associated with this connection.

attr_mask specifies which elements of the *audio_attributes* structure to use; it is the bitwise inclusive OR of the valid audio attribute masks.

If *attr_mask* is zero, the values in the **AudioAttributes** structure returned by **ABestAudioAttributes()** are used.

audio_attributes

contains values for type and sampled attributes. Type *must* be set, separate from the mask.

If *audio_attributes* is NULL, the values in the **AudioAttributes** structure returned by **ABestAudioAttributes()** are used; values in this structure are also used for unspecified attributes.

pp is the pointer to the play parameters associated with the play operation.

status_return receives the returned status of the operation unless it is set to NULL.

RETURN VALUE

Upon successful completion, **APlaySStream()** returns a transaction ID.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return:*

0	AENoError
2	AEBadAudio
7	AEBadDataFormat
10	AEBadGainMatrix
13	AEBadAttribute

EXAMPLES

The following example starts a play stream transaction and sets up *sstream* to receive the **SStream** structure and *status* to receive an error status return.

```
ATransID xid;                   /* transID */
Audio *audio;                   /* audio connection */
AudioAttrMask a_mask;           /* audio attribute mask */
AudioAttributes *attribs;       /* audio attributes*/
SSPlayParams ss_pp;             /* sstream play parameters */
SStream sstream;                /* sstream structure */
long status;                    /* error status */
```

```
        .
        .
/* play sstream */
xid = APlaySStream(audio, a_mask, attribs, ss_pp, &sstream, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

`APlaySStream()` was developed by HP.

SEE ALSO

AConnectRecordSStream(), ARecordSStream().

Using the Audio Application Program Interface.

NAME

AProtocolRevision - get minor revision number of protocol used by audio server

SYNOPSIS

```
#include <audio/Alib.h>

long AProtocolRevision(Audio *audio);
```

DESCRIPTION

`AProtocolRevision()` returns the minor revision number of the protocol used by the audio server for the connection specified by *audio*.

audio specifies the `Audio` structure associated with this connection.

RETURN VALUE

Upon successful completion, `AProtocolRevision()` returns the minor revision number of the protocol for the audio server associated with this connection.

ERRORS

`AProtocolRevision()` does not return an error status.

EXAMPLES

The following example returns the minor revision number of the protocol associated with the audio server for the connection specified by *audio*.

```
long p_rev;       /* minor protocol revision */
Audio *audio;     /* audio connection */
      .
      .
      .
/* get minor protocol revision */
p_rev = AProtocolRevision(audio);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

`AProtocolRevision()` was developed by HP.

SEE ALSO

AProtocolVersion(3X), AServerVendor(3X), AVendorRelease(3X).

Using the Audio Application Program Interface.

NAME

AProtocolVersion - get major version number of protocol used by audio server

SYNOPSIS

```
#include <audio/Alib.h>

long AProtocolVersion (Audio *audio);
```

DESCRIPTION

AProtocolVersion() returns the major version number of the protocol used by the audio server for the connection specified by *audio*.

audio specifies the **Audio** structure associated with this connection.

RETURN VALUE

Upon successful completion, **AProtocolVersion()** returns the major version number of the protocol for the audio server associated with this connection.

ERRORS

AProtocolVersion() does not return an error status.

EXAMPLES

The following example returns the major version number of the protocol associated with the audio server for the connection specified by *audio*.

```
long p_version;         /* major protocol version */
Audio *audio;           /* audio connection */
      .
      .
      .
/* get major protocol version */
p_version = AProtocolVersion(audio);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

AProtocolVersion() was developed by HP.

SEE ALSO

AProtocolRevision(3X), AServerVendor(3X), AVendorRelease(3X).

Using the Audio Application Program Interface.

NAME

APutBackEvent - push event onto head of audio event queue

SYNOPSIS

```
#include <audio/Alib.h>

void APutBackEvent (
    Audio *audio,
    AEvent *event,
    long *status_return
);
```

DESCRIPTION

`APutBackEvent()` pushes *event* onto the head of the audio event queue for the server specified by *audio*.

audio specifies the `Audio` structure associated with this connection.

event is the event to put on the queue.

status_return receives the returned status of the operation, unless it is set to NULL.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return*:

0 AENoError
2 AEBadAudio

EXAMPLES

The following example puts *event* at the head of the audio event queue and sets up *status_return* to receive an error status.

```
Audio *audio;          /* audio connection */
AEvent *event;         /* event */
long status;           /* error status */
      .
      .
      .
/* put event at head of queue */
event = event_return;  /* use event_return value from prior call */
APutBackEvent(audio, event, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

`APutBackEvent()` was developed by HP.

SEE ALSO

ACheckEvent(3X), ACheckMaskEvent(3X), AEventsQueued(3X), AMaskEvent(3X), ANextEvent(3X), APeekEvent(3X), AQLength(3X), ASelectInput(3X).

Using the Audio Application Program Interface.

NAME

APutSBucketData - copy audio data from buffer to sound bucket

SYNOPSIS

```
#include <audio/Alib.h>

unsigned long APutSBucketData (
    Audio *audio,
SBucket *sb,
    unsigned long start_offset,
    char *buffer,
    unsigned long length,
    long *status_return
);
```

DESCRIPTION

APutSBucketData() copies the data from a buffer to a sound bucket.

audio specifies the **Audio** structure associated with this connection.

sb specifies the sound bucket to receive the data.

start_offset specifies where to start writing the copied data, given as the byte offset from the beginning of the sound bucket.

buffer specifies the buffer containing the data to copy.

length specifies the length of the data in the buffer, in bytes.

status_return receives the returned status of the operation, unless it is set to NULL.

RETURN VALUE

Upon successful completion, **APutSBucketData()** returns the byte count of the copied data.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return*:

0 AENoError
2 AEBadAudio
20 AEBadSoundBucket

EXAMPLES

The following example copies the audio data from the buffer *buff* to the sound bucket *sb* and returns the number of bytes that were copied. The data is placed starting at the beginning of the sound bucket (offset 0). In this example, we assume that we are returning data from the buffer at *bufp* that was written there by **AGetSBucketData()**. We use the *datalen_g* value returned by **AGetSBucketData()** as the length of the data.

```
unsigned long datalen_p;  /* copied data length */
Audio *audio;               /* audio connection */
SBucket *sb;                /* sound bucket*/
unsigned long startoff;     /* start offset */
char *bufp;                 /* ptr to buffer */
long status;                /* error status */
      .
      .
      .
/* copy data from buffer to sb */
startoff = 0;
datalen_p = APutSBucketData(audio, sb, startoff, bufp, datalen_g, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

`APutSBucketData()` was developed by HP.

SEE ALSO

ACreateSBucket(3X), ADestroySBucket(3X), AGetSBucketData(3X), ALoadAFile(3X), APlaySBucket(3X), ARecordAData(3X), ASaveSBucket(3X).

Using the Audio Application Program Interface.

NAME

AQLength - return number of events on audio event queue

SYNOPSIS

```
#include <audio/Alib.h>

int AQLength (Audio *audio);
```

DESCRIPTION

AQLength() returns number of events on the audio event queue for the audio server connection specified by *audio*.

audio specifies the **Audio** structure associated with this connection.

RETURN VALUE

Upon successful completion, **AQLength()** returns the number of events on the audio event queue for the *audio* connection.

ERRORS

AQLength() does not return an error status.

EXAMPLES

The following example gets the number of events on the audio event queue.

```
int e_num;          /* number of events */
Audio *audio;       /* audio connection */
      .
      .
      .
/* get number of events on queue */
e_num = AQLength(audio);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

AQLength() was developed by HP.

SEE ALSO

ACheckEvent(3X), ACheckMaskEvent(3X), AEventsQueued(3X), AMaskEvent(3X), ANextEvent(3X), APeekEvent(3X), APutBackEvent(3X), ASelectInput(3X).

Using the Audio Application Program Interface.

NAME

AQueryAFile - get file format of specified file

SYNOPSIS

```
#include <audio/Alib.h>

AFileFormat AQueryAFile(Audio *audio, char *name, long *status_return);
```

DESCRIPTION

AQueryAFile() returns the file format of the file specified in *name*.

audio specifies the **Audio** structure associated with this connection.

name is the pathname of the audio data file to be queried.

status_return receives the returned status of the operation, unless it is set to NULL.

RETURN VALUE

Upon successful completion, **AQueryAFile()** returns the file format of the file specified in *name*. **AFFUnknown** is returned if the format type cannot be determined.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return*:

0 AENoError
2 AEBadAudio
8 AEFileNotFound
16 AECantDetermineFormat

EXAMPLES

The following example queries the file format of the file **/myhome/a_dir/a_file**:

```
ift .ft 4 AFileFormat file_fmt; /* file format */ Audio *audio;  /* audio connection */ long status;
/* status */  .  .  . /* load file into new sound bucket */ charfname[] =
"/myhome/a_dir/a_file"; file_fmt = AQueryAFile(audio, fname, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

AQueryAFile() was developed by HP.

SEE ALSO

AGetAFileAttributes() *Using the Audio Application Program Interface*.

NAME

ARecordAData - read audio data into sound bucket

SYNOPSIS

```
#include <audio/Alib.h>

ATransID ARecordAData (
    Audio *audio,
    SBucket *sb,
    SBRecordParams *rp,
    long *status_return
);
```

DESCRIPTION

ARecordAData() reads audio data from the specified server connection into the specified sound bucket and returns a transaction ID. **ARecordAData()** does not block until the record is complete, and so it can not be followed immediately by a call to **ASaveSBucket()**. See *ASaveSBucket*(3X) for suggested program actions.

audio specifies the **Audio** structure associated with this connection.

sb specifies the sound bucket to receive the data.

rp specifies the record parameters associated with the record operation.

status_return receives the returned status of the operation, unless it is set to NULL.

RETURN VALUE

Upon successful completion, **ARecordAData()** returns the transaction ID.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return*:

0 AENoError
2 AEBadAudio
10 AEBadGainMatrix
20 AEBadSoundBucket

EXAMPLES

The following example reads data from the connection associated with *audio* into the sound bucket specified by *sb* and returns a transaction ID.

```
TransID xid; /* transID */
Audio *audio;            /* audio connection */
SBucket *sb;            /* sound bucket*/
SBRecordParams rparams; /* record parameters */
long status;             /* error status */
      .
      .
      .
/* start record transaction */
xid = ARecordAData(audio, sb, &rparams, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

ARecordAData() was developed by HP.

SEE ALSO

ACreateSBucket(3X), ADestroySBucket(3X), AGetSBucketData(3X), ALoadAFile(3X), APlaySBucket(3X), APutSBucketData(3X), ASaveSBucket(3X).

Using the Audio Application Program Interface.

NAME

ARecordSStream - initiate transaction; return transaction ID and SStreams structure

SYNOPSIS

```
#include <audio/Alib.h>

ATransID ARecordSStream(
    Audio *audio,
    AudioAttrMask attr_mask,
    AudioAttributes *audio_attributes,
    SSRecordParams *rp,
    SStream *sstream_return,
    long *status_return
);
```

DESCRIPTION

ARecordSStream() initiates a sound stream record transaction and returns a transaction ID and an **SStream** structure that contains a TCP socket address.

The application connects the socket it has created to the TCP address. The record operation begins immediately or in pause mode, depending on the *pause_first* field in **SSRecordParams**. The record stream transaction can be controlled using **APauseAudio()**, **AResumeAudio()**, and **AStopAudio()**.

audio specifies the **Audio** structure associated with this connection.

attr_mask specifies which elements of the *audio_attributes* structure to use; it is the bitwise inclusive OR of the valid audio attribute masks.

If *attr_mask* is zero, the values in the **AudioAttributes** structure returned by **ABestAudioAttributes()** are used.

audio_attributes

contains values for type and sampled attributes. Type *must* be set, separate from the mask.

If *audio_attributes* is NULL, the values in the **AudioAttributes** structure returned by **ABestAudioAttributes()** are used; values in this structure are also used for unspecified attributes.

rp specifies the record parameters associated with the record operation.

sstream_return receives the returned **SStream** structure.

status_return receives the returned status of the operation unless it is set to NULL.

RETURN VALUE

Upon successful completion, **ARecordSStream()** returns a transaction ID.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return:*

0	AENoError
2	AEBadAudio
7	AEBadDataFormat
10	AEBadGainMatrix
13	AEBadAttribute

EXAMPLES

The following example starts a record stream transaction, setting up *sstream* to receive the **SStream** structure and *status* to receive an error status return.

```
ATransID xid;                   /* transID */
Audio *audio;                   /* audio connection */
AudioAttrMask a_mask;           /* audio attribute mask */
AudioAttributes attribs;        /* audio attributes*/
SSRecordParams ss_rp;           /* sstream record parameters */
SStream sstream;                /* sstream structure */
long status;                    /* error status */
```

```
        .
        .
        .
    /* record sstream */
    xid = ARecordSStream(audio, a_mask, &attribs, &ss_rp, &sstream, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

`ARecordSStream()` was developed by HP.

SEE ALSO

AConnectRecordSStream(), APlaySStream().

Using the Audio Application Program Interface.

NAME

AResumeAudio - resume specified audio transaction

SYNOPSIS

```
#include <audio/Alib.h>

void AResumeAudio (
    Audio *audio,
    ATransID xid,
    ATransStatus *trans_status_return,
    long *status_return
);
```

DESCRIPTION

AResumeAudio() resumes the specified transaction if the transaction was paused by **APauseAudio()**.

audio specifies the **Audio** structure associated with this connection.

xid specifies the transaction ID.

To use **AResumeAudio()** on a paused series of linked transactions, specify the first transaction in the linked list. The resume affects the current (paused) transaction.

trans_status_return receives the returned status value. Setting this argument to NULL prevents the data from being collected and returned, which may enhance performance.

status_return receives the returned status of the operation unless it is set to NULL.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return*:

0 AENoError
2 AEBadAudio
15 AEBadTransactionID

EXAMPLES

The following example resumes the transaction identified by *xid*, sets *trans_stat* to NULL, and sets up *status* to receive an error status.

```
Audio *audio;          /* audio connection */
ATransID xid;          /* transaction ID */
long status;           /* error status */
      .
      .
      .
/* resume transaction - xid returned from prior call */
AResumeAudio(audio, xid, NULL, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

AResumeAudio() was developed by HP.

SEE ALSO

APauseAudio(3X), AStopAudio(3X).

Using the Audio Application Program Interface.

NAME

ASamplingRates - return array of sampling rates supported by audio controller

SYNOPSIS

```
#include <audio/Alib.h>

unsigned long* ASamplingRates(Audio *audio);
```

DESCRIPTION

ASamplingRates() returns a pointer to an array of sampling rates supported by the audio controller associated with the *audio* connection.

The number of sampling rates in the array is obtained using the function **ANumSamplingRates()**.

audio specifies the **Audio** structure associated with this connection.

RETURN VALUE

Upon successful completion, **ASamplingRates()** returns a pointer to an array of sampling rates supported by the audio controller associated with the connection specified by *audio*.

ERRORS

ASamplingRates() does not return an error status.

EXAMPLES

The following example returns an array containing the sampling rates supported by the audio controller associated with *audio*.

```
unsigned long *s_rates, first_rate; /* supported sampling rates */
Audio *audio;                       /* audio connection */
      .
      .
      .
/* get pointer to array of sampling rates */
s_rates = ASamplingRates(audio);
/* get first sampling rate */
first_rate = s_rates[0];
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

ASamplingRates() was developed by HP.

SEE ALSO

ANumSamplingRates(3X).

Using the Audio Application Program Interface.

NAME

ASaveSBucket - write sound bucket data into file with data conversion

SYNOPSIS

```
#include <audio/Alib.h>

void ASaveSBucket(
    Audio *audio,
    SBucket *sb,
    char *pathname,
    AFileFormat file_format,
    AudioAttrMask attr_mask,
    AudioAttributes *target_attributes,
    ATime *offset,
    AWriteMode mode,
    long *status_return
);
```

DESCRIPTION

ASaveSBucket() writes the data in *sb* into the file specified by *pathname* after making conversions according to the specified attributes.

ARecordAData() does not block until the record operation is complete, so calling **ASaveSBucket()** immediately after **ARecordAData()** usually results in an error. To avoid the error, the application program can set up its own wait loop, or wait on a transaction-stopped or transaction-completed event. See the program **/usr/audio/examples/recorder.c** for an example of waiting either for a transaction-stopped event or for the user to terminate the record operation by pressing [Return].

When the sound bucket is no longer needed, call **ADestroySBucket()** to deallocate the space.

audio is the audio structure associated with this connection.

sb is the sound bucket that contains the audio data and associated attributes.

pathname specifies the file to receive the data. If the file does not exist, it is created.

file_format specifies the file format to use for the write. If this parameter is not set to a valid enumerated value, an error is returned.

If this parameter is set to **AFFUnknown**, the conversion utility checks for an extension on *pathname*. Extensions can be appended to the filename as follows:

name.sampling_rate.file_type.

Valid file type extensions are:

.u	Mulaw
.al	Alaw
.au	Sun (NeXT)
.wav	Riff
.snd	NeXT
.116	Linear16
.18	Linear8
.lo8	Linear8Offset

If no recognizable extension exists and *pathname* is an existing file, the utility checks the header on the existing file. If there is no determinable file format, an error is returned; there is no default.

If this parameter specifies a different format than the one indicated by an existing file in *pathname,* (unless *mode* is **AWMTruncateAppend** and *offset* is 0), an error is returned.

attr_mask specifies the audio attributes to associate with the data written to the file; conversion occurs where necessary. The mask is a bitwise inclusive OR of the values defined in **AudioAttrMask**. This mask is cleared if *target_attributes* is set to NULL.

When a mask bit is set to 0 and *pathname* is an existing file, the conversion utility checks the existing file type in the file header and uses a value that is compatible with it.

When a mask bit is set to 0 and *pathname* is not an existing file, the conversion utility checks the file type indicated by the *pathname* extension, if any, and uses a value that is compatible with it. If no value can be determined, the sound bucket value for the attribute is used.

When a mask bit is set to 1 and *pathname* is not an existing file, the specified attribute is checked for compatibility with the existing file. An error is returned if there is a discrepancy.

When a mask bit is set to 1 and a file does not exist in *pathname,* the specified attribute is used.

NOTES: If **ASDurationMask** is set, the sound bucket data is truncated or padded with zeros to match the length specified in **audio_attributes.sampled_attr.duration**.

If **ASSamplingRateMask** is set, it is used without checking the file name extension. If *sampling_rate* is not specified, the file name is checked for an extension. Sampling rate attributes can be specified in a filename extension as follows:

name . *sampling_rate* . *file_type*

Valid sampling rate extensions are *.n* and *.n***k** where *.n***k** is typically 8k to 22k.

target_attributes specifies the attributes that are affected by the mask. If set to NULL, *attr_mask* is cleared and attributes are determined according to compatibility with *pathname* and *sb* (see *attr_mask).* Audio type *must* be set (separate from the mask).

offset specifies where to begin writing in the destination file, given in **ATimeType** units (**ATTSamples**, **ATTMilliSeconds**, or **ATTFullLength**) from the beginning of the audio data, excluding the header.

If *pathname* is not an existing file and *offset* is not 0, the new file is padded with zeros up to the offset.

If *pathname* is an existing file and *offset* is greater than the length of the audio data, zeros are appended to the audio data until its length is equal to *offset*.

mode specifies how the data should be written into the file:

AWMOverWrite specifies that data from the sound bucket *sb* overwrites the data in *pathname* starting at *offset*. Data that precedes or follows the overwritten region remains unchanged. If necessary, the length of the file is increased to accommodate the new data.

AWMTruncateAppend specifies that the data in *pathname* is truncated at *offset* and the write begins at that point. If necessary, the length of the file is increased or decreased to accommodate the new data.

AWMInsert specifies that data from the sound bucket *sb* is inserted in the file *pathname* starting at *offset*. The length of the file is increased to accommodate the new data.

status_return receives the returned status of the operation, unless it is set to NULL.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return:*

0 AENoError
2 AEBadAudio
6 AEBadFileFormat
7 AEBadDataFormat
8 AEFileNotFound

11 AEBadFileHdr
13 AEBadAttribute
14 AEBadOffset
16 AECantDetermineFormat
20 AEBadSoundBucket

EXAMPLES

The following example copies the data from sound bucket *sb* to existing file **/myhome/a_dir/a_file**, starting at offset 1668. The file format and audio attributes of the existing file are to be used, so *file_fmt* is set to **AFFUnknown**, NULL is passed for audio attributes, and the attribute mask is set to 0.

The mode is set to **AWMOverwrite** so that data that precedes or follows the overwrite region will not be affected. If necessary, the length of the file will be increased to accommodate the new data.

```
Audio *audio; /* audio connection */
SBucket *sb; /* sound bucket */
AFileFormat file_fmt; /* file format */
AudioAttrMask a_mask; /* audio attributes mask */
ATime startoff; /* start offset */
AWriteMode mode; /* write mode */
long status; /* error status */
      .
      .
      .
/* save sound bucket data */
staticchara_name[] = {"/myhome/a_dir/a_file"};
file_fmt = AFFUnknown;
a_mask = 0;
startoff.type = ATTSamples;
startoff.u.samples = 1668;
mode = AWMOverwrite; /* overwrite without truncate */
ASaveSBucket(audio, sb, a_name, file_fmt, a_mask, NULL, &startoff,
mode, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

ASaveSBucket() was developed by HP.

SEE ALSO

ACreateSBucket(3X), ADestroySBucket(3X), AGetSBucketData(3X), ALoadAFile(3X), APlaySBucket(3X), APutSBucketData(3X), ARecordAData(3X).

Using the Audio Application Program Interface.

NAME

ASelectInput - request report of specified audio events

SYNOPSIS

```
void ASelectInput(
    Audio *audio,
    ATransID xid,
    AEventMask event_mask,
    long *status_return
);
```

DESCRIPTION

`ASelectInput()` requests the report of the audio events specified by the event mask.

audio is the `Audio` structure associated with this connection.

xid specifies the ID of the transaction whose events are of interest.

event_mask specifies the events for which a report is requested. Each bit in the mask corresponds to one type of audio event. The mask is the bitwise inclusive OR of the masks for the individual event types.

status_return receives the returned status of the operation unless it is set to NULL.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return*:

0 AENoError
2 AEBadAudio
3 AEBadValue

EXAMPLES

The following example sets the event mask to request record monitor and transaction pause events, and sets up *status_return* to receive an error status.

```
Audio *audio;          /* audio connection */
TransID xid;           /* transaction ID */
AEventMask emask;      /* event mask */
long status;           /* error status */
      .
      .
      .
/* request input event reports */
emask = (AETRecordMonitor | AETransPaused)
ASelectInput(audio, xid, emask, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

`ASelectInput()` was developed by HP.

SEE ALSO

ACheckEvent(3X), ACheckMaskEvent(3X), AEventsQueued(3X), AMaskEvent(3X), ANextEvent(3X), APeekEvent(3X), APutBackEvent(3X), AQlength(3X).

Using the Audio Application Program Interface.

NAME

AServerVendor - get vendor name of audio server for this connection

SYNOPSIS

```
#include <audio/Alib.h>

char* AServerVendor (Audio *audio);
```

DESCRIPTION

AServerVendor() returns a pointer to the name of the vendor of the audio server for the connection specified by *audio*.

audio specifies the **Audio** structure associated with this connection.

RETURN VALUE

Upon successful completion, **AServerVendor()** returns a pointer to the name of the vendor of the the audio server associated with this connection.

ERRORS

AServerVendor() does not return an error status.

EXAMPLES

The following example returns a pointer to the name of the vendor of the audio server for the connection specified by *audio*.

```
char* *vendor_name;          /* server vendor name  */
Audio *audio;                /* audio connection */
      .
      .
      .
/* get server vendor name */
vendor_name = AServerVendor(audio);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

AServerVendor() was developed by HP.

SEE ALSO

AProtocolRevision(3X), AProtocolVersion(3X), AVendorRelease(3X).

Using the Audio Application Program Interface.

NAME

ASetChannelGain - set transaction channel gain

SYNOPSIS

```
#include <audio/Alib.h>

void ASetSystemChannelGain(
    Audio *audio,
    ATransID xid,
    AChType channel,
    AGainDB gain,
    long *status_return
);
```

DESCRIPTION

ASetChannelGain() sets the transaction gain to the value in *gain*.

audio	**Audio** structure associated with this connection.
xid	Transaction ID.
channel	Type of channel: **ACTMono**, **ACTLeft**, or **ACTRight**.
gain	Specifies the volume: **AUnityGain**, **AZeroGain**, or a number of decibels.
status_return	Receives the returned status of the operation unless it is set to NULL.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return*.

0	AENoError
2	AEBadAudio
3	AEBadValue

EXAMPLES

The following example sets the transaction right channel gain to –6.

```
Audio *audio;          /* audio connection */
AChType *chtype;       /* type of channel */
AGainDB chgain;        /* gain specification*/
long status;           /* error status */
      .
      .
      .
/* set xid right channel gain to -6 */
chtype = ACTRight
chgain = -6;
ASetChannelGain(audio, xid, chtype, chgain, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

ASetChannelGain() was developed by HP.

SEE ALSO

AGetChannelGain(3X) AGetGain(3X), AGetSystemChannelGain(3X), AGMGainRestricted(3X), AInputChannels(3X), AInputSources(3X), AMaxInputGain(3X), AMaxOutputGain(3X), AMinInputGain(3X), AMinOutputGain(3X), AOutputChannels(3X), AOutputDestinations(3X), ASetGain(3X), ASetSystemChannelGain(3X), ASimplePlayer(3X), ASimpleRecorder(3X).

Using the Audio Application Program Interface.

NAME

ASetCloseDownMode - set close-down mode to destroy or complete transactions on specified connection

SYNOPSIS

```
#include <audio/Alib.h>

void ASetCloseDownMode (
    Audio *audio,
    ACloseDownMode close_mode,
    long *status_return
);
```

DESCRIPTION

ASetCloseDownMode() sets the close-down mode to keep or destroy active and pending transactions on the connection associated with *audio*.

audio specifies the **Audio** structure associated with this connection.

close_mode specifies one of two modes: **ADestroyAll** causes all active and pending transactions for this connection to be stopped and destroyed when the connection is closed; associated storage is freed immediately; **AKeepTransactions** prevents transactions for this connection from being destroyed and allows them to complete before the close.

status_return receives the returned status of the operation unless it is set to NULL by the application.

ERRORS

If *status_return* is not set to NULL by the application, one of the following is returned in *status_return*:

0 AENoError
2 AEBadAudio
3 AEBadValue

EXAMPLES

The following example sets the close-down mode to allow transactions to complete, and sets up *status* to receive an error status.

```
Audio * audio;  /* audio connection */
ACloseDownMode close_mode;  /* close down mode */
long status;  /* error status */
      .
      .
      .
/* set close-down mode */
close_mode = AKeepTransactions;
ASetCloseDownMode(audio, close_mode, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

ASetCloseDownMode() was developed by HP.

SEE ALSO

Using the Audio Application Program Interface.

NAME

ASetErrorHandler - replace default error handler with specified handler

SYNOPSIS

```
#include <audio/Alib.h>

AErrorHandler ASetErrorHandler (AErrorHandler handler);
```

DESCRIPTION

ASetErrorHandler() replaces the default error handler with the handler specified in *handler*, and returns a pointer to the handler that was previously in effect. The new error handler should return **AENoError**, if the error should be ignored. If the error should not be ignored and the handler cannot correct it, the handler should return the error code.

handler is the pointer to an application-supplied handler function.

RETURN VALUE

Upon successful completion, **ASetErrorHandler()** returns a pointer to the handler that was previously in effect.

ERRORS

ASetErrorHandler() does not return an error status.

EXAMPLES

The following example replaces the default error handler with a handler named *myhandler*.

```
    long myhandler(
    Audio       *audio,
    AErrorEvent *err_event
)
{
char errorbuff[132];
AGetErrorText(audio, err_event->error_code, errorbuff, 131);
printf ("Error is %s\n", errorbuff);
return (err_event->error_code);
}
      .
      .
      .
AErrorHandler prev_handler;     /* ptr to previous handler */
AErrorHandler myhandler;        /* this data type is a function*/
      .
      .
      .
/* replace default error handler */
prev_handler = ASetErrorHandler(myhandler);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

ASetErrorHandler() was developed by HP.

SEE ALSO

ASetIOErrorHandler(3X).

Using the Audio Application Program Interface.

NAME

ASetGain - set play volume or record gain of specified transaction

SYNOPSIS

```
#include <audio/Alib.h>

void ASetGain(
    Audio *audio,
    ATransID xid,
    AGainDB gain,
    long *status_return
);
```

DESCRIPTION

`ASetGain()` sets the play volume or record gain of the transaction specified in *xid*.

audio specifies the `Audio` structure associated with this connection.

xid specifies the ID of the transaction that was returned by `ACreateSBucket()` or `ALoadAFile()`.

gain specifies the new values for the play volume or record gain.

status_return receives the returned status of the operation unless it is set to NULL.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return*.

0	AENoError
2	AEBadAudio
15	AEBadTransactionID

EXAMPLES

The following example sets the gain for the *xid* transaction to be `AUnityGain` (unchanged) and sets up *status* to receive an error status:

```
Audio *audio;     /* audio connection */
ATransID xid;     /* transaction ID */
AGainDB gain;     /* gain */
long status;      /* error status */
      .
      .
      .
/* set gain for xid returned from prior call */
gain = AUnityGain;
ASetGain(audio, xid, gain, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

`ASetGain()` was developed by HP.

SEE ALSO

AGetChannelGain(3X) AGetGain(3X), AGetSystemChannelGain(3X), AGMGainRestricted(3X), AInputChannels(3X), AInputSources(3X), AMaxInputGain(3X), AMaxOutputGain(3X), AMinInputGain(3X), AMinOutputGain(3X), AOutputChannels(3X), AOutputDestinations(3X), ASetChannelGain(3X), ASetSystemChannelGain(3X), ASimplePlayer(3X), ASimpleRecorder(3X).

Using the Audio Application Program Interface.

NAME

ASetIOErrorHandler - replace default I/O error handler with specified handler

SYNOPSIS

```
#include <audio/Alib.h>

AIOErrorHandler ASetIOErrorHandler (AIOErrorHandler handler);
```

DESCRIPTION

ASetIOErrorHandler() replaces the default I/O error handler with the handler specified in *handler*, and returns a pointer to the handler that was previously in effect. When the new handler exits via return, the application program exits.

handler is the pointer to an application-supplied I/O handler function.

RETURN VALUE

Upon successful completion, **ASetIOErrorHandler()** returns a pointer to the handler that was previously in effect.

ERRORS

ASetIOErrorHandler() does not return an error status.

EXAMPLES

The following example replaces the default I/O error handler with a handler named *my_io_handler*.

```
long my_io_handler(Audio *audio) {
     printf ("An I/O Error Occurred!\n");
     return 0;    }          .          .          .     AIOErrorHandler
prev_io_handler;   /*  ptr  to  previous  handler  */  AIOErrorHandler
my_io_handler;    /* this data type is a function*/       .          .
     .   /* replace default I/O error handler */ prev_io_handler =
ASetIOErrorHandler(my_io_handler);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

ASetIOErrorHandler() was developed by HP.

SEE ALSO

ASetErrorHandler(3X).

Using the Audio Application Program Interface.

NAME

ASetSystemChannelGain - set system or monitor audio channel gain

SYNOPSIS

```
#include <audio/Alib.h>

void ASetSystemChannelGain(
    Audio *audio,
    ASystemGainType gain_type,
    AChType channel,
    AGainDB gain,
    long *status_return
);
```

DESCRIPTION

ASetSystemChannelGain() sets the system gain to the value in *gain*. If *gain_type* is **ASGTMonitor**, the setting controls how much of the record input signal is fed to the internal speaker or auxiliary output. This ability to monitor the input is particularly useful when the recording input is not from a microphone.

audio **Audio** structure associated with this connection.

gain_type Type of operation: **ASGTPlay**, **ASGTRecord**, or **ASGTMonitor**. If this field is set to **ASGTMonitor**, the channel specification must be **ACTMono**.

channel Type of channel: **ACTMono**, **ACTLeft**, or **ACTRight**. If *gain_type* is **ASGTMonitor**, this field must be **ACTMono**.

gain Specifies the volume: **AUnityGain**, **AZeroGain**, or a number of decibels.

status_return receives the returned status of the operation unless it is set to NULL.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return*.

0 AENoError
2 AEBadAudio
3 AEBadValue

EXAMPLES

The following example sets the gain on the monitor to –6.

```
Audio *audio;                   /* audio connection */
ASystemGainType *sgtype;        /* type of operation */
AChType *chtype;                /* type of channel */
AGainDB chgain;                 /* gain specification*/
long status;                    /* error status */
      .
      .
      .
/* set monitor gain to -6 */
sgtype = ASGTMonitor
chtype = ACTMono
chgain = -6;
ASetSystemChannelGain(audio, sgtype, chtype, chgain, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

ASetSystemChannelGain() was developed by HP.

SEE ALSO

AGetChannelGain(3X), AGetGain(3X), AGetSystemChannelGain(3X), AGMGainRestricted(3X), AInputChannels(3X), AInputSources(3X), AMaxInputGain(3X), AMaxOutputGain(3X), AMinInputGain(3X), AMinOutputGain(3X), AOutputChannels(3X), AOutputDestinations(3X),

ASetChannelGain(3X), ASetGain(3X), ASimplePlayer(3X), ASimpleRecorder(3X).
Using the Audio Application Program Interface.

NAME

ASetSystemPlayGain - set system play volume

SYNOPSIS

```
#include <audio/Alib.h>

void ASetSystemPlayGain(
    Audio *audio,
    AGainDB gain,
    long *status_return
);
```

DESCRIPTION

ASetSystemPlayGain() sets the system play volume to the value in *gain*.

audio specifies the **Audio** structure associated with this connection.

gain specifies in decibels the new value for the play volume.

status_return receives the returned status of the operation unless it is set to NULL by the application.

ERRORS

If *status_return* is not set to NULL by the application, one of the following is returned in *status_return*.

0 AENoError
2 AEBadAudio

EXAMPLES

The following example sets the system play volume to –6. This reduces the play volume by a factor of 4, relative to **AUnityGain**.

```
Audio *audio;      /* audio connection */
AGainDB spvol;     /* sys play vol */
long status;       /* error status */
     .
     .
     .
/* set system play volume */
spvol = -6;
ASetSystemPlayGain(audio, spvol, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

ASetSystemPlayGain() was developed by HP.

SEE ALSO

AGetGain(3X), AGetSystemMonitorGain(3X), AGetSystemPlayGain(3X), AGetSystemRecordGain(3X), AGMGainRestricted(3X), AInputChannels(3X), AInputSources(3X), AMaxInputGain(3X), AMaxOutputGain(3X), AMinInputGain(3X), AMinOutputGain(3X), AOutputChannels(3X), AOutputDestinations(3X), ASetGain(3X), ASetSystemMonitorGain(3X), ASetSystemRecordGain(3X), ASimplePlayer(3X), ASimpleRecorder(3X).

Using the Audio Application Program Interface.

NAME

ASetSystemRecordGain - set system record gain

SYNOPSIS

```
#include <audio/Alib.h>

void ASetSystemRecordGain(
    Audio *audio,
    AGainDB gain,
    long *status_return
);
```

DESCRIPTION

ASetSystemRecordGain() sets the system record gain to the value in *gain*.

audio specifies the **Audio** structure associated with this connection.

gain specifies the new value for the record gain.

status_return receives the returned status of the operation unless it is set to NULL.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return*.

0 AENoError
2 AEBadAudio

EXAMPLES

The following example sets the system record gain to –6.

```
Audio *audio;     /* audio connection */
AGainDB srgain;   /* sys record gain */
long status;      /* error status */
      .
      .
      .
/* set system record gain */
srgain = -6;
ASetSystemRecordGain(audio, srgain, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

ASetSystemRecordGain() was developed by HP.

SEE ALSO

AGetGain(3X) AGetSystemMonitorGain(3X), AGetSystemPlayGain(3X), AGetSystemRecordGain(3X), AGMGainRestricted(3X), AInputChannels(3X), AInputSources(3X), AMaxInputGain(3X), AMaxOutputGain(3X), AMinInputGain(3X), AMinOutputGain(3X), AOutputChannels(3X), AOutputDestinations(3X), ASetGain(3X), ASetSystemMonitorGain(3X), ASetSystemPlayGain(3X), ASimplePlayer(3X), ASimpleRecorder(3X).

Using the Audio Application Program Interface.

NAME

ASetupConversion - perform setup required for stream data conversion

SYNOPSIS

```
#include <audio/Alib.h>

AConvertParams*
ASetupConversion (
    Audio               * audio,
    AudioAttributes     * src_attributes,
    AByteOrder          * src_byte_order,
    AudioAttributes     * dest_attributes,
    AByteOrder          * dest_byte_order,
    long                * status_return );
```

DESCRIPTION

ASetupConversion( ) performs initialization for stream data conversion. The user specifies the source stream attributes and byte order and the desired destination stream attributes. **ASetupConversion** returns a pointer to an *AConvertParams* structure, which will be used by **AConvertBuffer** to perform the stream conversion.

audio	specifies the **Audio** structure associated with this connection.
src_attributes	specifies the attributes of the source stream.
src_byte_order	specifies the byte ordering of the source stream.
dest_attributes	specifies the attributes of the destination stream.
dest_byte_order	specifies the byte ordering of the destination stream.
status_return	receives the returned status of the operation, unless it is set to NULL.

RETURN VALUE

Upon successful completion, **ASetupConversion( )** returns a pointer to the conversion parameter structure *AConvertParams*. To free the space allocated for this structure, use **AEndConversion .**

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return:*

0 AENoError
17 AEOutOfMemory

EXAMPLE

For an example, see `/usr/audio/examples/splayer.c`

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

`ASetupConversion( )` was developed by HP.

SEE ALSO

AConvertBuffer(3X), AChooseSourceAttributes(3X), AChoosePlayAttributes(3X), AEndConversion(3X)

Using the Audio Application Program Interface.

NAME

ASimplePlayer - return gain matrix of basic play device

SYNOPSIS

```
#include <audio/Alib.h>

AGainMatrix ASimplePlayer(Audio *audio);
```

DESCRIPTION

ASimplePlayer() returns the gain matrix of the basic audio play device supported by the audio controller associated with the connection specified by *audio*.

audio specifies the **Audio** structure associated with this connection.

RETURN VALUE

Upon succesful completion, **ASimplePlayer()** returns the gain matrix of the basic audio play device supported by the audio controller associated with the connection specified by *audio*.

ERRORS

ASimplePlayer() does not return an error status.

EXAMPLES

The following example gets the gain matrix of the basic audio play device supported by the audio controller associated with the connection specified by *audio*.

```
AGainMatrix *spmatrix;    /* simple play gain matrix */
Audio *audio;             /* audio connection */
      .
      .
      .
/* get the simple record gain matrix */
spmatrix = ASimplePlayer(audio);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

ASimplePlayer() was developed by HP.

SEE ALSO

AGetChannelGain(3X) AGetGain(3X), AGetSystemChannelGain(3X), AGMGainRestricted(3X), AInputChannels(3X), AInputSources(3X), AMaxInputGain(3X), AMaxOutputGain(3X), AMinInputGain(3X), AMinOutputGain(3X), AOutputChannels(3X), AOutputDestinations(3X), ASetChannelGain(3X), ASetGain(3X), ASetSystemChannelGain(3X), ASimpleRecorder(3X).

Using the Audio Application Program Interface.

NAME

ASimpleRecorder - return gain matrix of basic recording device

SYNOPSIS

```
#include <audio/Alib.h>

AGainMatrix ASimpleRecorder(Audio *audio);
```

DESCRIPTION

ASimpleRecorder() returns the gain matrix of the basic audio recording device supported by the audio controller associated with the connection specified by *audio*.

audio specifies the **Audio** structure associated with this connection.

RETURN VALUE

Upon succesful completion, **ASimpleRecorder()** returns the gain matrix of the basic audio recording device supported by the audio controller associated with the connection specified by *audio*.

ERRORS

ASimpleRecorder() does not return an error status.

EXAMPLES

The following example gets the gain matrix of the basic audio recording device supported by the audio controller associated with the connection specified by *audio*.

```
AGainMatrix *srmatrix;      /* simple record gain matrix */
Audio *audio;               /* audio connection */
        .
        .
        .
/* get the simple record gain matrix */
srmatrix = ASimpleRecorder(audio);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

ASimpleRecorder() was developed by HP.

SEE ALSO

AGetChannelGain(3X) AGetGain(3X), AGetSystemChannelGain(3X), AGMGainRestricted(3X), AInputChannels(3X), AInputSources(3X), AMaxInputGain(3X), AMaxOutputGain(3X), AMinInputGain(3X), AMinOutputGain(3X), AOutputChannels(3X), AOutputDestinations(3X), ASetChannelGain(3X), ASetGain(3X), ASetSystemChannelGain(3X), ASimplePlayer(3X),

Using the Audio Application Program Interface.

NAME

asinh, acosh, atanh - inverse hyperbolic functions

SYNOPSIS

```
#include <math.h>

double asinh(double x);

double acosh(double x);

double atanh(double x);
```

DESCRIPTION

`asinh()`, `acosh()`, and `atanh()` return respectively the designated inverse hyberbolic sine, cosine, and tangent of their argument.

When x is ±INFINITY, `asinh()` returns ±INFINITY respectively.

When x is +INFINITY, `acosh()` returns +INFINITY.

ERRORS

/lib/libm.a

`asinh()`, `acosh()`, and `atanh()` return NaN and set `errno` to EDOM when x is NaN. In addition, a message indicating DOMAIN error is printed on the standard error output.

`acosh()` also returns NaN and sets `errno` to EDOM if $x < 1.0$.

`atanh()` also returns NaN and sets `errno` to EDOM if $|x| \geq 1.0$.

These error-handling procedures can be changed with the function `matherr()` (see *matherr*(3M)).

/lib/libM.a

No error messages are printed on the standard error output.

`asinh()`, `acosh()`, and `atanh()` return NaN and set `errno` to EDOM when x is NaN.

`acosh()` also returns NaN and sets `errno` to EDOM if $x < 1.0$.

`atanh()` also returns NaN and sets `errno to` EDOM if $|x| \geq 1.0$.

These error-handling procedures can be changed by using the `_matherr()` function (see *matherr*(3M)). Note that `_matherr()` is provided in order to assist in migrating programs from `libm.a` to `libM.a` and is *not* a part of XPG3, ANSI C, or POSIX.

DEPENDENCIES

Series 300/400

`asinh()`, `acosh()`, and `atanh()` are not supported on Series 300/400 systems.

Series 700/800

`asinh()`, `acosh()`, and `atanh()` are not specified by any standard. They are provided in the PA1.1 versions of the math library only. The `+DA1.1` linker option (default on Series 700 systems) links in a PA1.1 version automatically. A PA1.1 library can also be linked in explicitly. For more information, see the *HP-UX Floating-Point Guide*.

SEE ALSO

exp(3M), matherr(3M).

NAME

ASoundBitOrder - get bit order used for one-bit-per-sample data

SYNOPSIS

```
#include <audio/Alib.h>

ABitOrder ASoundBitOrder (Audio *audio);
```

DESCRIPTION

ASoundBitOrder() returns the bit order that will be used if data is one bit per sample.

audio specifies the **Audio** structure associated with this connection.

RETURN VALUE

Upon successful completion, **ASoundBitOrder()** returns the bit order that will be used if data is one bit per sample.

ERRORS

ASoundBitOrder() does not return an error status.

EXAMPLES

The following example returns the bit order that will be used if the data is one bit per sample.

```
ABitOrder bit_order;     /* bit order */
Audio *audio;            /* audio connection */
        .
        .
        .
/* get bit order */
bit_order = ASoundBitOrder(audio);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

ASoundBitOrder() was developed by HP.

SEE ALSO

ASoundByteOrder(3X).

Using the Audio Application Program Interface.

NAME

ASoundByteOrder - get byte order of audio data accepted by audio controller for this connection

SYNOPSIS

```
#include <audio/Alib.h>

AByteOrder ASoundByteOrder (Audio *audio );
```

DESCRIPTION

ASoundByteOrder() returns the byte order of audio data accepted by the audio controller associated with the *audio* connection.

audio specifies the **Audio** structure associated with this connection.

RETURN VALUE

Upon successful completion, **ASoundByteOrder()** returns the byte order of audio data accepted by the audio controller associated with this connection.

ERRORS

ASoundByteOrder() does not return an error status.

EXAMPLES

The following example returns the byte order accepted by the audio controller associated with the connection specified by *audio*.

```
AByteOrder byte_order;    /* acceptable byte order  */
Audio *audio;             /* audio connection */
      .
      .
      .
/* get acceptable byte order */
byte_order = ASoundByteOrder(audio);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

ASoundByteOrder() was developed by HP.

SEE ALSO

ASoundBitOrder(3X).

Using the Audio Application Program Interface.

NAME

assert() - verify program assertion

SYNOPSIS

```
#include <assert.h>

int assert(int expression);
```

DESCRIPTION

This macro is useful for putting diagnostics into programs. When it is executed, if *expression* is false (zero), `assert()` prints:

`Assertion failed:` *expression*`, file` *xyz*`, line` *nnn*

on the standard error output and aborts. In the error message, *xyz* is the name of the source file and *nnn* the source line number of the `assert()` statement.

Compiling with the preprocessor option `-DNDEBUG` (see *cpp*(1)), or with the preprocessor control statement `#define NDEBUG` ahead of the `#include <assert.h>` statement, stops assertions from being compiled into the program.

WARNINGS

The expression argument used by `assert()` in compatibility mode cannot contain string literals or double quotes without escapes.

SEE ALSO

cpp(1), abort(3C).

STANDARDS CONFORMANCE

`assert()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

AStopAudio - stop specified audio transaction

SYNOPSIS

```
#include <audio/Alib.h>

void AStopAudio (
    Audio .*audio,
    ATransID xid,
    AStopMode mode,
    ATransStatus *trans_status_return,
    long *status_return
);
```

DESCRIPTION

AStopAudio() stops the transaction specified in *xid*. A stopped transaction cannot be resumed.

audio specifies the **Audio** structure associated with this connection.

xid specifies the transaction ID that was returned by **ACreateSBucket()** or **ALoadAFile()**.

mode specifies the stop mode: **ASMLinkedTrans**, **ASMThisTrans**, or **ASMEndLoop**.

To stop the current and subsequent transactions in a linked list, use **ASMLinkedTrans** and specify the first transaction in the linked list as *xid*. The current and subsequent transactions in the linked list cannot be resumed after this stop.

To stop only the current transaction in a linked list, use **ASMThisTrans**. The specified transaction stops immediately, even if it is in the middle of a loop, and the remaining transactions in the linked list continue immediately.

To stop a looping transaction, use **ASMEndLoop**. The specified transaction stops at the end of the current loop. If the loop transaction is in a linked list, the remaining transactions continue.

If *xid* is not in a linked list, **ASMLinkedTrans()** has the same effect as **ASMThisTrans**.

trans_status_return receives the returned status value. Setting this argument to NULL prevents the data from being collected and returned, which may enhance performance.

status_return receives the returned status of the operation unless it is set to NULL.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return*

0	AENoError
2	AEBadAudio
3	AEBadValue
15	AEBadTransactionID

EXAMPLES

The following example stops the transaction identified by *xid*, sets *mode* to stop the specified transaction, passes NULL for *trans_stat*, and sets up *status* to receive an error status.

```
Audio *audio;        /* audio connection */
ATransID xid;        /* transaction ID */
AStopMode smode;     /* stop mode */
long status;         /* error status */
      .
      .
      .
/* stop transaction - xid returned from prior call */
smode = ASMThisTrans;
AStopAudio(audio, xid, smode, NULL, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

`AStopAudio()` was developed by HP.

SEE ALSO

APauseAudio(3X), AResumeAudio(3X).

Using the Audio Application Program Interface.

NAME

AtAddCallback - add callback procedure for audio toolkit

SYNOPSIS

```
#include <audio/Alib.h>

void AtAddCallback(
    Widget widget,
    char *event,
    XtCallbackProc proc,
    XtPointer client_data
);
```

DESCRIPTION

`AtAddCallback()` adds a callback for the audio toolkit to use. When the toolkit receives an event, it checks to see if a callback procedure has been entered for the event. If a procedure has been entered, the toolkit calls it.

Note that a callback for **AuNdataAvailable** must be added for a record stream widget operation to work, and a callback for **AuNdataNeeded** must be added for a play stream widget operation to work.

widget Name of the widget

event Event type of the callback. Acceptable values are:

AuNstarted	**AuNerror**	**AuNdataNeeded**
AuNstopped	**AuNmonitor**	**AuNloopStarted**
AuNpaused	**AuNpreempted**	**AuNloopStopped**
AuNresumed	**AuNdataAvailable**	**AuNbrokenConnection**
AuNcompleted		

Note: **AuNBrokenConnection** indicates that the connection to the server has been severed. A callback for this event should arrange for the application to exit gracefully.

proc Name of the callback procedure.

client_data Data that the client wants to use; can be NULL.

ERRORS

`AtAddCallback()` does not return an error status.

DEPENDENCIES

This function belongs to the Audio Application Program Interface toolkit. The audio server must run on a system equipped with audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual provided with your system.

AUTHOR

`AtAddCallback()` was developed by HP.

SEE ALSO

AtInitialize(), AuCreatePlay(), AuCreateRecord(), AuInvokePlay(), AuInvokeRecord().

Using the Audio Application Program Interface.

NAME

AtInitialize - add audio event handler for this connection

SYNOPSIS

```
#include <audio/Alib.h>

void AtInitialize(Audio *a_connection);
```

DESCRIPTION

`AtInitialize()` adds the audio event handler for the specified server connection. The graphical toolkit must be initialized and `AOpenAudio()` must be called before calling `AtInitialize()` because `AOpenAudio()` returns the pointer to the `Audio` structure for the connection, and `AtInitialize()` calls `XtAddInput()`.

The audio toolkit cannot be used without the graphical toolkit.

a_connection specifies the `Audio` structure associated with this connection.

ERRORS

`AtInitialize()` does not return an error status.

DEPENDENCIES

This function belongs to the Audio Application Program Interface toolkit. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

`AtInitialize()` was developed by HP.

SEE ALSO

AtAddCallback(), AtRemoveCallback(), AuCreatePlay(), AuCreateRecord(), AuInvokePlay(), AtPlayWidget(), AtRecordWidget(), AuInvokeRecord().

Using the Audio Application Program Interface.

NAME

AtRemoveCallback - set callback to NULL

SYNOPSIS

```
#include <audio/Alib.h>

void AtRemoveCallback(Widget widget, char *event);
```

DESCRIPTION

`AtRemoveCallback()` sets to NULL a callback that was added by `AtAddCallback()` . The *client_data* field is also set to NULL.

widget Name of the widget.

event Event type of the callback. Acceptable values are:

`AuNstarted`	`AuNerror`	`AuNdataNeeded`
`AuNstopped`	`AuNmonitor`	`AuNloopStarted`
`AuNpaused`	`AuNpreempted`	`AuNloopStopped`
`AuNresumed`	`AuNdataAvailable`	`AuNbrokenConnection`
`AuNcompleted`		

ERRORS

`AtRemoveCallback()` does not return an error status.

DEPENDENCIES

This function belongs to the Audio Application Program Interface toolkit. The audio server must run on a system equipped with audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual provided with your system.

AUTHOR

`AtRemoveCallback()` was developed by HP.

SEE ALSO

AtInitialize(), AuCreatePlay(), AuCreateRecord(), AuInvokePlay(), AuInvokeRecord(), AuPlayWidget(), AuRecordWidget().

Using the Audio Application Program Interface.

NAME

AuCreatePlay - create an audio play widget

SYNOPSIS

```
#include <audio/Play.h>

extern Widget AuCreatePlay(
    Widget parent,
    String *name,
    ArgList *arglist,
    Cardinal argcount
);
```

DESCRIPTION

`AuCreatePlay()` creates a play widget.

If you use the streams facility, the toolkit creates a file descriptor in *connectFd* during `AuInvokePlay()`. After calling `AuInvokePlay()`, retrieve the file descriptor by calling `XtSetArg(args[0], AuNconnectFd, &stream_fd);` and then call `XtGetValue(playWidget,args,1);`. Then, use the `select()`, `read()`, and `write()` system calls.

Call `AStopAudio()` to stop the transaction. A callback routine for `AuNStopped` can `close()` the file descriptor.

Note that for a play streams operation to work, a callback routine for `AuNdataNeeded` must be added using `AtAddCallback()`.

To enable an application to use a widget after it is created, bind the widget library with the application as follows:

`ld` *my_file*`.o... -lAt -lAlib`

Arguments

parent	Name of the parent widget
name	Name for this widget
arglist	The argument list for the widget
argcount	The number of arguments in *arglist*.

arglist can contain the following:

gain — Volume, in percent of total gain. Acceptable values are from `0` to `100`. Default is system dependent.

fileFormat — Audio file format. Acceptable values are:

`AuFAlaw` `AuFLinear8Offset` `AuFSun`
`AuFMulaw` `AuFLinear16` `AuFUnknown`
`AuFLinear8` `AuFRiff`

Default is `AuFUnknown`.

dataFormat — Audio data format. Acceptable values are:

`AuDMulaw` `AuDLinear16` `AuDLinear8Offset`
`AuDAlaw` `AuDLinear8` `AuDUnknown`

Default is `AuDUnknown`.

durationType — Duration units. Acceptable values are:

`AuSamples` `AuFullLength` `AuMilliseconds`

Default is `AuFullLength`.

duration — Number of units to play. Acceptable values are `-1` to `MAX_INT`. Default is `-0` (-1, play until notified).

fileName — Name of the file to play (must be set prior to invocation of the play widget). There is no default value.

startTimeType Type of start time unit. Acceptable values are **AuSamples** and **AuMilliseconds**. Default is **AuMilliseconds**.

startTime Number of units into the file to begin recording. Acceptable values are **0** to **MAX_INT**. Default value is **0** (BOF).

pause Start in paused state. Acceptable values are **ON** or **OFF**. Default is **OFF**.

stop Stop mode. Acceptable values are:

AuStopLinkTrans **AuStopEndLoopTrans**
AuStopThisTrans **AuStopNone**

Default is **AuStopNone**.

audioConnection Pointer to **Audio** structure for this connection, returned by **AOpenAudio()**. Specifying a valid pointer for this argument is mandatory; the default value is NULL, which causes the program to fail.

streamOrFile Source of audio data. Acceptable values are **AuStream** or **AuFile**. Default is **AuFile**.

SStream Pointer to **SStream** structure for this AuStream-type widget. Specifying a valid pointer for this argument is mandatory; the default value is NULL, which causes the program to fail.

connectFd File descriptor of the non-blocking connection made for the stream by the toolkit; created during **AuInvokeRecord()**.

reserved

speaker Speaker choice. Acceptable values are **AuInternal** or **AuExternal**. Default is **AuInternal**.

link name of another play widget; when *link* is finished playing, the current widget starts immediately and automatically. Default is NULL.

The link feature enables two or more play widgets to be linked into a continuous play operation. Follow these steps to link two or more widets:

1. Create widget A with *pause* **ON** and with NULL specified in *link*.
2. Create widget B with *pause* **ON** and with A's name specified in *link*.
3. Repeat step 2 for as many widgets as you want in the chain (creating C with B's name in *link*, and so on).
4. Invoke widget A.

loopCount Number of times to play this widget. Acceptable values are **-1** to **MAX_INT**. Default is **0**. Note that a value of **-1** specifies an infinite loop.

priority Priority level of play request. Acceptable values are:

AuUrgent **AuNormal** **AuHigh** **AuLow**

Default is **AuNormal**.

channels Number of channels. Acceptable values are **1** or **2**. Default is **1**.

samplingRate Number of cycles per second. Most Series 700 systems support the following values:

5512 **11025** **22050** **44100**
8000 **16000** **32000** **48000**

Default is **8000**.

To double-check the values that your system supports, use **ASamplingRates()**.

Values between 0.995 and 1.0125 times any of the supported values are handled at the supported rate. Rates outside these tolerances are converted by sound

bucket transactions to the nearest supported rate but cause streams transactions to fail and return **AEBadSamplingRate**.

leftChannel Gain, in %. Acceptable values are `1` through `100`. Default is system dependent.

rightChannel Gain, in %. Acceptable values are `1` through `100`. Default is system dependent.

RETURN VALUE

Upon successful completion, **AuCreatePlay()** returns the widget ID.

ERRORS

AuCreatePlay() does not return an error status.

DEPENDENCIES

This function belongs to the Audio Application Program Interface widget library. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

AuCreatePlay() was developed by HP.

SEE ALSO

AtAddCallback(3X), AtInitialize(3X), AtRemoveCallback(3X), AuCreateRecord(3X), AuInvokePlay(3X), AuInvokeRecord(3X), AuPlayWidget(3X), AuRecordWidget(3X), AuSaveFile(3X).

Using the Audio Application Program Interface.

NAME

AuCreateRecord - create an audio record widget

SYNOPSIS

```
#include <audio/Record.h>

extern Widget AuCreateRecord(
    Widget parent,
    String *name,
    ArgList *arglist,
    Cardinal argcount
);
```

DESCRIPTION

`AuCreateRecord()` creates a record widget.

If the record operation is file-based, the `AuNdataAvailable` event is returned when all of the data has been saved in the file. To use this information, add a callback routine for `AuNdataAvailable` using `AtAddCallback()`.

If you use the streams facility, the toolkit creates a file descriptor in *connectFd* during `AuInvokeRecord()`. After calling `AuInvokeRecord()`, retrieve the file descriptor by calling

```
XtSetArg(args[0], AuNconnectFd, &stream_fd);
```

and then call

```
XtGetValue(recordWidget,args,1);.
```

Then, use the `select()`, `read()`, and `write()` system calls.

After calling `AStopAudio()` to stop the transaction, the application program must retrieve all the data in the buffer and `close()` the file descriptor. A callback routine for `AuNStopped` can include all of these operations.

Note that for a record streams operation to work, a callback routine for `AuNdataAvailable` must be added using `AtAddCallback()`.

To enable an application to use a widget after it is created, bind the widget library with the application as follows:

`ld` *my_file*`.o... -lAt -lAlib`

Arguments

parent	Name of the parent widget
name	Name for this widget
arglist	The argument list for the widget
argcount	The number of arguments in *arglist*.

arglist can contain the following:

gain — Volume, in per cent of total gain. Acceptable values are from `0` through `100`. Default is system-dependent.

fileFormat — Audio file format. Acceptable values are:

`AuFMulaw`	`AuFSun`	`AuFLinear8Offset`
`AuFAlaw`	`AuFLinear8`	`AuFUnknown`
`AuFLinear16`	`AuFRiff`	

Default is `AuFUnknown`.

dataFormat — Audio data format. Acceptable values are:

`AuDMulaw`	`AuDLinear16`	`AuDLinear8Offset`
`AuDAlaw`	`AuDLinear8`	`AuDUnknown`

Default is `AuDUnknown`.

durationType Duration units. Acceptable values are:

AuSamples **AuMilliseconds** **AuFullLength**

Default is **AuFullLength**.

duration Number of units to record. Acceptable values are **-1** to **MAX_INT**. Default is **-0** (-1, record until notified).

fileName Name of the file to receive the data (must be set prior to invocation of the record widget). There is no default value.

startTimeType Type of start time unit. Acceptable values are **AuSamples** and **AuMilliseconds**. Default is **AuMilliseconds**.

startTime Number of units into the file to begin writing (when the recorded file is saved). Acceptable values are **0** to **MAX_INT**. Default value is **0** (BOF).

pause Start in paused state. Acceptable values are **ON** or **OFF**. Default is **OFF**.

stop Stop mode. Acceptable values are:

AuStopLinkTrans **AuStopEndLoopTrans**
AuStopThisTrans **AuStopNone**

Default is **AuStopNone**.

audioConnection Pointer to **Audio** structure for this connection, returned by **AOpenAUdio()**. Specifying a valid pointer for this argument is mandatory; the default value is NULL, which causes the program to fail.

streamOrFile Source of audio data. Acceptable values are **AuStream** or **AuFile**. Default is **AuFile**.

SStream Pointer to **SStream** structure for this AuStream-type widget. Specifying a valid pointer for this argument is mandatory; the default value is NULL, which causes the program to fail.

connectFd File descriptor of the non-blocking connection made for the stream by the toolkit; created during **AuInvokeRecord()**.

reserved

writeMode Mode for saving data. Acceptable values are:

AuOverWrite **AuInsert**
AuTruncAppend

Default is **AuOverWrite**.

channels Number of channels. Acceptable values are **1** or **2**. Default is **1**.

samplingRate Number of cycles per second. Most Series 700 systems support the following values:

5512 11025 22050 44100
8000 16000 32000 48000

Default is **8000**.

To double check on the values that your system supports, use **ASamplingRates()**.

Values between 0.995 and 1.0125 times any of the supported values are handled at the supported rate. Rates outside these tolerances are converted by sound bucket transactions to the nearest supported rate, but cause streams transactions to fail and return **AEBadSamplingRate**.

leftChannel Gain, in %. Acceptable values are **1** through **100**. Default is system-dependent.

rightChannel Gain, in %. Acceptable values are **1** through **100**. Default is system-dependent.

audioIn Line in or monaural microphone. Acceptable values are **AuMicrophone** or **AuLineIn**. Default is **AuMicrophone**.

RETURN VALUE

Upon successful completion, `AuCreateRecord()` returns the widget ID.

ERRORS

`AuCreateRecord()` does not return an error status.

DEPENDENCIES

This function belongs to the Audio Application Program Interface widget library. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

`AuCreateRecord()` was developed by HP.

SEE ALSO

AtAddCallback(), AtInitialize(), AtRemoveCallback(), AuCreatePlay(), AuInvokePlay(), AuInvokeRecord() AuPlayWidget, AuRecordWidget, AuSaveFile().

Using the Audio Application Program Interface.

NAME

AuInvokePlay - initiate a widget play operation

SYNOPSIS

```
#include <audio/Play.h>

extern void AuInvokePlay(Widget widget);
```

DESCRIPTION

AuInvokePlay() initiates a widget play operation. A play widget must be created before it can be invoked. If the application calls **AuInvokePlay()** more than once, set up callbacks for **AuNstopped** and **AuNcompleted**. Use these notifications to avoid overlapping with a transaction that is still active from a prior **AuInvokePlay()**.

widget the name of a play widget.

ERRORS

AuInvokePlay() does not return an error status.

DEPENDENCIES

This function belongs to the Audio Application Program Interface widget library. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

AuInvokePlay() was developed by HP.

SEE ALSO

AtAddCallback(), AtInitialize(), AtRemoveCallback, AuCreatePlay(), AuCreateRecord(), AuInvokeRecord(), AuPlayWidget, AuRecordWidget, AuSaveFile().

Using the Audio Application Program Interface.

NAME

AuInvokeRecord - initiate a widget record operation

SYNOPSIS

```
#include <audio/Record.h>

extern void AuInvokeRecord(Widget widget);
```

DESCRIPTION

AuInvokeRecord() initiates a widget record operation. A record widget must be created before it can be invoked. If the application calls this function more than once, set up callbacks for **AuNstopped** and **AuNcompleted**. Use these notifications to avoid overlapping with a transaction that is still active from a prior **AuInvokeRecord()**.

widget the name of a record widget.

ERRORS

AuInvokeRecord() does not return an error status.

DEPENDENCIES

This function belongs to the Audio Application Program Interface widget library. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

AuInvokeRecord() was developed by HP.

SEE ALSO

AtAddCallback(), AtInitialize(), AtRemoveCallback(), AuCreatePlay(), AuCreateRecord(), AuInvokePlay(), AuPlayWidget, AuRecordWidget, AuSaveFile().

Using the Audio Application Program Interface.

NAME

AUngrabServer - release server from exclusive use by this connection

SYNOPSIS

```
#include <audio/Alib.h>

void AUngrabServer (Audio *audio, long *status_return);
```

DESCRIPTION

AUngrabServer() releases the server from exclusive use by this connection (exclusive use established by **AGrabServer()**).

audio specifies the **Audio** structure associated with this connection.

status_return receives the returned status of the operation unless it is set to NULL.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return*:

0 AENoError
2 AEBadAudio

EXAMPLES

The following example releases the server for general use and sets up *status* to receive an error status.

```
Audio *audio;     /* audio connection */
long status;      /* error status */
      .
      .
      .
/* release server for general use */
AUngrabServer(audio, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

AUngrabServer() was developed by HP.

SEE ALSO

AGrabServer(3X).

Using the Audio Application Program Interface.

NAME

AUpdateDataLength - update a file's header

SYNOPSIS

```
#include <audio/Alib.h>

void
AUpdateDataLength (
    Audio               * audio,
    char                * pathname,
    AFileFormat         file_format,
    long                * status_return );
```

DESCRIPTION

AUpdateDataLength( ) opens the file specified by *pathname* (if the specified *file_format* requires data length or file length information in its header), determines the relevant lengths, writes them to the appropriate fields and closes the file. If the specified file format does not require a header with a data or file length field, **AUpdateDataLength** returns without doing anything.

audio	specifies the **Audio** structure associated with this connection.
pathname	the pathname of the audio file.
file_format	the format of the audio file at *pathname*.
status_return	receives the returned status of the operation, unless it is set to NULL.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return:*

0 AENoError
1 AESystemCall
6 AEBadFileFormat
17 AEOutOfMemory

EXAMPLE

The following example updates header of the Sun/NeXT file at `/myhome/a_dir/a_file`.

```
AFileFormat     file_fmt;  /* file format */
Audio           * audio;  /* audio connection */
long            status;  /* status */
                .
                .
                .
/* update file header of /myhome/a_dir/a_file with relevant lengths */
file_fmt = AFFSun;
char  fname[] = /myhome/a_dir/a_file ;
AUpdateDataLength(audio, fname, file_fmt, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

`AUpdateDataLength( )` was developed by HP.

SEE ALSO

Using the Audio Application Program Interface.

NAME

AuPlayWidget - audio play widget

SYNOPSIS

```
#include <audio/Play.h>
```

DESCRIPTION

The audio play widget is a member of a new `Audio` subclass of the X Core widget class. `AuCreatePlay()` creates a play widget.

If you use the streams facility, the toolkit creates a file descriptor in *connectFd* during `AuInvokePlay()`. After calling `AuInvokePlay()`, retrieve the file descriptor by calling

```
XtSetArg(args[0], AuNconnectFd, &stream_fd);
```

then call

```
XtGetValue(playWidget,args,1);
```

The `select()`, `read()`, and `write()` system calls can then be used in the usual manner.

Call `AStopAudio()` to stop the transaction. A callback routine for `AuNStopped` may `close` the file descriptor.

Note that for a play streams operation to work, a callback routine for `AuNdataNeeded` must be added using `AtAddCallback()`.

To enable an application to use a widget after it is created, bind the widget library with the application as follows:

`ld` *my_file*`.o... -lAt -lAlib`

RESOURCES

gain — Volume, in per cent of total gain. Acceptable values are from `0` to `100`. Default is system-dependent.

fileFormat — Audio file format. Acceptable values are:

`AuFMulaw` `AuFLinear16` `AuFLinear8` `AuFLinear8Offset`
`AuFAlaw` `AuFSun` `AuFRiff` `AuFUnknown`

Default is `AuFUnknown`.

dataFormat — Audio data format. Acceptable values are:

`AuDMulaw` `AuDLinear16` `AuDLinear8Offset`
`AuDAlaw` `AuDLinear8` `AuDUnknown`

Default is `AuDUnknown`.

durationType — Duration units. Acceptable values are:

`AuSamples` `AuMilliseconds` `AuFullLength`

Default is `AuFullLength`.

duration — Number of units to play. Acceptable values are `-1` to `MAX_INT`. Default is `~0` (-1, play until notified).

fileName — Name of the file to play (must be set prior to invocation of the play widget). There is no default value.

startTimeType — Type of start time unit. Acceptable values are `AuSamples` and `AuMilliseconds`. Default is `AuMilliseconds`.

startTime — Number of units into the file to begin recording. Acceptable values are `0` to `MAX_INT`. Default value is `0` (BOF).

pause — Start in paused state. Acceptable values are `ON` or `OFF`. Default is `OFF`.

stop — Stop mode. Acceptable values are:

```
AuStopLinkTrans     AuStopEndLoopTrans
AuStopThisTrans     AuStopNone
```

Default is **AuStopNone**.

audioConnection Pointer to **Audio** structure for this connection, returned by **AOpenAudio()**. Specifying a valid pointer for this argument is mandatory; the default value is NULL, which causes the program to fail.

streamOrFile Source of audio data. Acceptable values are **AuStream** or **AuFile**. Default is **AuFile**.

SStream Pointer to **SStream** structure for this AuStream-type widget. Specifying a valid pointer for this argument is mandatory; the default value is NULL, which causes the program to fail.

connectFd File descriptor of the non-blocking connection made for the stream by the toolkit; created during **AuInvokeRecord()**.

reserved

speaker Speaker choice. Acceptable values are **AuInternal** or **AuExternal**. Default is **AuInternal**.

link name of another play widget; when *link* is finished playing, the current widget starts immediately and automatically. Default is NULL.

The link feature enables two or more play widgets to be linked into a continuous play operation. Follow these steps to link two or more widets:

1. Create widget A with *pause* **ON** and with NULL specified in *link*.
2. Create widget B with *pause* **ON** and with A's name specified in *link*.
3. Repeat step 2 for as many widgets as you want in the chain (creating C with B's name in *link*, and so on).
4. Invoke widget A.

loopCount Number of times to play this widget. Acceptable values are **-1** to **MAX_INT**. Default is **0**. Note that a value of **-1** specifies an infinite loop.

priority Priority level of play request. Acceptable values are:

```
AuUrgent AuNormal
AuHigh AuLow
```

Default is **AuNormal**.

channels Number of channels. Acceptable values are **1** or **2**. Default is **1**.

samplingRate Number of cycles per second. The values that are supported by most Series 700 systems are:

```
5512    11025    22050    44100
8000    16000    32000    48000
```

Default is **8000**.

To double check on the values that your system supports, use **ASamplingRates()**. Values between 0.995 and 1.0125 times any of the supported values are handled at the supported rate. Rates outside these tolerances are converted by sound bucket transactions to the nearest supported rate, but cause streams transactions to fail and return **AEBadSamplingRate**.

leftChannel Gain, in percent. Acceptable values are **1** through **100**. Default is system-dependent.

rightChannel Gain, in percent. Acceptable values are **1** through **100**. Default is system-dependent.

DEPENDENCIES

This widget belongs to the Audio Application Program Interface widget library. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

SEE ALSO

AtAddCallback(), AtInitialize(), AtRemoveCallback(), AuCreateRecord(), AuInvokePlay(), AuInvokeRecord(), AuRecordWidget, AuSaveFile().

Using the Audio Application Program Interface.

NAME

AuRecordWidget - audio record widget

SYNOPSIS

```
#include <audio/Record.h>
```

DESCRIPTION

The audio record widget is a member of the new **Audio** subclass of the X Core widget class.

If the record operation is file-based, the **AuNdataAvailable** event is returned when all of the data has been saved in the file. To use this information, add a callback routine for **AuNdataAvailable** using **AtAddCallback()**.

If you use the streams facility, the toolkit creates a file descriptor in *connectFd* during **AuInvokeRecord()**. After calling **AuInvokeRecord()**, retrieve the file descriptor by calling

```
XtSetArg(args[0], AuNconnectFd, &stream_fd);
```

then call

```
XtGetValue(recordWidget,args,1);.
```

The **select()**, **read()**, and **write()** system calls can then be used in the usual manner.

After calling **AStopAudio()** to stop the transaction, the application program must retrieve all the data in the buffer and **close()** the file descriptor. A callback routine for **AuNStopped** can include all of these operations.

Note that for a record streams operation to work, a callback routine for **AuNdataAvailable** must be added using **AtAddCallback()**.

To enable an application to use a widget after it is created, bind the widget library with the application as follows:

ld *my_file.o...* **-lAt -lAlib**

RESOURCES

gain — Volume, in percent of total gain. Acceptable values are from **0** to **100**. Default is system-dependent.

fileFormat — Audio file format. Acceptable values are:

AuFMulaw **AuFLinear16** **AuFLinear8** **AuFLinear8Offset**
AuFAlaw **AuFSun** **AuFRiff** **AuFUnknown**

Default is **AuFUnknown**.

dataFormat — Audio data format. Acceptable values are:

AuDMulaw **AuDLinear16** **AuDLinear8Offset**
AuDAlaw **AuDLinear8** **AuDUnknown**

Default is **AuDUnknown.**

durationType — Duration units. Acceptable values are:

AuSamples **AuMilliseconds** **AuFullLength**

Default is **AuFullLength**.

duration — Number of units to record. Acceptable values are **-1** to **MAX_INT.** Default is **~0** (-1, record until notified).

fileName — Name of the file to receive the data (must be set prior to invocation of the record widget). There is no default value.

startTimeType — Type of start time unit. Acceptable values are **AuSamples** and **AuMilliseconds**. Default is **AuMilliseconds**.

startTime — Number of units into the file to begin writing (when the recorded file is saved). Acceptable values are **0** to **MAX_INT**. Default value is **0** (BOF).

pause Start in paused state. Acceptable values are **ON** or **OFF**. Default is **OFF**.

stop Stop mode. Acceptable values are:

AuStopLinkTrans AuStopEndLoopTrans
AuStopThisTrans AuStopNone

Default is **AuStopNone**.

audioConnection Pointer to **Audio** structure for this connection, returned by **AOpenAUdio()**. Specifying a valid pointer for this argument is mandatory. The default value is NULL, which causes the program to fail.

streamOrFile Source of audio data. Acceptable values are **AuStream** or **AuFile**. Default is **AuFile**.

SStream Pointer to **SStream** structure for this AuStream-type widget. Specifying a valid pointer for this argument is mandatory. The default value is NULL, which causes the program to fail.

connectFd File descriptor of the non-blocking connection made for the stream by the toolkit; created during **AuInvokeRecord()**.

reserved

writeMode Mode for saving data. Acceptable values are:

AuOverWrite AuTruncAppend AuInsert

Default is **AuOverWrite.**

channels Number of channels. Acceptable values are **1** or **2**. Default is **1**.

samplingRate Number of cycles per second. Hardware-supported values are

5512 11025 22050 44100
8000 16000 32000 48000

Default is **8000**.

Values between 0.995 and 1.0125 times any of the supported values are handled at the supported rate. Rates outside these tolerances are converted by sound bucket transactions to the nearest supported rate, but cause streams transactions to fail and return **AEBadSamplingRate**.

leftChannel Gain, in percent. Acceptable values are **1** through **100**. Default is system-dependent.

rightChannel Gain, in percent. Acceptable values are **1** through **100**. Default is system-dependent.

audioIn Line in or mono microphone. Acceptable values are **AuMicrophone** or **AuLineIn**. Default is **AuMicrophone**.

DEPENDENCIES

This widget belongs to the Audio Application Program Interface widget library. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

SEE ALSO

AtAddCallback(), AtInitialize(), AtRemoveCallback(), AuCreatePlay(), AuInvokePlay(), AuInvokeRecord(), AuPlayWidget, AuSaveFile().

Using the Audio Application Program Interface.

NAME

AuSaveFile - save sound bucket data created by record widget

SYNOPSIS

```
#include <audio/Record.h>

extern void AuSaveFile(Widget widget);
```

DESCRIPTION

`AuSaveFile()` saves into a file the sound bucket data created by a record widget.

After recording, prepare for the `AuSaveFile()` call by calling `XtSetArg()` and `XtSetValue()`. For example:

```
XtSetArg(args[0], AuNfileName, path_name);
XtSetValue(recordWidget,args,1);
```

widget The name of a record widget.

ERRORS

`AuSaveFile()` does not return an error status.

DEPENDENCIES

This function belongs to the Audio Application Program Interface widget library. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

`AuInvokePlay()` was developed by HP.

SEE ALSO

AtAddCallback(), AtInitialize(), AtRemoveCallback(), AuCreatePlay(), AuCreateRecord(), AuInvokePlay(), AuInvokeRecord(), AuPlayWidget(), AuRecordWidget().

Using the Audio Application Program Interface.

NAME

AVendorRelease - get vendor release number of audio server for this connection

SYNOPSIS

```
#include <audio/Alib.h>

int AVendorRelease (Audio *audio);
```

DESCRIPTION

AVendorRelease() returns the vendor release number of the audio server for the connection specified by *audio*.

audio specifies the **Audio** structure associated with this connection.

RETURN VALUE

Upon successful completion, **AVendorRelease()** returns the vendor release number of the the audio server associated with this connection.

ERRORS

AVendorRelease() does not return an error status.

EXAMPLES

The following example returns the vendor release number of the audio server for the connection specified by *audio*.

```
int vendor_rel;     /* vendor release number for server */
Audio *audio;       /* audio connection */
      .
      .
      .
/* get vendor release number for server */
vendor_rel = AVendorRelease(audio);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

AVendorRelease() was developed by HP.

SEE ALSO

AProtocolRevision(3X), AProtocolVersion(3X), AServerVendor(3X).

Using the Audio Application Program Interface.

NAME

AWriteAFileHeader - write a header for an audio file

SYNOPSIS

```
#include <audio/Alib.h>

long
AWriteAFileHeader (
    Audio               *audio,
    char                *pathname,
    AFileFormat         file_format,
    AudioAttributes     *audio_attributes,
    long                *status_return
);
```

DESCRIPTION

AWriteAFileHeader() opens the specified file (truncating it to zero if it exists, or creating it if it does not exist), and writes a file header suitable for the specified file format and attributes.

audio	specifies the **Audio** structure associated with this connection.
pathname	the pathname of the audio data file for which a header will be written.
file_format	specifies format of the file for which the header will be written. Must be a valid format (not **AFFUnknown**).
audio_attributes	specifies attributes of the audio file for which the header will be written. Must be a complete audio attributes structure. The header's data length will be written as zero if the duration field of *audio_attributes* is set to **ATTFullLength**.
status_return	receives the returned status of the operation, unless it is set to NULL.

RETURN VALUE

Upon successful completion, **AWriteAFileHeader()** returns the length of the file header.

ERRORS

If *status_return* is not set to NULL, one of the following is returned in *status_return:*

0 AENoError
1 AESystem Call
6 AEBadFileFormat
7 AEBadDataFormat
17 AEOutOFMemory

EXAMPLE

The following example writes a file header to the file `/myhome/a_dir/a_file.au` .

```
AFileFormat     file_fmt;  /* file format */
Audio           * audio;  /* audio connection */
long            header_len;  /* length of header */
AudioAttributes
                attribs;  /* previously defined attributes */
long            status;  /* status */
                .
                .
                .
/* write a Sun/NeXT file header to /myhome/a_dir/a_file */
char  fname[] = /myhome/a_dir/a_file.au ;
file_fmt = AFFSun;
header_len = AWriteAFileHeader(audio, fname, file_fmt,
    &attribs, &status);
```

DEPENDENCIES

This function belongs to the Audio Library of functions that manage connections to an audio server. The audio server must run on a system that has audio hardware. To find out whether or not your system has audio hardware, refer to the hardware manual that accompanies your system.

AUTHOR

`AWriteAFileHeader( )` was developed by HP.

SEE ALSO

Using the Audio Application Program Interface.

NAME

j0, j1, jn, y0, y1, yn - Bessel functions

SYNOPSIS

```
#include <math.h>

double j0(double x);

double j1(double x);

double jn(int n, double x);

double y0(double x);

double y1(double x);

double yn(int n, double x);
```

DESCRIPTION

`j0()` and `j1()` return Bessel functions of x of the first kind of orders 0 and 1 respectively. `jn()` returns the Bessel function of x of the first kind of order n.

`y0()` and `y1()` return the Bessel functions of x of the second kind of orders 0 and 1 respectively. `yn()` returns the Bessel function of x of the second kind of order n. The value of x must be positive.

ERRORS

/lib/libm.a

Non-positive arguments cause `y0()`, `y1()`, and `yn()` to return the value `-HUGE_VAL` and to set `errno` to EDOM. They also cause a message indicating DOMAIN error to be printed on the standard error output.

Arguments too large in magnitude cause `j0()`, `j1()`, `jn()`, `y0()`, `y1()`, and `yn()` to return 0.0 and set `errno` to ERANGE. In addition, a message indicating TLOSS error is printed on the standard error output.

`j0()`, `j1()`, `jn()`, `y0()`, `y1()`, and `yn()` return `NaN` and set `errno` to EDOM when x is NaN or ±INFINITY. In addition, a message indicating DOMAIN error is printed on the standard error output.

These error-handling procedures can be changed with the function *matherr*(3M).

/lib/libM.a

No error messages are printed on the standard error output.

Non-positive arguments cause `y0()`, `y1()`, and `yn()` to return the value `NaN` and to set `errno` to EDOM.

Arguments too large in magnitude cause `j0()`, `j1()`, `jn()`, `y0()`, `y1()`, and `yn()` to return 0.0 and set `errno` to ERANGE.

`j0()`, `j1()`, `jn()`, `y0()`, `y1()`, and `yn()` return `NaN` and set `errno` to EDOM when x is NaN or ±INFINITY.

These error-handling procedures can be changed by using the `_matherr()` function (see *matherr*(3M)). Note that `_matherr()` is provided in order to assist in migrating programs from `libm.a` to `libM.a` and is *not* a part of XPG3, ANSI C, or POSIX.

SEE ALSO

isinf(3M), isnan(3M), matherr(3M).

STANDARDS CONFORMANCE

`j0()` in libm.a: AES, SVID2, XPG2, XPG3
`j0()` in libM.a: AES, XPG3, XPG4

`j1()` in libm.a: AES, SVID2, XPG2, XPG3
`j1()` in libM.a: AES, XPG3, XPG4

`jn()` in libm.a: AES, SVID2, XPG2, XPG3
`jn()` in libM.a: AES, XPG3, XPG4

`y0()` in libm.a: AES, SVID2, XPG2, XPG3
`y0()` in libM.a: AES, XPG3, XPG4

`y1 ( )` in libm.a: AES, SVID2, XPG2, XPG3
`y1 ( )` in libM.a: AES, XPG3, XPG4

`yn ( )` in libm.a: AES, SVID2, XPG2, XPG3
`yn ( )` in libM.a: AES, XPG3, XPG4

NAME

bindresvport() - bind socket to privileged IP port

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>

int bindresvport(int sd, struct sockaddr_in *sin);
```

DESCRIPTION

bindresvport() is used to bind a socket descriptor to a privileged IP port; that is, a port number in the range 0 to 1023. *sd* is a socket descriptor that was previously defined by a successful call to *socket*(2). Upon successful completion of **bindresvport()**, the *sin_port* field in the struct pointed to by *sin* contains the number of the privileged port bound to the *sd* socket. Due to the need to protect the port numbers used by various networking commands, **bindresvport()** only returns a port number within a smaller subrange in the range of 0 to 1023.

Only the super-user can bind to a privileged port; this call fails for any other users.

RETURN VALUE

bindresvport() returns 0 if successful. Otherwise it returns –1 and sets **errno** to indicate the cause of the error.

ERRORS

bindresvport fails if any of the following conditions are encountered:

[EPFNOSUPPORT]	The value specified in the *sin_family* field of the *sockaddr_in* struct was not **AF_INET**.
[EBADF]	*sd* is not a valid descriptor.
[ENOTSOCK]	*sd* is not a socket.
[EADDRNOTAVAIL]	The specified address is bad or not available from the local machine.
[EADDRINUSE]	The specified address is already in use.
[EINVAL]	The socket is already bound to an address, or the socket has been shut down.
[EAFNOSUPPORT]	Requested address does not match the address family of this socket.
[EACCESS]	The requested address is protected, and the current user has inadequate permission to access it.
[EOPNOTSUPP]	The socket whose descriptor is *sd* is of a type that does not support address binding.
[ENOBUFS]	Insufficient buffer memory is available.

AUTHOR

bindresvp() was developed by Sun Microsystems, Inc.

SEE ALSO

bind(2), socket(2).

NAME

blopen(), blclose(), blread(), blget(), blset() - terminal block-mode library interface

SYNOPSIS

```
#include <sys/blmodeio.h>

int blopen(int fildes);

int blclose(int bfdes);

int blread(int bfdes, char *buf, size_t nbyte);

int blget(int bfdes, struct blmodeio *arg);

int blset(int bfdes, const struct blmodeio *arg);
```

DESCRIPTION

This terminal library interface allows support of block-mode transfers with HP terminals. Block mode only affects input processing. Therefore, data is written with the standard `write()` interface (see *write*(2)).

In character mode, the terminal sends each character to the system as it is typed. However, in block mode, data is buffered and possibly edited locally in the terminal memory as it is typed, then sent as a block of data when the **[Enter]** key is pressed on the terminal. During block-mode data transmissions, the incoming data is not echoed by the interface and no special character processing is performed, other than recognizing a data block terminator character. For subsequent character mode transmissions, the existing termio state (see *termio*(7)) continues to determine echo and character processing.

Block-mode protocol has two component parts: block-mode handshake and block-mode transmission.

Block-Mode Handshake

At the beginning of a read, a *trigger* character is sent to the terminal to notify it that the system wants a block of data (the *trigger* character, if defined, is sent at the beginning of all reads, whether in character- or block-mode. It is necessary for block-mode reads to work correctly).

After receiving the *trigger* character, and when the user has typed all the data into the terminal's memory and pressed the **[Enter]** key, the terminal sends an *alert* character to the system to notify it that the terminal has a block of data to send.

The system might then send user-definable cursor-positioning or other data sequences to the terminal, such as for cursor-home or lock-keyboard.

The system then sends a second *trigger* character to the terminal. In response, the terminal transmits the data block as described in the Block-Mode Transmission section.

Block-Mode Transmission

The second part of the block-mode protocol is the block-mode transmission. After the block-mode handshake has successfully completed, the terminal transmits the data block to the system. During this transmission of data, the incoming data is not echoed by the system and no special character processing is performed, other than recognizing the data block termination character. It is possible to bypass the block-mode handshake and have the block-mode transmission occur after only the first *trigger* character is sent, see `CB_BMTRANS` below.

It is possible to intermix both character-mode and block-mode data transmissions. If `CB_BMTRANS` (see below) is set, all transfers are block-mode transfers. When is not set, character mode transmissions are processed as described in *termio*(7). In this case, if an *alert* character is received anywhere in the input data, the transmission mode is automatically switched to block mode for a single transmission. Any data received before the *alert* is discarded. The *alert* character can be escaped with a backslash (\) character.

XON/XOFF Flow Control

To prevent data loss, XON/XOFF flow control should be used between the system and the terminal. The IXOFF bit (see *termio*(7)) should be set and the terminal strapped appropriately. If flow control is not used, it is possible for incoming data to overflow and be lost. (Note: some older terminals do not support XON/XOFF flow control.)

Read Requests

Read requests that receive data from block-mode transmissions do not return until the transmission is complete (the terminal has transmitted all characters). If the read is satisfied by byte count or if a data transmission error occurs, all subsequent data is discarded until the transmission is complete. The read

waits until a terminator character is seen, or until a time interval specified by the system has passed that is longer than necessary for the number of characters specified.

The data-block-terminator character is included in the data returned to the user, and is included in the byte count. If the number of bytes transferred by the terminal in a block-mode transfer exceeds the number of bytes requested by the user, the read returns the requested number of bytes and the remaining bytes are discarded. The user can determine if data was discarded by checking the last character of the returned data. If the last character is not the terminator character, then more data was received than was requested and data was discarded.

The EIO error can be caused by several events, including errors in transmission, framing, parity, break, and overrun, or if the internal timer expires. The internal timer starts when the second trigger character is sent by the computer, and ends when the terminating character is received by the computer. The length of this timer is determined by the number of bytes requested in the read and the current baud rate, plus an additional ten seconds.

User Control of Handshaking

If desired, the application program can provide its own handshake mechanism in response to the *alert* character by selecting the `OWNTERM` mode (see `CB_OWNTERM` below). With this mode selected, the driver completes a read request when the *alert* character is received. No data is discarded before the *alert*, and the *alert* is returned in the data read. The *alert* character may be escaped with a backslash (\) character. The second *trigger* is sent when the application issues the next read.

blmode Control Calls

First, the standard `open()` call to a tty device must be made to obtain a file descriptor for the subsequent block-mode control calls (an `open()` is done automatically by the system for `stdin` on the terminal).

```
int bfdes;
```

`bfdes = blopen (int fildes)`

A call to `blopen()` must be made before any block-mode access is allowed on the specified file descriptor. `blopen()` initializes the block-mode parameters as described below. The return value from `blopen()` is a block-mode file descriptor that must be passed to all subsequent block-mode control calls.

`int blclose (int bfdes)`

A call to `blclose()` must be issued before the standard `close()` to ensure proper closure of the device (see *close*(2)). Otherwise unpredictable results can occur. The argument *bfdes* is the file descriptor returned from a previous `blopen()` system call.

`int blread (int bfdes, char *buf, size_t nbyte)`

The `blread()` routine has the same parameters as the `read()` sytem call (see *read*(2)). At the beginning of a read, the `cb_triglc` character (if defined) is sent to the device. If `CB_BMTRANS` is not set, and no `cb_alertc` character is received, the read data is processed according to *termio*(7). If `CB_BMTRANS` is set, or if a non-escaped `cb_alertc` character is received, echo is turned off for the duration of the transfer, and no further special character processing is done other than that required for the termination character. The argument *bfdes* is the file descriptor returned from a previous `blopen()` system call.

`int blget (int bfdes, struct blmodeio *arg)`

A call to `blget()` returns the current values of the `blmodeio` structure (see below). The argument *bfdes* is the file descriptor returned from a previous `blopen()` system call.

`int blset (int bfdes, const struct blmodeio *arg)`

A call to `blset()` sets the block-mode values from the structure whose address is *arg*. The argument *bfdes* is the file descriptor returned from a previous `blopen()` system call.

blmode Structure

The two block-mode control calls, `blget()` and `blset()`, use the following structure, defined in `<sys/blmodeio.h>`:

```
#define     NBREPLY     64
struct      blmodeio      {
    unsigned long       cb_flags;           /* Modes */
    unsigned char       cb_triglc;          /* First trigger */
```

```
        unsigned char     cb_trig2c;          /* Second trigger */
        unsigned char     cb_alertc;          /* Alert character */
        unsigned char     cb_termc;           /* Terminating char */
        unsigned char     cb_replen;          /* cb_reply length */
        char              cb_reply[NBREPLY];  /* optional reply */
};
```

The **cb_flags** field controls the basic block-mode protocol:

CB_BMTRANS	0000001	Enable mandatory block-mode transmission.
CB_OWNTERM	0000002	Enable user control of handshake.

If **CB_BMTRANS** is set, all transmissions are processed as block-mode transmissions. The block-mode handshake is not required and data read is processed as block-mode transfer data. The block-mode handshake can still be invoked by receipt of an *alert* character as the first character seen. A **blread()** issued with the **CB_BMTRANS** bit set causes any existing input buffer data to be flushed.

If **CB_BMTRANS** is not set, and if the *alert* character is defined and is detected anywhere in the input stream, the input buffer is flushed and the block-mode handshake is invoked. The system then sends the **cb_trig2c** character to the terminal, and a block-mode transfer follows. The *alert* character can be escaped by preceding it with a backslash (\).

If **CB_OWNTERM** is set, reads are terminated upon receipt of a non-escaped *alert* character. No input buffer flushing is performed, and the *alert* character is returned in the data read. This allows application code to perform its own block-mode handshaking. If the bit is clear, a non-escaped *alert* character causes normal block-mode handshaking to be used.

The initial **cb_flags** value is all-bits-cleared.

There are several special characters (both input and output) that are used with block mode. These characters and the initial values for these characters are described below. Any of these characters can be undefined by setting its value to 0377.

cb_trig1c (default DC1) is the initial *trigger* character sent to the terminal at the beginning of a read request.

cb_trig2c (default DC1) is the secondary *trigger* character sent to the terminal after the *alert* character has been seen.

cb_alertc (default DC2) is the *alert* character sent by the terminal in response to the first *trigger* character. It signifies that the terminal is ready to send the data block. The *alert* character can be escaped by preceding it with a backslash ("\").

cb_termc (default RS) is sent by the terminal after the block-mode transfer has completed. It signifies the end of the data block to the computer.

The **cb_replen** field specifies the length in bytes of the **cb_reply** field. If set to zero, the **cb_reply** string is not used. The **cb_replen** field is initially set to zero.

The **cb_reply** array contains a string to be sent out after receipt of the *alert* character, but before the second *trigger* character is sent by the computer. Any character can be included in the reply string. The number of characters sent is specified by **cb_replen**. The initial value of all characters in the **cb_reply** array is NULL.

RETURNS

If an error occurs, all calls return a value of –1 and **errno** is set to indicate the error. If no error is detected, **blread()** returns the number of characters read. All other calls return 0 upon successful completion.

During a read, it is possible for the user's buffer to be altered, even if an error value is returned. The data in the user's buffer should be ignored as it is not complete. The following errors can be returned by the library calls indicated:

blopen()

[ENOTTY] The file descriptor specified is not related to a terminal device.

blclose()

[ENOTTY] No previous **blopen** has been issued for the specified file descriptor.

blread()

[EDEADLK] A resource deadlock would occur as a result of this operation (see *lockf*(2)).

[EFAULT] **buf** points outside the allocated address space. The reliable detection of this error is implementation dependent.

[EINTR] A signal was caught during the **read** system call.

[EIO] An I/O error occured during block-mode data transmissions.

[ENOTTY] No previous **blopen** has been issued for the specified file descriptor.

blget()

[ENOTTY] No previous **blopen** has been issued for the specified file descriptor.

blset()

[EINVAL] An illegal value was specified in the structure passed to the system.

[ENOTTY] No previous **blopen** has been issued for the specified file descriptor.

WARNINGS

Once **blopen** has been called with a file descriptor and returned successfully, that file descriptor should not subsequently be used as a parameter to the following system calls: **close()**, **dup()**, **dup2()**, **fcntl()**, **ioctl()**, **read()**, or **select()** until a **blclose** is called with the same file descriptor as its parameter. Additionally, **scanf()**, **fscanf()**, **getc()**, **getchar()**, **fgetc()**, and **fgetw()** should not be called for a stream associated with a file descriptor that has been used in a **blopen()** call but has not been used in a **blclose()** call. These functions call **read()**, and calling these routines results in unpredictable behavior.

AUTHOR

blopen(), **blclose()**, **blread()**, **blget()**, and **blset()** were developed by HP.

SEE ALSO

termio(7).

NAME

bsearch() - binary search a sorted table

SYNOPSIS

```
#include <stdlib.h>

void *bsearch(
    const void *key,
    const void *base,
    size_t nel,
    size_t size,
    int (*compar)(const void *, const void *)
);
```

DESCRIPTION

`bsearch()` is a binary search routine generalized from Knuth (6.2.1) Algorithm B. It returns a pointer into a table indicating where a datum may be found. The table must be previously sorted in increasing order according to a provided comparison function. *key* points to a datum instance to be sought in the table. *base* points to the element at the base of the table. *nel* is the number of elements in the table. *size* is the size of each element in the table. *compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. The function must return an integer less than, equal to, or greater than zero indicating that the first argument is to be considered less than, equal to, or greater than the second.

NOTES

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-void.

The comparison function need not compare every byte, so arbitrary data can be contained in the elements in addition to the values being compared.

Although declared as type pointer-to-void, the value returned should be cast into type pointer-to-element.

RETURN VALUE

A NULL pointer is returned if the key cannot be found in the table.

EXAMPLES

The example below searches a table containing pointers to nodes consisting of a string and its length. The table is ordered alphabetically on the string in the node pointed to by each entry.

This code fragment reads in strings and either finds the corresponding node and prints out the string and its length, or prints an error message.

```
#include <stdio.h>

#define TABSIZE  1000

struct node {                          /* these are stored in the table */
    char *string;
    int length;
};
struct node table[TABSIZE];            /* table to be searched */
    .
    .
    .
{
    struct node *node_ptr, node;
    int node_compare( );   /* routine to compare 2 nodes */
    char str_space[20];    /* space to read string into */
    .
    .
    .
    node.string = str_space;
    while (scanf("%s", node.string) != EOF) {
```

```
            node_ptr = (struct node *)bsearch((void *)(&node),
                   (void *)table, TABSIZE,
                   sizeof(struct node), node_compare);
            if (node_ptr != NULL) {
                (void)printf("string = %20s, length = %d\n",
                    node_ptr->string, node_ptr->length);
            } else {
                (void)printf("not found: %s\n", node.string);
            }
        }
    }
    /* This routine compares two nodes based on an
          alphabetical ordering of the string field.  */
    int
    node_compare(node1, node2)
    struct node *node1, *node2;
    {
        return strcmp(node1->string, node2->string);
    }
```

WARNINGS

If the table being searched contains two or more entries that match the selection criteria, a random entry is returned by **bsearch()** as determined by the search algorithm.

SEE ALSO

hsearch(3C), lsearch(3C), qsort(3C), tsearch(3C).

STANDARDS CONFORMANCE

bsearch(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

htonl(), htons(), ntohl(), ntohs() - convert values between host and network byte order

SYNOPSIS

```
#include <netinet/in.h>

unsigned long htonl(unsigned long hostlong);

unsigned short htons(unsigned short hostshort);

unsigned long ntohl(unsigned long netlong);

unsigned short ntohs(unsigned short netshort);
```

DESCRIPTION

These routines convert 16- and 32-bit quantities between network byte order and host byte order. On HP-UX systems, network and host byte orders are identical, so these routines are defined as null macros in the include file `<netinet/in.h>`.

These routines are most often used in conjunction with Internet addresses and ports as returned by `gethostent()` and `getservent()` (see *gethostent*(3N) and *getservent*(3N)). Use these routines to write portable programs.

AUTHOR

`byteorder()` was developed by the University of California, Berkeley.

SEE ALSO

gethostent(3N), getservent(3N).

NAME

cachectl() - flush and/or purge the cache

SYNOPSIS

```
#include <sys/cache.h>

int cachectl(int cachecmd, void *address, size_t length);
```

DESCRIPTION

`cachectl()` permits a program to flush or purge data in the data and/or instruction caches. The features provided by `cachectl()` are not needed by most programs. It is primarily used for programs that do dynamic loading or contain self-modifying code. Programs that do dynamic loading or contain self-modifying code can use the `CC_IPURGE` request, after the new code has been written to memory, to ensure that the correct code will be fetched by the instruction cache during execution. The `CC_FLUSH`, `CC_PURGE`, and `CC_EXTPURGE` requests should only be used by applications that are highly hardware dependent and which have detailed knowledge of hardware internals.

The *cachecmd* parameter specifies what operations to carry out on the cache or chaches. *cachecmd* should contain one of the following values, which are defined in `<sys/cache.h>`:

`CC_PURGE` Purge the cache. Dirty cache entries are discarded without being written to memory. A "dirty" cache entry is an entry that has been modified, but has not been written back to the corresponding memory location.

`CC_FLUSH` Flush the cache. Dirty cache entries are copied back to the corresponding memory locations. This operation is the same as `CC_PURGE` on models that do not have a copyback cache.

`CC_IPURGE` Flush any dirty data cache entries, then purge any instruction cache entries which are "stale". A "stale" instruction cache entry is an entry that is older than the corresponding memory location. This can happen if the corresponding memory location was written to (via the data cache). This operation is useful for self-modifying code.

The following mask, defined in `<sys/cache.h>`, can be ORed together with one of the above values in order to purge the external cache (if one exists) at the same time.

`CC_EXTPURGE` Purge the external cache (if any).

The *address* parameter specifies the start address of the area to be flushed and/or purged. If the specified start address is a null pointer, the operation is applied to the entire cache or caches specified by the *cachecmd* parameter. Selective flushing and/or purging is not supported on all models. Some models have restrictions on the legal values for the address parameter. See DEPENDENCIES for details about specific hardware.

The *length* parameter is used only when a start address is specified. It controls the length of the area to be flushed or purged.

EXAMPLES

The following call to `cachectl()` requests that the entire data cache be flushed, followed by a purge of the instruction cache.

```
cachectl (CC_IPURGE, 0, 0);
```

RETURN VALUE

`cachectl()` returns `0` if the operation succeeds. Otherwise it returns -1. The semantics of `cachectl()`, when the *address* parameter contains a bad address, is subject to change and may vary from machine to machine.

ERRORS

`cachectl()` fails and sets `errno` to the value indicated if:

[EINVAL] *cachecmd* is not a a valid request.

DEPENDENCIES

Series 300/400

The MC68020 and MC68030 processors do not have a copyback cache. Selective purging is not supported for the MC68020 and MC68030 processors. Selective purging and flushing is supported on the MC68040 processor,

but only under the following conditions:

- If the *length* parameter is 16, the cache line which includes *address* is flushed and/or purged (i.e., the 4 least significant bits of the address are ignored).
- If the *length* parameter is 4096, the page which includes *address* is flushed and/or purged (i.e., the 12 least significant bits of the address are ignored). If the *length* parameter is not 16 or 4096, the operation is applied to the entire cache or caches specified by the *cachecmd* parameter.

On the MC68040 microprocessor, `CC_PURGE` instead performs a `CC_FLUSH` if the *length* parameter is not 16 or 4096.

AUTHOR

`cachectl()` was developed by HP.

NAME

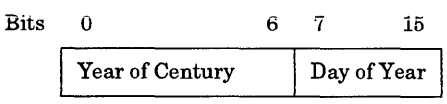
calendar() - return the MPE calendar date

SYNOPSIS

```
#include <portnls.h>

unsigned short calendar(void);
```

DESCRIPTION

This routine returns the calendar date in the format:



RETURN VALUE

An unsigned short integer containing the calendar format.

WARNINGS

This routine is provided for compatibility with MPE, another HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described on *hpnls*(5) for HP-UX NLS support.

AUTHOR

`calendar()` was developed by HP.

SEE ALSO

portnls(5).

NAME

catgetmsg() - get message from a message catalog

SYNOPSIS

```
#include <nl_types.h>

char *catgetmsg(
    nl_catd catd,
    int set_num,
    int msg_num,
    char *buf,
    size_t buflen
);
```

DESCRIPTION

`catgetmsg()` reads message *msg_num* in set *set_num* from the message catalog indentified by *catd*, a catalog descriptor returned from a previous call to `catopen()` (see *catopen*(3C)). The return message is stored in *buf*, a buffer of length *buflen* bytes.

A message longer than *buflen* - 1 bytes is silently truncated. The return message is always terminated with a null byte.

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUE

If successful, `catgetmsg()` returns a pointer to the message in *buf*. Otherwise, `catgetmsg()` returns a pointer to an empty (null) string and sets `errno` to indicate the error. If *buflen* is greater than zero, the pointer returned is *buf*.

ERRORS

`catgetmsg()` fails and `errno` is set to the value indicated if one of the following conditions is true:

[EBADF]	*catd* is not a valid catalog descriptor.
[EINVAL]	*buflen* is less than 1.
[EINVAL]	*set_num* is not in the message catalog.
[EINVAL]	The message catalog identified by *catd* is corrupted.
[EINTR]	A signal was caught during the `read()` system call.
[EFAULT]	*buf* points outside the allocated address space. The reliable detection of this error is implementation dependent.
[ENOMSG]	*msg_num* is not in the message catalog.
[ERANGE]	A message longer than *buflen* - 1 bytes was truncated.

AUTHOR

`catgetmsg()` was developed by HP.

SEE ALSO

catopen(3C), catgets(3C), read(2).

STANDARDS CONFORMANCE

`catgetmsg()`: XPG2

NAME

catgets() - get a program message

SYNOPSIS

```
#include <nl_types.h>

char *catgets(
    nl_catd catd,
    int set_num,
    int msg_num,
    const char *def_str
);
```

DESCRIPTION

`catgets()` reads message *msg_num* in set *set_num* from the message catalog identified by *catd*, a catalog descriptor returned from a previous call to `catopen()` (see *catopen*(3C)). *def_str* points to a default message string returned by `catgets()` if the call fails.

A message longer than **NL_TEXTMAX** bytes is truncated. The returned message string is always terminated with a null byte. **NL_TEXTMAX** is defined in `<limits.h>`.

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUE

If the call is successful, `catgets()` returns a pointer to an internal buffer area containing the null-terminated message string. If the call is unsuccessful, `catgets()` returns a pointer to *def_str*.

ERRORS

`catgets()` fails and sets `errno` if the following condition is encountered:

[EBADF]	*catd* is not a valid catalog descriptor.
[EINTR]	A signal was caught during the *read*(2) system call.
[EINVAL]	The message catalog identified by *catd* is corrupted.
[ENOMSG]	Message identified by *set_num* or *msg_num* is not in the message catalog.
[ERANGE]	A message longer than **NL_TEXTMAX** bytes was truncated.

`catgets()` fails and `errno` if any of the following conditions are encountered:

WARNINGS

`catgets()` returns a pointer to a static area that is overwritten on each call.

AUTHOR

`catgets()` was developed by HP.

SEE ALSO

catopen(3C), catgetmsg(3C).

STANDARDS CONFORMANCE

`catgets()`: AES, XPG2, XPG3, XPG4

NAME

catopen(), catclose() - open and close a message catalog for reading

SYNOPSIS

```
#include <nl_types.h>

nl_catd catopen(const char *name, int oflag);

int catclose(nl_catd catd);
```

DESCRIPTION

`catopen()` opens a message catalog and returns a catalog descriptor. *name* specifies the name of the message catalog being opened. A *name* containing a slash (/) specifies a path name for the message catalog. Otherwise, the environment variable **NLSPATH** is used (see *environ*(5)). If **NLSPATH** specifies more than one path, `catopen()` returns the catalog descriptor for the first path on which it is able to successfully open the specified message catalog. If **NLSPATH** does not exist in the environment, or if a message catalog cannot be opened for any **NLSPATH**-specified path, `catopen()` uses a system-wide default path. The default is affected by the setting of **LANG**. *name* must not contain **%N**.

A message catalog descriptor remains valid in a process until the process closes it, or until a successful call to one of the `exec()` functions. A change in the setting of **LANG** category has no effect on existing open catalogs.

If a file descriptor is used to implement message catalog descriptors, the **FD_CLOEXEC** flag will be set; see `<fcntl.h>`.

oflag is reserved for future use and should be set to zero (`0`). The results of setting this field to any other value are undefined.

`catclose()` closes the message catalog *catd*, a message catalog descriptor returned from an earlier successful call to `catopen()`.

RETURN VALUE

`catopen()` returns a message catalog descriptor if successful. Otherwise, a value of (*nl_catd*)−1 is returned and `errno` is set to indicate the error.

`catclose()` returns `0` if successful. Otherwise, −1 is returned and `errno` is set to indicate the error.

ERRORS

`catopen()` fails, no message catalog is opened, and `errno` is set for the last path attempted if any of the following conditions is true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named catalog does not exist.
[ENOENT]	The path is null.
[EACCES]	A component of the path prefix denies search permission.
[EACCES]	Read permission is denied for the named file.
[EMFILE]	The maximum number of file descriptors allowed are currently open.
[ENAMETOOLONG]	The length of the specified path name exceeds **PATH_MAX** bytes, or the length of a component of the path name exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect.
[ENFILE]	The system file table is full.
[ENOTDIR]	A component of the path prefix is not a directory.

`catclose()` fails if the following is true:

[EBADF]	*catd* is not a valid open message-catalog descriptor.

WARNINGS

When using **NLSPATH**, `catopen()` does not provide a default value for **LANG**.

NOTES

`catgets()` can be used to provide default messages when called following a failed `catopen()` (see

catgets(3C)). `catgets()` returns its *def_str* parameter if it is passed an invalid catalog descriptor.

AUTHOR

`catopen()` was developed by HP.

FILES

`/usr/lib/nls` Message catalog default path.

SEE ALSO

catgets(3C), environ(5), <fcntl.h>, <nl_types.h>.

STANDARDS CONFORMANCE

`catopen()`: AES, XPG2, XPG3, XPG4

`catclose()`: AES, XPG2, XPG3, XPG4

NAME

catread( ) - MPE/RTE-style message catalog support

SYNOPSIS

```
#include <portnls.h>

int catread(
    int fd,
    int set_num,
    int msg_num,
    char *msg_buf,
    int buflen,
    /* arg, */ ...
);
```

DESCRIPTION

`catread()` reads message number *msg_num* of set *set_num* in the message catalog identified by *fd*, a file descriptor returned from a previous call to `open()` (see *open*(2)). The return message is stored in *buf*, a buffer of length *buflen* bytes.

The message read from the catalog can have embedded formatting information in the form `!`[*digit*]. Exclamation marks must be all numbered or all unnumbered. If exclamation marks are numbered, an exclamation mark followed by digit *n* is replaced by the *n*th *arg*. If exclamation marks are unnumbered, they are replaced by the *args* in serial order. If there are fewer *args* than exclamation marks, the results are undefined. If there are more *args* than exclamation marks, the excess *args* are ignored.

A character in a message can be quoted (that is, made to stand for itself) by preceding it with a tilde (~). To use the special characters `!` or `~` in a message, preceed the special character with `~`.

A message longer than *buflen*– 1 bytes is silently truncated. The return message is always terminated with a null byte.

`catread()` is provided to support message catalog applications from The HP MPE and RTE operating systems.

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUE

If successful, `catread()` returns the length, in bytes, of the formatted message in *msg_buf*. Otherwise, if *set_num* or *msg_num* is not found in the catalog, `catread()` returns a negative integer.

ERRORS

`catread()` succeeds, but sets `errno` if the following condition is true:

[ERANGE] Formatted message exceeds *buflen* –1 bytes.

AUTHOR

`catread()` was developed by HP.

SEE ALSO

gencat(1), getmsg(3C), hpnls(5).

NAME

cfgetospeed(), cfsetospeed(), cfgetispeed(), cfsetispeed() - tty baud rate functions

SYNOPSIS

```
#include <termios.h>

speed_t cfgetospeed(const struct termios *termios_p);

int cfsetospeed(struct termios *termios_p, speed_t speed);

speed_t cfgetispeed(const struct termios *termios_p);

int cfsetispeed(struct termios *termios_p, speed_t speed);
```

DESCRIPTION

These functions set and get the input and output speed codes in the *termios* structure referenced by *termios_p*. The *termios* structure contains these speed codes representing input and output baud rates as well as other terminal related parameters. Setting the parameters on a terminal file does not become effective until `tcsetattr()` is successfully called.

`cfgetospeed()` returns the output speed code from the *termios* structure referenced by *termios_p*.

`cfsetospeed()` sets the output speed code in the *termios* structure referenced by *termios_p* to *speed*. The speed code for a baud rate of zero, `B0`, is used to terminate the connection. If `B0` is specified, the modem control lines are no longer asserted, which normally disconnects the line.

`cfgetispeed()` returns the input speed code from the *termios* structure referenced by *termios_p*.

`cfsetispeed()` sets the input speed code in the *termios* structure referenced by *termios_p* to *speed*.

RETURN VALUE

`cfgetospeed()` returns the output speed code from the *termios* structure referenced by *termios_p*.

`cfgetispeed()` returns the input speed code from the *termios* structure referenced by *termios_p*.

`cfsetispeed()` and `cfsetospeed()` return zero upon successful completion. Otherwise, they return -1 and set `errno` to indicate the error.

ERRORS

`cfsetispeed()` and `cfsetospeed()` fail when the following condition is encountered:

[EINVAL] The value of *speed* is outside the range of possible speed codes as specified in `<termios.h>`.

WARNINGS

`cfsetispeed()` and `cfsetospeed()` can be used to set speed codes in the *termios* structure that are not supported by the terminal hardware.

SEE ALSO

tcattribute(3C), termio(7).

STANDARDS CONFORMANCE

`cfgetispeed()`: AES, XPG3, XPG4, FIPS 151-2, POSIX.1

`cfgetospeed()`: AES, XPG3, XPG4, FIPS 151-2, POSIX.1

`cfsetispeed()`: AES, XPG3, XPG4, FIPS 151-2, POSIX.1

`cfsetospeed()`: AES, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

chownacl() - change owner and/or group represented in a file's access control list (ACL)

SYNOPSIS

```
#include <acllib.h>

void chownacl(
    int nentries,
    const struct acl_entry *acl,
    uid_t olduid,
    gid_t oldgid,
    uid_t newuid,
    gid_t newgid
);
```

Remarks:

To ensure continued conformance with emerging industry standards, features described in this manual entry are likely to change in a future release.

DESCRIPTION

This routine alters an access control list (ACL) to reflect the change in a file's owner or group ID when an old file is copied to a new file and the ACL is also copied. `chownacl()` transfers ownership (that is, it modifies base ACL entries) in a manner similar to `chown()` (see *chown*(2)). The algorithm is described below and also in *acl*(5).

The *nentries* parameter is the current number of ACL entries in the `acl[]` array (zero or more; a negative value is treated as zero). The *olduid* and *oldgid* values are the user and group IDs of the original file's owner, typically the `st_uid` and `st_gid` values from `stat()` (see *stat*(2)). The *newuid* and *newgid* values are the user and group IDs of the new file's owner, typically the return values from `geteuid()` and `getegid()` (see *geteuid*(2) and *getegid*(2)).

If an ACL entry in `acl[]` has a *uid* of *olduid* and a *gid* of `ACL_NSGROUP` (that is, an owner base ACL entry), `chownacl()` changes *uid* to *newuid* (with exceptions – see below). If an entry has a *uid* of `ACL_NSUSER` and a *gid* of *oldgid* (that is, a group base ACL entry), `chownacl()` changes *gid* to *newgid*. In either case, only the last matching ACL entry is altered; a valid ACL can have only one of each type.

As with *chown*(2), if the new user or group already has an ACL entry (that is, a *uid* of *newuid* and a *gid* of `ACL_NSGROUP`, or a *uid* of `ACL_NSUSER` and a *gid* of *newgid*), `chownacl()` does not change the old user or group base ACL entry; both the old and new ACL entries are preserved.

As a special case, if *olduid* (*oldgid*) is equal to *newuid* (*newgid*), `chownacl()` does not search `acl[]` for an old user (group) base ACL entry to change. Calling it with both *olduid* equal to *newuid* and *oldgid* equal to *newgid* causes `chownacl()` to do nothing.

Suggested Use

This routine is useful in a program that creates a new or replacement copy of a file whose original was (or possibly was) owned by a different user or group, and that copies the old file's ACL to the new file. Copying another user's and/or group's file is equivalent to having the original file's owner and/or group copy and then transfer a file to a new owner and/or group using `chown()`. This routine is not needed for merely changing a file's ownership; `chown()` modifies the ACL appropriately in that case.

If a program also copies file miscellaneous mode bits from an old file to a new one, it must use `chmod()` (see *chmod*(2)). However, since `chmod()` deletes optional ACL entries, it must be called before `setacl()` (see *setacl*(2)). Furthermore, to avoid leaving a new file temporarily unprotected, the `chmod()` call should set only the file miscellanous mode bits, with all access permission mode bits set to zero (that is, mask the mode with 07000). The `cpacl()` library call encapsulates this operation, and handles remote files appropriately too.

EXAMPLES

The following code fragment gets `stat()` information and the ACL from `oldfile`, transfers ownership of `newfile` to the caller, and sets the revised ACL to `newfile`.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/acl.h>
```

```
int nentries;
struct acl_entry acl [NACLENTRIES];
struct stat statbuf;

if (stat ("oldfile", & statbuf) < 0)
    error (...);

if ((nentries = getacl ("oldfile", NACLENTRIES, acl)) < 0)
    error (...);

chownacl (nentries, acl, statbuf.st_uid, statbuf.st_gid,
    geteuid(), getegid());

if (setacl ("newfile", nentries, acl))
    error (...);
```

AUTHOR

`chownacl()` was developed by HP.

SEE ALSO

chown(2), getacl(2), getegid(2), geteuid(2), setacl(2), stat(2), acltostr(3C), cpacl(3C), setaclentry(3C), strtoacl(3C), acl(5).

NAME

clearenv - clear the process environment

SYNOPSIS

```
#include <stdlib.h>

int clearenv(void);
```

DESCRIPTION

`clearenv()` clears the process environment. No environment variables are defined immediately after a call to `clearenv()`.

`clearenv()` modifies the value of the pointer *environ*. This means that copies of that pointer are invalid after a call to `clearenv()`.

RETURN VALUE

Upon successful completion, `clearenv()` returns zero; otherwise, it returns –1 and sets `errno` to indicate the error.

ERRORS

`clearenv()` fails if the following condition is encountered:

[ENOMEM] Failed to free or reallocate memory for the process environment.

SEE ALSO

environ(5), getenv(3C), putenv(3C), <stdlib.h>.

STANDARDS CONFORMANCE

`clearenv()`: AES

NAME

clock() - report CPU time used

SYNOPSIS

```
#include <time.h>

clock_t clock(void);
```

DESCRIPTION

`clock()` returns the amount of CPU time (in microseconds) used since the first call to `clock()`. The time reported is the sum of the user and system times of the calling process and its terminated child processes for which it has executed `wait()` or `system()` (see *wait*(2) and *system*(3S)). To determine the time in seconds, the value returned by `clock()` should be divided by the value of the macro `CLOCKS_PER_SEC`.

The resolution of the clock varies, depending on the hardware and on software configuration.

If the processor time used is not available or its value cannot be represented, the function returns the value `(clock_t)-1`.

WARNINGS

The value returned by `clock()` is defined in microseconds for compatibility with systems that have CPU clocks with much higher resolution. Because of this, the value returned wraps around after accumulating only 2147 seconds of CPU time (about 36 minutes).

DEPENDENCIES

Series 300/400

The clock resolution is 20 milliseconds.

Series 700/800

The default clock resolution is 10 milliseconds.

SEE ALSO

times(2), wait(2), system(3S).

STANDARDS CONFORMANCE

`clock()`: AES, SVID2, XPG2, XPG3, XPG4, ANSI C

NAME

clock() - return the MPE clock value

SYNOPSIS

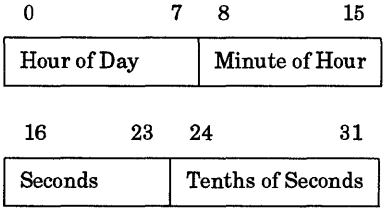
```
#include <portnls.h>

unsigned int clock(void);
```

DESCRIPTION

This routine returns the clock value in the MPE format.

RETURN VALUE

`clock()` returns an unsigned int in the format:



WARNINGS

This routine is provided for compatibility with the HP MPE operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described in *hpnls*(5) for HP-UX NLS support.

AUTHOR

`clock()` was developed by HP.

SEE ALSO

nlconvclock(3X), nlfmtclock(3X), portnls(5).

NAME

confstr() - get string-valued configuration values

SYNOPSIS

```
#include <unistd.h>

size_t confstr(int name, char *buf, size_t len);
```

DESCRIPTION

`confstr()` provides a method for applications to get configuration-defined string values. Its use and purpose are similar to `sysconf()` (see *sysconf*(2)) function, except that it is used where string values rather than numeric values are returned.

The *name* parameter can take on the following *name* values, which are defined in <`unistd.h`>.

`_CS_PATH` A default value for the `PATH` environment variable which can be used to locate commands in Section 1 of the *HP-UX Reference* and utilities defined in the POSIX.2 standard that are currently implemented in the HP-UX operating system.

If *len* is not zero, and if *name* is known and has a configuration-defined value, `confstr()` copies that value into the *len*-byte buffer pointed to by *buf*. If the string to be returned is longer than *len* bytes, including the terminating null, `confstr()` truncates the string to *len* - 1 bytes and null-terminates the result. The application can detect that the string was truncated by comparing the value returned by `confstr()` with *len*.

If *len* is zero and *buf* is NULL, `confstr()` returns the integer value as defined below, but does not return a string. If *len* is zero but *buf* is not NULL, the result is unspecified.

RETURN VALUE

If *name* is invalid, `confstr()` returns zero and sets `errno` to EINVAL.

If *name* does not have a configuration-defined value, `confstr()` returns 1 and returns a null string in *buf*.

If *name* has a configuration-defined value, `confstr()` returns the size of buffer that would be needed to hold the entire configuration-defined value. If this return value is less than *len*, the string returned in *buf* has been truncated.

FILES

`/usr/include/unistd.h`

EXAMPLES

The following code fragment calls `confstr()` to determine the correct buffer size for `_CS_PATH`, allocates space for this buffer, then gets the configuration value for `_CS_PATH`.

```
#include <unistd.h>
#include <stddef.h>

size_t bufsize;
char *buffer;

bufsize=confstr(_CS_PATH,NULL,(size_t)0);
buffer=(char *)malloc(bufsize);
confstr(_CS_PATH,buffer,bufsize);
```

AUTHOR

`confstr()` was developed by HP.

SEE ALSO

getconf(1), errno(2), sysconf(2), pathconf(2), fpathconf(2), malloc(3C).

STANDARDS CONFORMANCE

`confstr()`: XPG4, POSIX.2

NAME

toupper(), tolower(), _toupper(), _tolower(), toascii() - translate characters

SYNOPSIS

```
#include <ctype.h>

int toupper(int c);

int tolower(int c);

int _toupper(int c);

int _tolower(int c);

int toascii(int c);
```

DESCRIPTION

`toupper()` and `tolower()` have as domain the range of *getc*(3S): the integers from –1 through 255. If the argument of `toupper()` represents a lowercase letter, the result is the corresponding uppercase letter. If the argument of `tolower()` represents an uppercase letter, the result is the corresponding lowercase letter. All other arguments in the domain are returned unchanged. Arguments outside the domain cause undefined results.

The macros `_toupper()` and `_tolower()` perform the same translations as `toupper()` and `tolower()`, but have restricted domains and are faster. The domains of `_toupper()` and `_tolower()` are the integers from 0 through 255. Arguments outside of the domain cause undefined results.

`toascii()` yields its argument with all bits turned off that are not part of a standard 7-bit ASCII character; it is intended for compatibility with other systems.

WARNING

`toascii()` is supplied both as a library function and as a macro defined in the `<ctype.h>` header. Normally, the macro version is used. To obtain the library function, either use a `#undef` to remove the macro definition or, if compiling in ANSI C mode, enclose the function name in parenthesis or take its address. The following examples use the library function for `toascii()`:

```
#include <ctype.h>
#undef toascii
     ...
main()
{
          ...
     c1 = toascii(c);
          ...
}
```

or

```
#include <ctype.h>
     ...
main()
{
   int (*conv_func)();
        ...
   c1 = (toascii)(c);
        ...
   conv_func = toascii;
        ...
}
```

EXTERNAL INFLUENCES

Locale

The `LC_CTYPE` category determines the translations to be done.

International Code Set Support

Single-byte character code sets are supported.

AUTHOR

`conv()` was developed by AT&T and HP.

SEE ALSO

ctype(3C), getc(3S), setlocale(3C), lang(5).

STANDARDS CONFORMANCE

`_tolower()`: AES, SVID2, XPG2, XPG3, XPG4

`_toupper()`: AES, SVID2, XPG2, XPG3, XPG4

`toascii()`: AES, SVID2, XPG2, XPG3, XPG4

`tolower()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`toupper()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

cpacl(), fcpacl() - copy the access control list (ACL) and mode bits from one file to another

SYNOPSIS

```
#include <acllib.h>

int cpacl(
    const char *fromfile,
    const char *tofile,
    mode_t frommode,
    uid_t fromuid,
    gid_t fromgid,
    uid_t touid,
    gid_t togid
);

int fcpacl(
    int fromfd,
    int tofd,
    mode_t frommode,
    uid_t fromuid,
    gid_t fromgid,
    uid_t touid,
    gid_t togid
);
```

Remarks:

To ensure continued conformance with emerging industry standards, features described in this manual entry are likely to change in a future release.

DESCRIPTION

Both `cpacl()` and `fcpacl()` copy the access control list and mode bits (that is, file access permission bits and miscellaneous mode bits; see *chmod*(2)) from one file to another, and transfer ownership much like *chown*(2). `cpacl()` and `fcpacl()` take the following parameters:

- Path names (*fromfile* and *tofile*) or open file descriptors (*fromfd* and *tofd*).
- A mode value (*frommode*, typically the `st_mode` value returned by `stat()` – see *stat*(2)) containing file miscellaneous mode bits which are always copied, and file access permission bits which are copied instead of the access control list if either file is remote.
- User ID and group ID of the file (*fromuid*, *touid* and *fromgid*, *togid*) for transferring ownership. (Typically *fromuid* and *fromgid* are the `st_uid` and `st_gid` values returned by `stat()`, and *touid* and *togid* are the return values from `geteuid()` and `getegid()` – see *geteuid*(2) and *getegid*(2).)

When both files are local, the `cpacl()` routines copy the access control list and call `chownacl()` (see *chownacl*(3C)) to transfer ownership from the *fromfile* to the *tofile*, if necessary.

`cpacl()` (`fcpacl()`) handles remote copying (via NFS) after recognizing failures of `getacl()` (`fgetacl()`) or `setacl()` (`fsetacl()`) (see *setacl*(2)). When copying the mode from *fromfile* (*fromfd*) to *tofile* (*tofd*), `cpacl()` copies the entire *frommode* (that is, the file miscellaneous mode bits and the file access permission bits) to *tofile* (*tofd*) using `chmod()` (`fchmod()`). Some of the miscellaneous mode bits can be turned off; see *chmod*(2).

`cpacl()` (`fcpacl`) can copy an access control list from *fromfile* (*fromfd*) to *tofile* (*tofd*) without transferring ownership, but ensuring error checking and handling of remote files. This is done by passing *fromuid* equal to *touid* and *fromgid* equal to *togid* (that is, four zeros). For remote files, *fromuid*, *touid*, *fromgid*, and *togid* are ignored.

RETURN VALUE

If successful, `cpacl()` and `fcpacl()` return zero. If an error occurs, they set `errno` to indicate the cause of failure and return a negative value, as follows:

–1 Unable to perform `getacl()` (`fgetacl()`) on a local *fromfile* (*fromfd*).

–2 Unable to perform `chmod()` (`fchmod()`) on *tofile* (*tofd*) to set its file miscellaneous mode bits. `cpacl()` (`fcpacl()`) attempts this regardless of whether a file is local or remote, as long as *fromfile* (*fromfd*) is local.

–3 Unable to perform `setacl()` (`fsetacl()`) on a local *tofile* (*tofd*). As a consequence, the file's optional ACL entries are deleted, its file access permission bits are zeroed, and its miscellaneous mode bits might be altered.

–4 Unable to perform `chmod()` (`fchmod()`) on *tofile* (*tofd*) to set its mode. As a consequence, if *fromfile* (*fromfd*) is local, *tofile*'s (*tofd*'s) optional ACL entries are deleted, its access permission bits are zeroed, and its file miscellaneous mode bits might be altered, regardless of whether the file is local or remote.

EXAMPLES

The following code fragment gets *stat* information on `oldfile` and copies its file miscellaneous bits and access control list to `newfile` owned by the caller. If either file is remote, only the `st_mode` on `oldfile` is copied.

```
#include <sys/types.h>
#include <sys/stat.h>

struct stat statbuf;

if (stat ("oldfile", & statbuf) <
      error (...);

if (cpacl ("oldfile", newfile , statbuf.st_mode,
      statbuf.st_uid, statbuf.st_gid, geteuid(), getegid()) < 0)
{
      error (...);
}
```

AUTHOR

`cpacl()` and `fcpacl()` were developed by HP.

SEE ALSO

chown(2), getacl(2), getegid(2), geteuid(2), setacl(2), stat(2). acltostr(3C), chownacl(3C), setentry(3C), strtoacl(3C), acl(5).

NAME

crt0.o, gcrt0.o, mcrt0.o, frt0.o, gfrt0.o, mfrt0.o - execution startup routines

DESCRIPTION

The C and Pascal compilers link in files `crt0.o`, `gcrt0.o`, or `mcrt0.o` to provide startup capabilities and environment for program execution. All are identical except that `gcrt0.o` and `mcrt0.o` provide additional functionality for *gprof*(1) and *prof*(1) profiling support respectively. Similarly, the FORTRAN compiler links in either `frt0.o`, `gfrt0.o`, or `mfrt0.o`.

The following symbols are defined in these routines:

`__argc_value`	A variable of type *int* containing the number of arguments.
`__argv_value`	An array of character pointers to the arguments themselves.
`_environ`	An array of character pointers to the environment in which the program will run. This array is terminated by a null pointer.
`_SYSTEM_ID`	A variable of type int containing the system id value for an executable program.

DEPENDENCIES

Series 300/400

The symbols above are shown as they are visible from C. To access them from assembly language, add an additional underscore to the beginning of the symbol. For example, an assembly language program refers to `__argc_value` as `___argc_value`.

Series 300/400 startup files also define the following symbols which are listed as when used from assembly language. The state of these variables can be determined from C by using other library routines (see *is_hw_present*(3C)).

`flag_68010`	A variable of type `short`. Non-zero if the processor is a 68010; zero if not.
`float_soft`	A variable of type `short`. Zero if the HP 98635 floating-point card is present; non-zero if it is not present.
`float_loc`	A constant defining the location in memory of the HP 98635 floating-point card.
`flag_68881`	A variable of type `short`. Non-zero if the HP 68881 floating-point coprocessor is present; zero if it is not present.
`flag_fpa`	A variable of type `short`. Non-zero if the HP 98248 floating-point card is present; zero if it is not present.
`fpa_loc`	A constant defining the location in memory of the HP 98248 floating-point card.

Series 700/800

All compilers on Series 700 and 800 use the `crt0.o`, `gcrt0.o`, or `mcrt0.o` file; the files `frt0.o`, `gfrt0.o`, and `mfrt0.o` do not exist.

The Series 700 and 800 start-up files also define the following additional symbols:

`$START$`	Execution start address.
`_start`	A secondary startup routine for C programs, called from `$START$`, which in turn calls `main`. This routine is contained in the C library rather than the `crt0.o` file. For Pascal and FORTRAN programs, this symbol labels the beginning of the outer block (main program) and is generated by the compilers.
`$global$`	The initial address of the program's data pointer. The startup code loads this address into general register 27.
`$UNWIND_START`	The beginning of the stack unwind table.
`$UNWIND_END`	The end of the stack unwind table.
`$RECOVER_START`	The beginning of the try/recover table.
`$RECOVER_END`	The end of the try/recover table.

The `crt0.o` file defines a null procedure for `_mcount`, so programs compiled with profiling can be linked without profiling.

The linker defines the following two symbols:

`__text_start`	The beginning address of the program's text area.
`__data_start`	The beginning address of the program's data area.

AUTHOR

The features described in this entry originated from AT&T UNIX System III.

SEE ALSO

cc(1), f77(1), ld(1), pc(1), prof(1), gprof(1), pc(1), profil(2), exec(2), is_hw_present(3C), monitor(3C).

NAME

crypt, setkey, encrypt - generate hashing encryption

SYNOPSIS

```
#include <unistd.h>

char *crypt(const char *key, const char *salt);

void setkey(const char *key);

void encrypt(char block[64], int edflag);
```

DESCRIPTION

crypt():

`crypt()` is the password encryption function. It is based on a one way hashing encryption algorithm with variations intended (among other things) to frustrate use of hardware implementations of a key search.

key is a user's typed password. *salt* is a two-character string chosen from the set `[a-zA-Z0-9./]`; this string is used to perturb the hashing algorithm in one of 4096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password. The first two characters are the salt itself.

setkey() and encrypt():

The version of the `encrypt()` function currently shipped on standard HP-UX systems fails when `edflag` is non-zero (for decryption) and `errno` is set to ENOSYS in order to comply with industry standards and U.S. government regulations. However, fully functional versions are available from HP in certain geographic areas, and behave as described below:

`setkey()` and `encrypt()` provide (rather primitive) access to the actual hashing algorithm. The argument to `setkey()` is a character array of length 64 containing only the characters with numerical value 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored; this gives a 56-bit key which is set into the machine. This is the key that is used with the hashing algorithm to encrypt or decrypt the string *block* with the function `encrypt()`.

The *block* argument to `encrypt()` is a character array of length 64 containing only the characters with numerical value 0 and 1. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the hashing algorithm using the key that was set by `setkey()`. If *edflag* is zero, the argument is encrypted; if non-zero it is decrypted.

SEE ALSO

crypt(1), login(1), passwd(1), getpass(3C), passwd(4).

WARNINGS

The return value points to static data whose content is overwritten by each call.

STANDARDS CONFORMANCE

`crypt()`: SVID2, XPG2, XPG3, XPG4

`encrypt()`: SVID2, XPG2, XPG3, XPG4

`setkey()`: SVID2, XPG2, XPG3, XPG4

NAME

ctermid() - generate file name for terminal

SYNOPSIS

```
#include <stdio.h>

char *ctermid(char *s);
```

DESCRIPTION

`ctermid()` generates a string that, when used as a pathname, refers to the the controlling terminal for the current process.

If *s* is a NULL pointer, the string is stored in an internal static area, the contents of which are overwritten at the next call to `ctermid()`, and the address of which is returned. Otherwise, *s* is assumed to point to a character array of at least `L_ctermid` elements; the path name is placed in this array and the value of *s* is returned. The constant `L_ctermid` is defined in the `<stdio.h>` header file.

If the process has no controlling terminal, the pathname for the controlling terminal cannot be determined, or some other error occurs, `ctermid()` returns an empty string.

NOTES

The difference between `ctermid()` and `ttyname()` is that `ttyname()` must be handed a file descriptor and returns the actual name of the terminal associated with that file descriptor, while `ctermid()` returns a string (`/dev/tty`) that refers to the terminal if used as a file name. (see *ttyname*(3C)). Thus `ttyname()` is useful only if the process already has at least one file open to a terminal.

SEE ALSO

ttyname(3C).

STANDARDS CONFORMANCE

`ctermid()`: SVID2, XPG2, XPG3, POSIX.1, FIPS 151-1

NAME

ctime(), localtime(), gmtime(), mktime(), difftime(), asctime(), timezone(), daylight(), tzname(), tzset(), nl_ctime(), nl_cxtime(), nl_asctime(), nl_ascxtime() - convert date and time to string

SYNOPSIS

```
#include <time.h>

char *asctime(const struct tm *timeptr);

char *ctime(const time_t *timer);

double difftime(time_t time1, time_t time0);

struct tm *gmtime(const time_t *timer);

struct tm *localtime(const time_t *timer);

time_t mktime(struct tm *timeptr);

extern long timezone;

extern int daylight;

extern char *tzname[2];

void tzset(void);

char *nl_asctime(const struct tm *timeptr, const char *format, int langid);

char *nl_ascxtime(const struct tm *timeptr, const char *format);

char *nl_ctime(const time_t *timer, const char *format, int langid);

char *nl_cxtime(const time_t *timer, const char *format);
```

DESCRIPTION

asctime() Convert the broken-down time contained in the structure pointed to by *timeptr* and return a pointer to a 26-character string in the form:

```
Sun Sep 16 01:03:52 1973\n\0
```

All the fields have constant width.

ctime() Convert the calendar time pointed to by *timer*, representing the time in seconds since the Epoch, and return a pointer to the local time in the form of a string. Equivalent to:

```
asctime(localtime(timer))
```

gmtime() Convert directly to Coordinated Universal Time (UTC), the time standard used by the HP-UX operating system. **gmtime()** returns a pointer to the **tm** structure described below.

localtime() Correct for the time zone and any summer time zone adjustments (such as Daylight Savings Time in the USA), according to the contents of the TZ environment variable (see Environment Variables below). **localtime()** returns a pointer to the **tm** structure described below.

difftime() Return the difference in seconds between two calendar times: *time1* – *time0*.

mktime() Convert the broken-down time (expressed as local time) in the structure pointed to by *timeptr* into a calendar time value with the same encoding as that of the values returned by *time*(2). The original values of the **tm_wday** and **tm_yday** components of the structure are ignored, and the original values of the other components are not restricted to the ranges indicated below.

A positive or zero value for **tm_isdst** causes **mktime()** to initially presume that Daylight Saving Time respectively is or is not in effect for the specified time. A negative value for **tm_isdst** causes **mktime()** to attempt to determine whether Daylight Saving Time is in effect for the specified time.

Upon successful completion, all the components are set to represent the specified calendar time, but with their values forced to the ranges indicated below. The final value of

`tm_mday` is not set until `tm_mon` and `tm_year` are determined. `mktime()` returns the specified calendar time encoded as a value of type `time_t`.

If the calendar time cannot be represented, the function returns the value (`time_t`)—1 and sets `errno` to **ERANGE**. Note the value (`time_t`)—1 also corresponds to the time 23:59:59 on Dec 31, 1969 (plus or minus time zone and Daylight Saving Time adjustments). Thus it is necessary to check both the return value and `errno` to reliably detect an error condition.

`tzset()` Sets the values of the external variables *timezone*, *daylight*, and *tzname* according to the contents of the `TZ` environment variable (independent of any time value). The functions `localtime()`, `mktime()`, `ctime()`, `nl_ctime()`, `nl_cxtime()`, `asctime()`, `nl_asctime()`, `nl_ascxtime()`, and `strftime()` (see *strftime*(3C)) call `tzset()` and use the values returned in the external variables described below for their operations. `tzset()` can also be called directly by the user.

The <`time.h`> header file contains declarations of all relevant functions and externals. It also contains the `tm` structure, which includes the following members:

```
int tm_sec;     /* seconds after the minute - [0,61] */
int tm_min;     /* minutes after the hour - [0,59] */
int tm_hour;    /* hours - [0,23] */
int tm_mday;    /* day of month - [1,31] */
int tm_mon;     /* month of year - [0,11] */
int tm_year;    /* years since 1900 */
int tm_wday;    /* days since Sunday - [0,6] */
int tm_yday;    /* days since January 1 - [0,365] */
int tm_isdst;   /* daylight savings time flag */
```

The value of `tm_isdst` is positive if a summer time zone adjustment such as Daylight Savings Time is in effect, zero if not in effect, and negative if the information is not available.

The external variable `timezone` contains the difference, in seconds, between UTC and local standard time (for example, in the U.S. Eastern time zone (EST), `timezone` is 5*60*60). The external variable `daylight` is non-zero only if a summer time zone adjustment is specified in the TZ environment variable. The external variable `tzname[2]` contains the local standard and local summer time zone abbreviations as specified by the `TZ` environment variable.

EXTERNAL INFLUENCES

Locale

The `LC_TIME` category determines for the functions `nl_cxtime()`, `nl_ctime()`, `nl_ascxtime()`, and `nl_asctime()` the characters to be substituted for the directives described in *strftime*(3C) as being from the locale. It also determines the default output format used when a null format string is supplied to these functions.

The `LC_CTYPE` category determines the interpretation of the bytes within *format* as single and/or multibyte characters.

Environment Variables

The `tzset()` function uses the contents of `TZ` to set the values of the external variables `timezone`, `daylight` , and `tzname`. `TZ` also determines the time zone name substituted for the `%Z` and `%z` directives and the time zone adjustments performed by `localtime()`, `mktime()`, `ctime()`, `nl_ctime()`, and `nl_cxtime()`. Two methods for specifying a time zone within `TZ` are described in *environ*(5).

International Code Set Support

Single- and multi-byte character code sets are supported.

WARNINGS

Return values point to static data whose contents is overwritten by each call.

The range of `tm_sec([0,61])` extends to 61 to allow for the occasional one or two leap seconds. However, the "seconds since the Epoch" value returned by *time*(2) and passed as the *timer* argument does not include accumulated leap seconds. The `tm` structure generated by `localtime()` and `gmtime()` will never reflect any leap seconds. Upon successful completion, `mktime()` forces the value of the `tm_sec`

component to the range [0,59].

`ctime()`, `nl_cxtime()`, `nl_ctime()`, `asctime()`, `nl_ascxtime()`, and `nl_asctime()` are considered obsolescent and may be removed in a future release. Use of *strftime*(3C) is recommended in their stead.

AUTHOR

`ctime()` was developed by AT&T and HP.

SEE ALSO

time(2), nl_init(3C), getdate(3C), setlocale(3C), strftime(3C), tztab(4), environ(5), hpnls(5), lang(5), langinfo(5).

STANDARDS CONFORMANCE

`ctime()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
`asctime()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`daylight`: AES, SVID2, XPG2, XPG3, XPG4
`difftime()`: AES, XPG4, ANSI C

`gmtime()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
`localtime()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`mktime()`: AES, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
`nl_ascxtime()`: XPG2

`nl_cxtime()`: XPG2
`timezone`: AES, XPG2, XPG3, XPG4

`tzname`: AES, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1
`tzset()`: AES, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

isalpha(), isupper(), islower(), isdigit(), isxdigit(), isalnum(), isspace(), ispunct(), isprint(), isgraph(), iscntrl(), isascii() - classify characters

SYNOPSIS

```
#include <ctype.h>

int isalnum(int c);
int isalpha(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
int isascii(int c);
```

DESCRIPTION

These functions classify character-coded integer values according to the rules of the coded character set identified by the last successful call to `setlocale()` (see *setlocale*(3C)). Each function is a predicate returning non-zero for true, zero for false.

If `setlocale()` has not been called successfully, characters are classified according to the rules of the default ASCII 7-bit coded character set (see *setlocale*(3C)).

`isascii()` is defined on all integer values; the other functions are defined for the range `-1` (EOF) through `255`.

The functions return non-zero under the following circumstances; zero otherwise:

`isalpha(c)`	*c* is a letter.
`isupper(c)`	*c* is an uppercase letter.
`islower(c)`	*c* is a lowercase letter.
`isdigit(c)`	*c* is a decimal digit (in ASCII: characters [0-9]).
`isxdigit(c)`	*c* is a hexadecimal digit (in ASCII: characters [0-9], [A-F] or [a-f]).
`isalnum(c)`	*c* is an alphanumeric (letters or digits).
`isspace(c)`	*c* is a character that creates "white space" in displayed text (in ASCII: space, tab, carriage return, new-line, vertical tab, and form-feed).
`ispunct(c)`	*c* is a punctuation character (in ASCII: any printing character except the space character (040), digits, letters).
`isprint(c)`	*c* is a printing character.
`isgraph(c)`	*c* is a visible character (in ASCII: printing characters, excluding the space character (040)).
`iscntrl(c)`	*c* is a control character (in ASCII: character codes less than 040 and the delete character (0177)).
`isascii(c)`	*c* is any ASCII character code between 0 and 0177, inclusive.

If the argument to any of these functions is outside the domain of the function, the result is undefined.

EXTERNAL INFLUENCES

Locale

The `LC_CTYPE` category determines the classification of character type.

International Code Set Support

Single-byte character code sets are supported.

WARNINGS

These functions are supplied both as library functions and as macros defined in the <ctype.h> header. Normally, the macro versions are used. To obtain the library function, either use a **#undef** to remove the macro definition or, if compiling in ANSI-C mode, enclose the function name in parenthesis or take its address. The following example uses the library functions for **isalpha()**, **isdigit()**, and **isspace()**:

```
#include <ctype.h>
#undef isalpha
        ...
main()
{
    int (*ctype_func)();
          ...
    if ( isalpha(c) )
          ...
    if ( (isdigit)(c) )
          ...
    ctype_func = isspace;
          ...
}
```

AUTHOR

ctype() was developed by AT&T and HP.

SEE ALSO

setlocale(3C), ascii(5).

STANDARDS CONFORMANCE

isalnum(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
isalpha(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

isascii(): AES, SVID2, XPG2, XPG3, XPG4
iscntrl(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

isdigit(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
isgraph(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

islower(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
isprint(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

ispunct(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
isspace(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

isupper(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
isxdigit(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

curses() - CRT screen handling and optimization package

SYNOPSIS

```
#include <curses.h>
```

cc [*flags*] *file* ... `-lcurses` [*libraries*]

DESCRIPTION

These routines provide a method for updating screens with reasonable optimization. To initialize `curses` routines, `initscr()` must be called before calling any other routine that deals with windows and screens. `endwin()` should be called before exiting. To get character-at-a-time input without echoing, (most interactive, screen oriented-programs need this) after calling `initscr()` the program should call:

```
nonl(); cbreak(); noecho();
```

The full `curses` interface permits manipulation of data structures called "windows", which can be thought of as two-dimensional arrays of characters representing all or part of a CRT screen. A default window called `stdscr` is supplied, and others can be created using `newwin`. Windows are referred to by variables declared `WINDOW *`, the type `WINDOW` is defined in <`curses.h`> to be a C structure. These data structures are manipulated by using functions described below, among which the most basic are `move()`, and `addch()`. (More general versions of these functions are included. Their names begin with `w`, allowing the programmer to specify a window. Routines not beginning with `w` affect `stdscr`.) Then `refresh()` is called, telling the routines to make the user's CRT screen resemble `stdscr`.

Mini-Curses is a subset of curses which does not allow manipulation of more than one window. To invoke this subset, use `-DMINICURSES` as an option to the *cc*(1) command. This level is smaller and faster than full curses.

If the environment variable `TERMINFO` is defined, any program using curses checks for a local terminal definition before checking in the standard place. For example, if the standard place is `/usr/lib/terminfo`, and `TERM` is set to `vt100`, the compiled file is normally found in `/usr/lib/terminfo/v/vt100` (the `v` is copied from the first letter of `vt100` to avoid creation of huge directories). However, if `TERMINFO` is set to `/usr/mark/myterms`, `curses` first checks `/usr/mark/myterms/v/vt100`, then, if that fails, checks `/usr/lib/terminfo/v/vt100`. This is useful for developing experimental definitions, or when write permission in `/usr/lib/terminfo` is not available.

Functions

All routines listed here can be called when using the full curses. Those marked with an asterisk can be called when using Mini-Curses.

`addch(`*ch*`)*`	Add a character to *stdscr* (similar to `putchar()`; wraps to next line at end of line).
`addstr(`*str*`)*`	Call `addch()` with each character in *str*
`attroff(`*attrs*`)*`	Turn off attributes named
`attron(`*attrs*`)*`	Turn on attributes named
`attrset(`*attrs*`)*`	Set current attributes to *attrs*
`baudrate()*`	Current terminal speed
`beep()*`	Sound beep on terminal
`box(`*win, vert, hor*`)`	Draw a box around edges of *win*. *vert* and *hor* are chars to use for vertical and horizontal edges of box
`clear()`	Clear *stdscr*
`clearok(`*win, bf*`)`	Clear screen before next redraw of *win*
`clrtobot()`	Clear to bottom of *stdscr*
`clrtoeol()`	Clear to end of line on *stdscr*
`cbreak()*`	Set cbreak mode
`delay_output(`*ms*`)*`	Insert *ms* millisecond pause in output
`delch()`	Delete a character
`deleteln()`	Delete a line
`delwin(`*win*`)`	Delete *win*
`doupdate()`	Update screen from all `wnooutrefresh()`
`echo()*`	Set echo mode

endwin()*	End window modes
erase()	Erase *stdscr*
erasechar()	Return user's erase character
fixterm()	Restore tty to "in-curses" state
flash()	Flash screen or beep
flushinp()*	Throw away any type-ahead characters
getch()	Get a char from tty
getstr(*str*)	Get a string through *stdscr*
gettmode()	Establish current tty modes
getyx(*win*, *y*, *x*)	Get (*y*, *x*) co-ordinates
has_ic()	True if terminal can do insert character
has_il()	True if terminal can do insert line
idlok(*win*, *bf*)*	Use terminal's insert/delete line if *bf* != 0
inch()	Get char at current (*y*, *x*) co-ordinates
initscr()*	Initialize screens
insch(*c*)	Insert a char
insertln()	Insert a line
intrflush(*win*, *bf*)	Interrupts flush output if *bf* is TRUE
keypad(*win*, *bf*)	Enable keypad input
killchar()	Return current user's kill character
leaveok(*win*, *flag*)	Permissible to leave cursor anywhere after refresh if *flag* !=0 for *win*; otherwise cursor must be left at current position.
longname()	Return verbose name of terminal
meta(*win*, *flag*)*	Allow meta characters on input if *flag* != 0
move(*y*, *x*)*	move to (*y*, *x*) on *stdscr*
mvaddch(*y*, *x*, *ch*)	**move**(*y*, *x*) then **addch**(*ch*)
mvaddstr(*y*, *x*, *str*)	Similar...
mvcur(*oldrow*, *oldcol*, *newrow*, *newcol*)	Low-level cursor motion
mvdelch(*y*, *x*)	Similar to **delch**(), but **move**(*y*, *x*) first
mvgetch(*y*, *x*)	etc.
mvgetstr(*y*, *x*)	
mvinch(*y*, *x*)	
mvinsch(*y*, *x*, *c*)	
mvprintw(*y*, *x*, *fmt*, *args*)	
mvscanw(*y*, *x*, *fmt*, *args*)	
mvwaddch(*win*, *y*, *x*, *ch*)	
mvwaddstr(*win*, *y*, *x*, *str*)	
mvwdelch(*win*, *y*,, *x*)	
mvwgetch(*win*, *y*,, *x*)	
mvwgetstr(*win*, *y*,, *x*)	
mvwin(*win*, *by*,, *bx*)	
mvwinch(*win*, *y*,, *x*)	
mvwinsch(*win*, *y*, *x*, *c*)	
mvwprintw(*win*, *y*, *x*, *fmt*, *args*)	
mvwscanw(*win*, *y*, *x*, *fmt*, *args*)	
newpad(*nlines*, *ncols*)	Create a new pad with given dimensions
newterm(*type*, *outfd*, *infd*)	Set up new terminal of given type to output on *outfd*, using input (if needed) from *infd*
newwin(*lines*, *cols*, *begin_y*, *begin_x*)	Create a new window
nl()*	Set new-line mapping
nocbreak()*	Unset cbreak mode
nodelay(*win*, *bf*)	Enable nodelay input mode through **getch**()
noecho()*	Unset echo mode
nonl()*	Unset new-line mapping
noraw()*	Unset raw mode
overlay(*win1*, *win2*)	Overlay *win1* on *win2*

overwrite(*win1*, *win2*)	Overwrite *win1* on *win2*
pnoutrefresh(*pad*, *pminrow*, *pmincol*, *sminrow*, *smincol*, *smaxrow*, *smaxcol*)	Similar to **prefresh**() but with no output until **doupdate**() called
prefresh(*pad*, *pminrow*, *pmincol*, *sminrow*, *smincol*, *smaxrow*, *smaxcol*)	Refresh from pad starting with given upper left corner of pad with output to given portion of screen
printw(*fmt*, *arg1*, *arg2*, ...)	**printf**() on *stdscr*
raw()*	Set raw mode
refresh()*	Make current screen look like *stdscr*
resetterm()*	Set tty modes to "out of curses" state
resetty()*	Reset tty flags to stored value
saveterm()*	Save current modes as "in curses" state
savetty()*	Store current tty flags
scanw(*fmt*, *arg1*, *arg2*, ...)	**scanf**() through *stdscr*
scroll(*win*)	Scroll *win* one line
scrollok(*win*, *flag*)	Allow terminal to scroll if *flag* != 0
set_term(*new*)	Switch to terminal *new*
setscrreg(*t*, *b*)	set user scrolling region to lines t through b
setterm(*type*)	Establish terminal with given type
setupterm(*term*, *filenum*, *errret*)	
standend()*	Clear standout mode attribute
standout()*	Set standout mode attribute
subwin(*win*, *lines*, *cols*, *begin_y*, *begin_x*)	create a subwindow
touchwin(*win*)	Change all of *win*
traceoff()	Turn off debugging trace output
traceon()	Turn on debugging trace output
typeahead(*fd*)	Use file descriptor *fd* to check type-ahead
unctrl(*ch*)*	Printable version of *ch*
waddch(*win*, *ch*)	Add char to *win*
waddstr(*win*, *str*)	Add string to *win*
wattroff(*win*, *attrs*)	Turn off *attrs* in *win*
wattron(*win*, *attrs*)	Turn on *attrs* in *win*
wattrset(*win*, *attrs*)	Set attrs in *win* to *attrs*
wclear(*win*)	Clear *win*
wclrtobot(*win*)	Clear to bottom of *win*
wclrtoeol(*win*)	Clear to end of line on *win*
wdelch(*win*, *c*)	Delete char from *win*
wdeleteln(*win*)	Delete line from *win*
werase(*win*)	Erase *win*
wgetch(*win*)	Get a char through *win*
wgetstr(*win*, *str*)	Get a string through *win*
winch(*win*)	Get char at current (*y*, *x*) in *win*
winsch(*win*, *c*)	Insert char into *win*
winsertln(*win*)	Insert line into *win*
wmove(*win*, *y*, *x*)	Set current (*y*, *x*) co-ordinates on *win*
wnoutrefresh(*win*)	Refresh but no screen output
wprintw(*win*, *fmt*, *arg1*, *arg2*, ...)	**printf**() on *win*
wrefresh(*win*)	Make screen look like *win*
wscanw(*win*, *fmt*, *arg1*, *arg2*, ...)	**scanf**() through *win*
wsetscrreg(*win*, *t*, *b*)	Set scrolling region of *win*
wstandend(*win*)	Clear standout attribute in *win*
wstandout(*win*)	Set standout attribute in *win*

Terminfo Level Routines

These routines should be called by programs that need to deal directly with the *terminfo*(4) database. Due to the low level of this interface, its use is discouraged. Initially, **setupterm**() should be called to define

the set of terminal-dependent variables defined in *terminfo*(4). The header files <curses.h> and <term.h> should be included to get the definitions for these strings, numbers, and flags. Parameterized strings should be passed through **tparm()** to instantiate them. All *terminfo*(4) strings (including the output of **tparm()**) should be printed with **tputs()** or **putp()**. Before exiting, **resetterm()** should be called to restore the tty modes. (Programs desiring shell escapes or suspending with control-Z can call **resetterm()** before the shell is called and **fixterm()** after returning from the shell.)

fixterm()	Restore tty modes for terminfo use (called by **setupterm()**)
resetterm()	Reset tty modes to state before program entry
setupterm(*term, fd, rc***)**	Read in database. Terminal type is the character string *term*, all output is to HP-UX System file descriptor *fd*. A status value is returned in the integer pointed to by *rc*: 1 is normal. The simplest call would be **setupterm(0, 1, 0)** which uses all defaults.
tparm(*str, p1, p2, ..., p9***)**	Instantiate string str with parms p_i.
tputs(*str, affcnt, putc***)**	Apply padding info to string *str*. *affcnt* is the number of lines affected, or 1 if not applicable. **putc()** is a putchar-like function to which the characters are passed, one at a time.
putp(*str***)**	A handy function that calls **tputs(***str***, 1, *putchar*)**.
vidputs(*attrs, putc***)**	output the string to put terminal in video attribute mode *attrs*, which is any combination of the attributes listed below. Chars are passed to putchar-like function **putc()**.
vidattr(*attrs***)**	Like vidputs but outputs through putchar
set_curterm(*term***)**	Set the database pointed to by term
del_curterm(*term***)**	Free the space pointed to by term

Termcap Compatibility Routines

These routines were included as a conversion aid for programs that use termcap. Calling parameters are the same as for termcap. They are emulated using the *terminfo*(4) database. Their use in new software is not recommended because they might be deleted in future HP-UX releases.

tgetent(*bp, name***)**	look up termcap entry for name
tgetflag(*id***)**	get boolean entry for id
tgetnum(*id***)**	get numeric entry for id
tgetstr(*id, area***)**	get string entry for id
tgoto(*cap, col, row***)**	apply parms to given cap
tputs(*cap, affcnt, fn***)**	apply padding to cap calling fn as putchar

Attributes

The following video attributes can be passed to the functions **attron()**, **attroff()**, and **attrset()**.

A_STANDOUT	Terminal's best highlighting mode
A_UNDERLINE	Underlining
A_REVERSE	Reverse video
A_BLINK	Blinking
A_DIM	Half bright
A_BOLD	Extra bright or bold
A_BLANK	Blanking (invisible)
A_PROTECT	Protected
A_ALTCHARSET	Alternate character set

NLS Attributes

The following NLS attributes might be returned by **inch()**:

A_FIRSTOF2	First byte of 16-bit character
A_SECOF2	Second byte of 16-bit character

Function Keys

The following function keys could possibly be returned by *getch* if *keypad* has been enabled. Note that not all of these are currently supported due to lack of definitions in *terminfo* or the terminal not transmitting a

unique code when the key is pressed.

Name	*Value*	*Key name*
KEY_BREAK	0401	break key (unreliable)
KEY_DOWN	0402	The four arrow keys ...
KEY_UP	0403	
KEY_LEFT	0404	
KEY_RIGHT	0405	
KEY_HOME	0406	Home key (upward+left arrow)
KEY_BACKSPACE	0407	backspace (unreliable)
KEY_F0	0410	Function keys. Space reserved for up to 64 keys.
KEY_F(n)	(KEY_F0+(n))	Formula for fn.
KEY_DL	0510	Delete line
KEY_IL	0511	Insert line
KEY_DC	0512	Delete character
KEY_IC	0513	Insert char or enter insert mode
KEY_EIC	0514	Exit insert char mode
KEY_CLEAR	0515	Clear screen
KEY_EOS	0516	Clear to end of screen
KEY_EOL	0517	Clear to end of line
KEY_SF	0520	Scroll 1 line forward
KEY_SR	0521	Scroll 1 line backwards (reverse)
KEY_NPAGE	0522	Next page
KEY_PPAGE	0523	Previous page
KEY_STAB	0524	Set tab
KEY_CTAB	0525	Clear tab
KEY_CATAB	0526	Clear all tabs
KEY_ENTER	0527	Enter or send (unreliable)
KEY_SRESET	0530	soft (partial) reset (unreliable)
KEY_RESET	0531	reset or hard reset (unreliable)
KEY_PRINT	0532	print or copy
KEY_LL	0533	home down or bottom (lower left)

Window-Change Signal Support

All curses routines except the min-curses subset provide `SIGWINCH` support. Applications that are linked with curses routines immediately redraw the screen in response to window size changes. The environmental variables `LINES` and `COLUMNS` are also updated so that children processes work with the correct window size.

If there is a window size reduction, part of the application display is trimmed. The trimmed portion is saved in internal memory at the time of resize. Moreover, this portion is not affected by the application as long as it stays invisible. If the application's cursor is trimmed, unexpected behavior results.

On the other hand, if the window is enlarged, any previously trimmed area is re-displayed (and re-activated). If the window is enlarged beyond its initial size, the extra area is padded with blank spaces.

The default `SIGWINCH` support can be disabled by installing a custom `SIGWINCH` signal handler via the sigvector command (see *sigvector*(2)).

WARNINGS

HP supports only terminals listed on the current list of HP-supported devices. However, the *terminfo*(4) database may contain information for other terminals besides those that are officially supported. If you use such unsupported terminals, they may not work correctly.

The `endwin()` routine does not release memory allocated by the `initscr()` routine.
Repeated calls to `initscr()` can cause a program to use more memory than was intended.

Some of these routines call `malloc()` to allocate memory (see *malloc*(3C)) and can therefore fail for any of the reasons described in the *malloc*(3C) manual entry.

SEE ALSO

sigvector(2), terminfo(4).

Using Curses and Terminfo, tutorial in *Terminal Control User's Guide.*

STANDARDS CONFORMANCE

`curses()`: SVID2, XPG2, XPG3, XPG4

NAME

cuserid() - get character login name of the user

SYNOPSIS

```
#include <stdio.h>

char *cuserid(char *s);
```

Remarks:

Because this function behaved differently in previous releases of HP-UX, and behaves differently on other systems, its use is not recommended. It is provided only for conformance to current industry standards, and is subject to withdrawal in future releases of HP-UX.

For portability and security, application writers and maintainers should search their existing code and replace references to `cuserid()` with equivalent calls to `getpwuid(getuid())`, `getpwuid(geteuid())`, or `getlogin()`, depending on which user name is desired.

DESCRIPTION

`cuserid()` generates a character-string representation of the user name corresponding to the effective user ID of the process. If *s* is a NULL pointer, this representation is generated in an internal static area, the address of which is returned. Otherwise, *s* is assumed to point to an array of at least `L_cuserid` characters; the representation is left in this array. The constant `L_cuserid` is defined in the `<stdio.h>` header file.

DIAGNOSTICS

If the login name cannot be found, `cuserid()` returns a NULL pointer; if *s* is not a NULL pointer, a null character (`\0`) is placed at *s*[0].

SEE ALSO

geteuid(2), getlogin(3C) getpwuid(3C).

STANDARDS CONFORMANCE

`cuserid()`: AES, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

cvtnum() - convert string to floating point number

SYNOPSIS

```
#include <cvtnum.h>

int cvtnum(
    const unsigned char *src,
    unsigned char *dst,
    int typ,
    int rnd,
    unsigned char **ptr,
    int *inx
);
```

DESCRIPTION

cvtnum() converts an ASCII character string to a number in one of four floating-point formats: single precision, double precision, extended precision, or packed decimal string.

The string pointed to by *src* is the string representation of a standard number, an infinity, or a not-a-number. A standard number begins with an optional plus or minus sign followed by a string of digits optionally containing a decimal point. It can then have an optional **e** or **E** followed by an optional sign followed by an integer. Infinities are represented by **INF** preceded by an optional plus or minus sign. The string for a not-a-number is an optional sign followed by **NaN** followed by any number of hexadecimal digits enclosed in parentheses.

The result is moved to *dst* and will be of the size and format as defined for the MC68881 floating-point coprocessor.

typ indicates the type of conversion to be done. It may be one of four values: **C_SNGL**, **C_DBLE**, **C_EXT**, or **C_DPACK**, indicating single precision, double precision, extended precision and packed decimal string, respectively.

rnd specifies the type of rounding mode and can be one of four values: **C_NEAR**, **C_POS_INF**, **C_NEG_INF**, or **C_TOZERO**, indicating round to nearest, to positive infinity, to negative infinity, or to zero, respectively.

If the value of ***ptr* is not (char **)NULL, a pointer to the character terminating the scan is returned in the location pointed to by *ptr*. If no number can be formed, ***ptr* is set to *str*.

If *inx* is not (int *)NULL, **cvtnum()** uses this to return an indication of the inexactness of the conversion. A zero indicates exact; a non-zero value, inexact.

RETURN VALUE

If no errors occur or no non-standard conversions are done, **cvtnum()** returns 0. Otherwise, it returns one of the following:

[C_BADCHAR]	Illegal character or unexpected end of string
[C_OVER]	Overflow
[C_UNDER]	Underflow
[C_INF]	Infinity
[C_QNAN]	Quiet NaN
[C_SNAN]	Signalling NaN

cvtnum() does not use **errno** when reporting errors.

SEE ALSO

scanf(3S), strtod(3C), strtol(3C).
MC 68881 Floating-Point Coprocessor User's Manual.

NAME

datalock() - lock process into memory after allocating data and stack space

SYNOPSIS

```
#include <sys/lock.h>

int datalock(size_t datsiz, size_t stsiz);
```

DESCRIPTION

`datalock()` allocates at least *datsiz* bytes of data space and *stsiz* bytes of stack space, then locks the program in memory. The data space is allocated by `malloc()` (see *malloc*(3C)). After the program is locked, this space is released by `free()` (see *malloc*(3C)), making it available for use. This allows the calling program to use that much space dynamically without receiving the `SIGSEGV` signal.

The effective user ID of the calling process must be super-user or be a member of or have an effective group ID of a group having PRIV_MLOCK access to use this call (see *setprivgrp*(2)).

EXAMPLES

The following call to `datalock()` allocates 4096 bytes of data space and 2048 bytes of stack space, then locks the process in memory:

```
datalock (4096, 2048);
```

RETURN VALUE

`datalock()` returns -1 if `malloc()` cannot allocate enough memory or if `plock()` returned an error (see *plock*(2)).

WARNINGS

Multiple datalocks cannot be the same as one big one.

Methods for calculating the required size are not yet well developed.

AUTHOR

`datalock()` was developed by HP.

SEE ALSO

getprivgrp(2), plock(2).

NAME

dbminit, fetch, store, delete, firstkey, nextkey, dbmclose - database subroutines

SYNOPSIS

```
int dbmclose(void);
```

DESCRIPTION

These functions maintain key/content pairs in a database. They handle very large (a billion blocks (block = 1024 bytes)) databases and can locate a keyed item in one or two file system accesses. This package is superseded by the newer *ndbm*(3X) library, which can manage multiple databases. The functions can be accessed by giving the **-ldbm** option to *ld*(1) or *cc*(1).

key and *content* parameters are described by the **datum** type. A **datum** specifies a string of *dsize* bytes pointed to by *dptr*. Arbitrary binary data, as well as normal ASCII strings, are allowed. The database is stored in two files. One file is a directory containing a bit map of keys and has **.dir** as its suffix. The second file contains all data and has **.pag** as its suffix.

Before a database can be accessed, it must be opened by **dbminit**. At the time of this call, the files *file***.dir** and *file***.pag** must exist. (An empty database is created by creating zero-length **.dir** and **.pag** files.)

Once open, data stored under a key is accessed by *fetch*, and data is placed under a key by *store*. Storing data on an existing key replaces the existing data. A key (and its associated contents) is deleted by **delete**. A linear pass through all keys in a database can be made, in (apparently) random order by using **firstkey** and **nextkey**. **firstkey** returns the first key in the database. With any key, **nextkey** returns the next key in the database. The following code can be used to traverse the database:

```
for (key = firstkey(); key.dptr != NULL; key = nextkey(key))
```

A database can be closed by calling **dbmclose**. A currently open database must be closed before opening a new one.

DIAGNOSTICS

All functions that return an **int** indicate errors with negative values and success with zero. Functions that return a **datum** indicate errors with a null *dptr*.

WARNINGS

The *dbm* functions provided in this library should not be confused in any way with those of a general-purpose database management system such as ALLBASE/HP-UX SQL. These functions *do not* provide for multiple search keys per entry, they *do not* protect against multi-user access (in other words they do not lock records or files), and they *do not* provide the many other useful data base functions that are found in more robust database management systems. Creating and updating databases by use of these functions is relatively slow because of data copies that occur upon hash collisions. These functions *are useful* for applications requiring fast lookup of relatively static information that is to be indexed by a single key.

The **.pag** file will contain holes so that its apparent size is about four times its actual content. Some older UNIX systems create real file blocks for these holes when touched. These files cannot be copied by normal means (such as *cp*(1), *cat*(1), *tar*(1), or *ar*(1)) without expansion.

dptr pointers returned by these subroutines point into static storage that is changed by subsequent calls.

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 1024 bytes). Moreover, all key/content pairs that hash together must fit on a single block. **store** returns an error if a disk block fills with inseparable data.

delete does not physically reclaim file space, although it does make it available for reuse.

The order of keys presented by **firstkey** and **nextkey** depends on a hashing function, not on anything interesting.

A **store** or **delete** during a pass through the keys by **firstkey** and **nextkey** may yield unexpected results.

AUTHOR

dbm(3X) was developed by the University of California, Berkeley.

SEE ALSO

ndbm(3X).

NAME

devnm - map device ID to file path

SYNOPSIS

```
#include <devnm.h>

int devnm (
    mode_t devtype,
    dev_t devid,
    char *path,
    size_t pathlen,
    int cache
    );
```

DESCRIPTION

Given a device type, a device ID, and a string in which to return the result, **devnm()** maps the type and ID to a block or character special file (device file) name by searching **/dev**. It returns in *path* the full path name of the first special file encountered with a matching device type and ID. It searches **/dev** and all its subdirectories recursively in unspecified order.

The parameters are:

devtype One of the file type values **S_IFBLK** or **S_IFCHR** documented in *stat*(5). Bits other than those in the **S_IFMT** set are ignored. Hence the value can be, for example, an **st_mode** value returned by **stat()** (see *stat*(2)).

devid A device ID (major/minor) such as returned by **stat()** in the **st_dev** or **st_rdev** field.

path Pointer to the buffer in which to return the path name result.

pathlen Tells the available length of the *path* string, including the NUL terminator character. If *path* is too short to hold the full path name, only the first *pathlen*–1 characters are returned in a null-terminated string, and the return value is altered (see below).

cache A flag that tells **devnm()** whether to save file information in **malloc()**'d memory, and later, whether to use that saved information instead of searching **/dev** again. A subsequent call with *cache* non-zero can be much faster, especially if **/dev** is a large tree. However, the first call with *cache* true might be slower because **devnm()** must read all of the **/dev** tree once to create the cache, rather than returning immediately upon finding a matching file. Any call with *cache* set to zero ignores the cache, if any, and reads the directory.

To allow for possible future enhancements, *cache* should be restricted to the values 0 and 1.

There is no way to tell **devnm()** to free its cached memory.

devnm() ignores unreadable directories and files for which **stat()** fails.

devnm() does not examine alternate (hidden) elements of context-dependent files (CDFs).

RETURN VALUE

devnm() returns one of the following values:

0 Successful. The result is in *path*.

–1 **ftw()** failed. **errno** contains the value returned from **ftw()**. *path* might be altered if *cache* is set. If *cache* was set for the first time, **devnm()** freed any memory allocated by the current call.

–2 No matching special file was found. **errno** is undefined. *path* is unaltered.

–3 A matching special file was found, but the name was truncated to fit in *path*. **errno** is undefined.

If **malloc()** fails, **devnm()** silently abandons the attempt to do caching in the current or any later call with *cache* true, and frees any memory allocated by the current call.

AUTHOR

devnm( ) was created by HP.

SEE ALSO

devnm(1M), stat(2), ftw(3), malloc(3), ttyname(3), stat(5).

NAME

dial(), undial() - establish an out-going terminal line connection

SYNOPSIS

```
#include <dial.h>

int dial(CALL call);

void undial(int fd);
```

DESCRIPTION

`dial()` returns a file-descriptor for a terminal line open for read/write. The argument to `dial()` is a **CALL** structure (defined in the <`dial.h`> header file).

When finished with the terminal line, the calling program must invoke `undial()` to release the semaphore that has been set during the allocation of the terminal device.

The definition of **CALL** in the <`dial.h`> header file is:

```
typedef struct {
    struct termio    *attr;     /* pointer to termio attribute struct */
    int              baud;      /* transmission data rate */
    int              speed;     /* 212A modem: low=300, high=1200 */
    char             *line;     /* device name for out-going line */
    char             *telno;    /* pointer to tel-no digits string */
    int              modem;     /* specify modem control for direct lines */
    char             *device;   /* Will hold the name of the device used
                                   to make a connection */
    int              dev_len;   /* The length of the device used to
                                   make connection */
} CALL;
```

CALL elements are as follows:

speed Intended only for use with an outgoing dialed call, in which case its value should be either 300 or 1200 to identify the 113A modem, or the high- or low-speed setting on the 212A modem. Note that the 113A modem or the low-speed setting of the 212A modem transmits at any rate between 0 and 300 bits per second. However, the high-speed setting of the 212A modem transmits and receives at 1200 bits per second only.

baud Desired transmission baud rate. For example, one might set *baud* to 110 and *speed* to 300 (or 1200). However, if *speed* set to 1200 *baud* must be set to high (1200).

line If the desired terminal line is a direct line, a string pointer to its device-name should be placed in the *line* element in the **CALL** structure. Legal values for such terminal device names are kept in the `Devices` file. In this case, the value of the *baud* element need not be specified as it will be determined from the `Devices` file.

telno A pointer to a character string representing the telephone number to be dialed. Such numbers can consist only of symbols described below. The termination symbol is supplied by the `dial()` function, and should not be included in the *telno* string passed to `dial()` in the **CALL** structure.

Permissible codes

0-9	dial 0-9
* or :	dial *
# or ;	dial #
-	4-second delay for second dial tone
e or <	end-of-number
w or =	wait for secondary dial tone
f	flash off hook for 1 second

modem Specifies modem control for direct lines. Set to non-zero if modem control is required.

attr Pointer to a `termio` structure, as defined in the <`termio.h`> header file. A NULL value for this pointer element can be passed to the `dial()` function, but if such a

structure is included, the elements specified in it are set for the outgoing terminal line before the connection is established. This is often important for certain attributes such as parity and baud-rate.

device Holds the device name (cul..) that establishes the connection.

dev_len Length of the device name that is copied into the array device.

RETURN VALUE

On failure, a negative value indicating the reason for the failure is returned. Mnemonics for these negative indices as listed here are defined in the <dial.h> header file.

```
INTRPT    -1    /* interrupt occurred */
D_HUNG    -2    /* dialer hung (no return from write) */
NO_ANS    -3    /* no answer within 10 seconds */
ILL_BD    -4    /* illegal baud-rate */
A_PROB    -5    /* automatic call unit (acu) problem (open() failure) */
L_PROB    -6    /* line problem (open() failure) */
NO_Ldv    -7    /* can't open LDEVS file */
DV_NT_A   -8    /* requested device not available */
DV_NT_K   -9    /* requested device not known */
NO_BD_A   -10   /* no device available at requested baud */
NO_BD_K   -11   /* no device known at requested baud */
```

WARNINGS

Including the <dial.h> header file automatically includes the <termio.h> header file.

The above routine uses <stdio.h>, which causes unexpected increases in the size of programs that otherwise do not use standard I/O.

DEPENDENCIES

HP Clustered Environment

`dial()` is not supported on client nodes of an HP Cluster.

Series 300/400

An `alarm()` (see *alarm*(2)) system call for 3600 seconds is made (and caught) within the `dial()` module for the purpose of "touching" the `LCK..` file and constitutes the device allocation semaphore for the terminal device. Otherwise, *uucp*(1) may simply delete the `LCK..` entry on its 90-minute clean-up rounds. The alarm may go off while the user program is in a `read()` or `write()` system call, causing an apparent error return. If the user program expects to be around for an hour or more, error returns from reads should be checked for `(errno==EINTR)`, and the read possibly reissued.

FILES

`/usr/lib/uucp/Devices`
`/usr/spool/uucp/LCK..`*tty-device*

SEE ALSO

uucp(1), alarm(2), read(2), write(2), termio(7).

UUCP tutorial in *Remote Access User's Guide*.

NAME

opendir(), readdir(), telldir(), seekdir(), rewinddir(), closedir() - directory operations

SYNOPSIS

```
#include <dirent.h>

DIR *opendir(const char *dirname);

struct dirent *readdir(DIR *dirp);

long int telldir(DIR *dirp);

void seekdir(DIR *dirp, long int loc);

void rewinddir(DIR *dirp);

int closedir(DIR *dirp);
```

DESCRIPTION

This library package provides functions that allow programs to read directory entries without having to know the actual directory format associated with the file system. Because these functions allow programs to be used portably on file systems with different directory formats, this is the recommended way to read directory entries.

`opendir()` opens the directory *dirname* and associates a directory stream with it. `opendir()` returns a pointer used to identify the directory stream in subsequent operations. `opendir()` uses *malloc*(3C) to allocate memory.

`readdir()` returns a pointer to the next directory entry. It returns a NULL pointer upon reaching the end of the directory or detecting an invalid `seekdir()` operation. See *dirent*(5) for a description of the fields available in a directory entry.

`telldir()` returns the current location (encoded) associated with the directory stream to which *dirp* refers.

`seekdir()` sets the position of the next `readdir()` operation on the directory stream to which *dirp* refers. The *loc* argument is a location within the directory stream obtained from `telldir()`. The position of the directory stream is restored to where it was when `telldir()` returned that *loc* value. Values returned by `telldir()` are valid only while the `DIR` pointer from which they are derived remains open. If the directory stream is closed and then reopened, the `telldir()` value might be invalid.

`rewinddir()`
resets the position of the directory stream to which *dirp* refers to the beginning of the directory. It also causes the directory stream to refer to the current state of the corresponding directory, as a call to `opendir()` would have done.

`closedir()` closes the named directory stream, then frees the structure associated with the `DIR` pointer.

RETURN VALUE

`opendir()`, upon successful completion, returns a pointer to an object of type `DIR` referring to an open directory stream. Otherwise, it returns a NULL pointer and sets the global variable `errno` to indicate the error.

`readdir()`, upon successful completion, returns a pointer to an object of type `struct dirent` describing a directory entry. Upon reaching the end of the directory, `readdir()` returns a NULL pointer and does not change the value of `errno`. Otherwise, it returns a NULL pointer and sets `errno` to indicate the error.

`telldir()`, upon successful completion, returns a long value indicating the current position in the directory. Otherwise it returns `-1` and sets `errno` to indicate the error.

`seekdir()` does not return any value, but if an error is encountered, `errno` is set to indicate the error.

`closedir()`, upon successful completion, returns a value of `0`. Otherwise, it returns a value of `-1` and sets `errno` to indicate the error.

ERRORS

`opendir()` fails if any of the following conditions are encountered:

[EACCES]	Search permission is denied for a component of *dirname*, or read permission is denied for *dirname*.
[EFAULT]	*dirname* points outside the allocated address space of the process. The reliable detection of this error is implementation dependent.
[ELOOP]	Too many symbolic links were encountered in translating the path name.
[EMFILE]	Too many open file descriptors are currently open for the calling process.
[ENAMETOOLONG]	A component of *dirname* exceeds **PATH_MAX** bytes, or the entire length of *dirname* exceeds **PATH_MAX** – 1 bytes while **_POSIX_NO_TRUNC** is in effect.
[ENFILE]	Too many open file descriptors are currently open on the system.
[ENOENT]	A component of *dirname* does not exist.
[ENOMEM]	`malloc()` failed to provide sufficient memory to process the directory.
[ENOTDIR]	A component of *dirname* is not a directory.
[ENOENT]	The *dirname* argument points to an empty string.

`readdir()` might fail if any of the following conditions are encountered:

[EBADF]	*dirp* does not refer to an open directory stream.
[ENOENT]	The directory stream to which *dirp* refers is not located at a valid directory entry.
[EFAULT]	*dirp* points outside the allocated address space of the process.

`telldir()` might fail if any of the following conditions are encountered:

[EBADF]	*dirp* does not refer to an open directory stream.
[ENOENT]	*dirp* specifies an improper file system block size.

`seekdir()` might fail if the following condition is encountered:

[ENOENT]	*dirp* specifies an improper file system block size.

`closedir()` might fail if any of the following conditions are encountered:

[EBADF]	*dirp* does not refer to an open directory stream.
[EFAULT]	*dirp* points outside the allocated address space of the process.

`rewinddir()` might fail if any of the following conditions are encountered:

[EBADF]	*dirp* does not refer to an open directory stream.
[EFAULT]	*dirp* points outside the allocated address space of the process.

EXAMPLES

The following code searches the current directory for an entry *name*:

```
DIR *dirp;
struct dirent *dp;

dirp = opendir(".");
while ((dp = readdir(dirp)) != NULL) {
     if (strcmp(dp->d_name, name) == 0) {
          (void) closedir(dirp);
          return FOUND;
     }
}
(void) closedir(dirp);
return NOT_FOUND;
```

WARNINGS

readdir() and **getdirentries()** (see *getdirentries*(2) are the only ways to access remote NFS directories. Attempting to read a remote directory via NFS by using **read()** returns **-1** and sets **errno** to EISDIR (see *read*(2)).

APPLICATION USAGE

The header file required for these functions and the type of the return value from **readdir()** has been changed for compatibility with System V Release 3 and the *X/Open Portability Guide*. See *ndir*(5) for a description of the header file <**ndir.h**>, which is provided to allow existing HP-UX applications to compile unmodified.

New applications should use the <**dirent.h**> header file for portability to System V and X/Open systems.

AUTHOR

directory was developed by AT&T, HP, and the University of California, Berkeley.

SEE ALSO

close(2), getdirentries(2), lseek(2), open(2), read(2), dir(4), dirent(5), ndir(5).

STANDARDS CONFORMANCE

closedir(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1
opendir(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

readdir(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1
rewinddir(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

seekdir(): AES, XPG2, XPG3, XPG4
telldir(): AES, XPG2, XPG3, XPG4

NAME

div(), ldiv() - integer division and remainder

SYNOPSIS

```
#include <stdlib.h>

div_t div(int numer, int denom);

ldiv_t ldiv(long int numer, long int denom);
```

DESCRIPTION

`div()` Computes the quotient and remainder of the division of the numerator *numer* by the denominator *denom*. If the division is inexact, the sign of the resulting quotient is that of the algebraic quotient, and the magnitude of the resulting quotient is the largest integer less than the magnitude of the algebraic quotient. If the result can be represented, the result is returned in a structure of type `div_t` (defined in `<stdlib.h>`) having members *quot* and *rem* for the quotient and remainder respectively. Both members have type `int` and values such that *quot* × *denom* + *rem* = *numer*. If the result cannot be represented, the behavior is undefined.

`ldiv()` Similar to `div()`, except that the arguments each have type `long int` and the result is returned in a structure of type `ldiv_t` (defined in `<stdlib.h>`) having `long int` members *quot* and *rem* for the quotient and remainder respectively.

WARNINGS

Behavior is undefined if *denom* is zero.

SEE ALSO

floor(3M).

STANDARDS CONFORMANCE

`div()`: AES, XPG4, ANSI C

`ldiv()`: AES, XPG4, ANSI C

NAME

drand48(), erand48(), lrand48(), nrand48(), mrand48(), jrand48(), srand48(), seed48(), lcong48() - generate uniformly distributed pseudo-random numbers

SYNOPSIS

```
#include <stdlib.h>

double drand48(void);

double erand48(unsigned short int xsubi[3]);

long int lrand48(void);

long int nrand48(unsigned short int xsubi[3]);

long int mrand48(void);

long int jrand48(unsigned short int xsubi[3]);

void srand48(long int seedval);

unsigned short int *seed48(unsigned short int seed16v[3]);

void lcong48(unsigned short int param[7]);
```

DESCRIPTION

This family of functions generates pseudo-random numbers using the well-known linear congruential algorithm and 48-bit integer arithmetic.

In the following discussion, the formal mathematical notation [0.0, 1.0) indicates an interval including 0.0 but not including 1.0.

drand48() and **erand48()** return non-negative double-precision floating-point values uniformly distributed over the interval [0.0, 1.0).

lrand48() and **nrand48()** return non-negative long integers uniformly distributed over the interval $[0, 2^{31})$.

mrand48() and **jrand48()** return signed long integers uniformly distributed over the interval $[-2^{31}, 2^{31})$.

srand48(), **seed48()**, and **lcong48()** are initialization entry points, one of which should be invoked before either **drand48()**, **lrand48()**, or **mrand48()** is called. (Although it is not recommended practice, constant default initializer values are supplied automatically if **drand48()**, **lrand48()**, or **mrand48()** is called without a prior call to an initialization entry point.) **erand48()**, **nrand48()**, and **jrand48()** do not require an initialization entry point to be called first.

All the routines work by generating a sequence of 48-bit integer values, X_i, according to the linear congruential formula

$$X_{n+1} = (aX_n + c) \bmod m \qquad n \geq 0$$

The parameter $m = 2^{48}$; hence 48-bit integer arithmetic is performed. Unless **lcong48()** has been invoked, the multiplier value a and the addend value c are given by

$a = 5\text{DEECE66D}_{16} = 273673163155_8$
$c = \text{B}_{16} = 13_8$.

The value returned by any of the functions **drand48()**, **erand48()**, **lrand48()**, **nrand48()**, **mrand48()**, or **jrand48()** is computed by first generating the next 48-bit X_i in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (leftmost) bits of X_i and transformed into the returned value.

The functions **drand48()**, **lrand48()**, and **mrand48()** store the last 48-bit X_i generated in an internal buffer; that is why they must be initialized prior to being invoked. The functions **erand48()**, **nrand48()**, and **jrand48()** require the calling program to provide storage for the successive X_i values in the array specified as an argument when the functions are invoked. That is why these routines do not have to be initialized; the calling program merely has to place the desired initial value of X_i into the array and pass it as an argument. By using different arguments, **erand48()**, **nrand48()**, and **jrand48()** allow separate modules of a large program to generate several *independent* streams of pseudo-random

numbers; i.e., the sequence of numbers in each stream do *not* depend upon how many times the routines have been called to generate numbers for the other streams.

The initializer function **srand48()** sets the high-order 32 bits of X_i to the 32 bits contained in its argument. The low-order 16 bits of X_i are set to the arbitrary value $330E_{16}$.

The initializer function **seed48()** sets the value of X_i to the 48-bit value specified in the argument array. In addition, the previous value of X_i is copied into a 48-bit internal buffer, used only by **seed48()**, and a pointer to this buffer is the value returned by **seed48()**. This returned pointer, which can just be ignored if not needed, is useful if a program is to be restarted from a given point at some future time - use the pointer to get at and store the last X_i value, and then use this value to reinitialize via **seed48()** when the program is restarted.

The initialization function **lcong48()** allows the user to specify the initial X_i, the multiplier value a, and the addend value c. Argument array elements *param*[0-2] specify X_i, *param*[3-5] specify the multiplier a, and *param*[6] specifies the 16-bit addend c. After **lcong48()** has been called, a subsequent call to either **srand48()** or **seed48()** restores the "standard" multiplier and addend values, a and c, specified above.

SEE ALSO

rand(3C).

STANDARDS CONFORMANCE

drand48(): AES, SVID2, XPG2, XPG3, XPG4
erand48(): AES, SVID2, XPG2, XPG3, XPG4

jrand48(): AES, SVID2, XPG2, XPG3, XPG4
lcong48(): AES, SVID2, XPG2, XPG3, XPG4

lrand48(): AES, SVID2, XPG2, XPG3, XPG4
mrand48(): AES, SVID2, XPG2, XPG3, XPG4

nrand48(): AES, SVID2, XPG2, XPG3, XPG4
seed48(): AES, SVID2, XPG2, XPG3, XPG4

srand48(): AES, SVID2, XPG2, XPG3, XPG4

NAME

ecvt(), fcvt,() gcvt,() nl_gcvt() - convert floating-point number to string

SYNOPSIS

```
char *ecvt(double value, size_t ndigit, int *decpt, int *sign);
char *fcvt(double value, size_t ndigit, int *decpt, int *sign);
char *gcvt(double value, size_t ndigit, char *buf);
char *nl_gcvt(double value, size_t ndigit, char *buf, int langid);
```

DESCRIPTION

`ecvt()` Converts *value* to a null-terminated string of *ndigit* digits and returns a pointer to the string. The high-order digit is non-zero, unless the value is zero. The low-order digit is rounded. The position of the radix character relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). The radix character is not included in the returned string. If the sign of the result is negative, the word pointed to by *sign* is non-zero, otherwise it is zero.

One of three non-digit characters strings could be returned if the converted value is out of range. A `--` or `++` is returned if the value is larger than the exponent can contain, and is negative, or positive, respectively. The third string is returned if the number is illegal, a zero divide for example. The result value is Not A Number (NAN) and would return a `?` character.

`fcvt()` Identical to `ecvt()`, except that the correct digit has been rounded for printf `%f` (FORTRAN F-format) output of the number of digits specified by *ndigit*.

`gcvt()` Converts the *value* to a null-terminated string in the array pointed to by *buf* and returns *buf*. It produces *ndigit* significant digits in FORTRAN F-format if possible, or E-format otherwise. A minus sign, if required, and a radix character is included in the returned string. Trailing zeros are suppressed. The radix character is determined by the currently loaded NLS environment (see *setlocale*(3C)). If `setlocale()` has not been called successfully, the default NLS environment, "C", is used (see *lang*(5)). The default environment specifies a period (`.`) as the radix character.

`nl_gcvt()`
differs from `gcvt()` only by first calling `langinit()` (see *nl_init*(3C)) to load the NLS environment according to the language specified by *langid*.

EXTERNAL INFLUENCES

Locale

The `LC_NUMERIC` category determines the value of the radix character within the current NLS environment.

WARNINGS

The values returned by `ecvt()` and `fcvt()` point to a single static data array whose content is overwritten by each call.

`nl_gcvt()` is provided for historical reasons only; its use is not recommended.

AUTHOR

`ecvt()` and `fcvt()` were developed by AT&T. `gcvt()` was developed by AT&T and HP. `nl_gcvt()` was developed by HP.

SEE ALSO

setlocale(3C), printf(3S), hpnls(5), lang(5).

STANDARDS CONFORMANCE

`ecvt()`: XPG2

`fcvt()`: XPG2

`gcvt()`: XPG2

NAME

end, etext, edata - last locations in program

SYNOPSIS

```
extern void *_end, *end, *_etext, *etext, *_edata, *edata;
```

DESCRIPTION

These names refer neither to routines nor to locations with interesting contents. The address of the symbols **_etext** and **etext** is the first address above the program text, the address of **_edata** and **edata** is the first address above the initialized data region, and the address of **_end** and **end** is the first address above the uninitialized data region.

The linker defines these symbols with the appropriate values if they are referenced by the program but not defined. The linker issues an error if the user attempts to define **_etext**, **_edata**, or **_end**.

When execution begins, the program break (the first location beyond the data) coincides with **_end**, but the program break can be reset by the routines of *brk*(2), *malloc*(3C), standard input/output (*stdio*(3S)), the profile (**-p**) option of *cc*(1), and so on. Thus, the current value of the program break should be determined by **sbrk(0)** (see *brk*(2)).

WARNINGS

In C, these names must look like addresses. Thus, use **&end** instead of **end** to access the current value of *end*.

DEPENDENCIES

Series 700 and 800:

The linker defines the following two symbols:

_text_start	The beginning address of the program's text area.
_data_start	The beginning address of the program's data area.

SEE ALSO

cc(1), ld(1), brk(2), crt0(3), malloc(3C), stdio(3S).

STANDARDS CONFORMANCE

end: XPG2

edata: XPG2

etext: XPG2

NAME

erf, erfc - error function and complementary error function

SYNOPSIS

```
#include <math.h>

double erf(double x);

double erfc(double x);
```

DESCRIPTION

`erf()` returns the error function of x, defined as:

$$\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt.$$

`erfc()`, which returns 1.0 − `erf(`x`)`, is provided because of the extreme loss of relative accuracy if `erf(`x`)` is called for large x and the result subtracted from 1.0 (for example, for x = 5, twelve places are lost).

`erf()` returns 1.0 when x is +INFINITY , or −1.0 when x is −INFINITY .

`erfc()` returns 0.0 when x is +INFINITY , or 2.0 when x is −INFINITY .

ERRORS

`erf()` and `erfc()` return NaN and set `errno` to EDOM when x is NaN.

SEE ALSO

isinf(3M), isnan(3M), exp(3M).

STANDARDS CONFORMANCE

`erf()` in libm.a: AES, SVID2, XPG2, XPG3
`erf()` in libM.a: AES, XPG3, XPG4

`erfc()` in libm.a: AES, SVID2, XPG2, XPG3
`erfc()` in libM.a: AES, XPG3, XPG4

NAME

error_$c_get_text - return subsystem, module, and error texts for a status code

SYNOPSIS

C Syntax

```
void error_$c_get_text(
     status_$t status,
     char *subsys,
     long subsysmax,
     char *module,
     long modulemax,
     char *error,
     long errormax)
```

Pascal Syntax

```
procedure error_$c_get_text(
     in status: status_$t;
     out subsys: univ char;
     in subsysmax: integer32;
     out module: univ char;
     in modulemax: integer32;
     out error: univ char;
     in errormax: integer32);
```

Remarks

To view this manual entry via the *man*(1) command, use the function name shown above without the "$" character.

DESCRIPTION

The `error_$c_get_text()` call returns predefined text strings that describe the subsystem, the module, and the error represented by a status code. The strings are null terminated.

status
A status code in `status_$t` format.

subsys
A character string. The subsystem represented by the status code.

subsysmax
The maximum number of bytes to be returned in *subsys*.

module
A character string. The module represented by the status code.

modulemax
The maximum number of bytes to be returned in *module*.

error
A character string. The error represented by the status code.

errormax
The maximum number of bytes to be returned in *error*.

EXAMPLE

The following statement returns text strings for the subsystem, module, and error represented by the status code `st`:

```
error_$c_get_text (st, subsys, MAX, module, MAX, error, MAX);
```

SEE ALSO

error_$c_text(3).

NAME

error_$c_text - return an error message for a status code

SYNOPSIS

C Syntax

```
char *error_$c_text(
    status_$t status,
    char *message,
    int messagemax)
```

Pascal Syntax

```
procedure error_$c_text(
    in status: status_$t;
    out message: univ char;
    in messagemax: integer32);
```

Remarks

To view this manual entry via the *man*(1) command, use the function name shown above without the "$" character.

DESCRIPTION

`error_$c_text()` returns a null-terminated error message for reporting the completion status of a call. The error message is composed from predefined text strings that describe the subsystem, the module, and the error represented by the status code.

status	A status code in `status_$t` format.
message	A character string. The error message represented by the status code.
messagemax	The maximum number of bytes to be returned in *message*.

EXAMPLE

The following statement returns an error message for reporting the status code `st`:

```
error_$c_text (st, message, MAX);
```

SEE ALSO

error_$c_get_text(3).

NAME

error_$intro - error text database operations

Remarks

To view this manual entry via the *man*(1) command, use the function name shown above without the "$" character.

DESCRIPTION

`error_$()` calls convert status codes into textual error messages, and include:

`error_$c_get_text()`
: Return subsystem, module, and error texts for a status code.

`error_$c_text()`
: Return an error message for a status code.

There is no header file for the `error_$()` calls. They can be declared as follows:

```
extern void error_$c_get_text();
extern char *error_$c_text();
```

`error_$()` calls use the `status_$t` data type, which is defined in `<idl/c/nbase.h>`.

Data Types

`error_$()` calls take as input a status code in `status_$t` format.

`status_$t`
: A status code. Most NCS calls supply their completion status in this format. The `status_$t` type is defined as a structure containing a long integer:

```
struct status_$t {
      long all;
      }
```

However, the calls can also use `status_$t` as a set of bit fields. To access the fields in a returned status code, assign the value of the status code to a union defined as follows:

```
typedef union {
      struct {
            unsigned fail : 1,
                  subsys : 7,
                  modc : 8;
            short code;
      } s;
      long all;
} status_u;
```

`all`
: All 32 bits in the status code. If `all` is equal to `status_$ok`, the call that supplied the status was successful.

`fail`
: If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module.

`subsys`
: This indicates the subsystem that encountered the error.

`modc`
: This indicates the module that encountered the error.

`code`
: This is a signed number that identifies the type of error that occurred.

SEE ALSO

error_$c_get_text(3), error_$c_text(3).

NAME

exp, log, log10, log2, pow, sqrt, cbrt - exponential, logarithm, power, square root, cube root functions

SYNOPSIS

```
#include <math.h>

double exp(double x);

double log(double x);

double log10(double x);

double log2(double x);

double pow(double x, double y);

double sqrt(double x);

double cbrt(double x);

float expf(float x);

float logf(float x);

float log10f(float x);

float log2f(float x);

float powf(float x, float y);

float sqrtf(float x);

float cbrtf(float x);
```

DESCRIPTION

`exp()` returns e^x.

`log()` returns the natural logarithm of x. The value of x must be positive.

`log10()` returns the logarithm base ten of x. The value of x must be positive.

`log2()` returns the logarithm base two of x. The value of x must be positive.

`pow()` returns x^y. If x is 0.0, y must be positive. If x is negative, y must be an integer.

`sqrt()` returns the non-negative square root of x. The value of x must not be negative.

`cbrt()` returns the cube root of x. The value of x must not be negative.

`expf()`, `logf()`, `log10f()`, `log2f()`, `powf()`, `sqrtf()`, and `cbrtf()` are `float` versions of `exp()`, `log()`, `log10()`, `log2()`, `pow()`, `sqrt()`, and `cbrt()`; they take `float` arguments and return `float` results. Their performance is significantly faster than that of the `double` versions of the functions. Programs must be compiled in ANSI mode (with the `-Aa` option) in order to use these functions; otherwise, the compiler promotes the `float` arguments to `double`, and the functions return incorrect results.

DEPENDENCIES

Series 300/400

`log2()`, `cbrt()`, `expf()`, `logf()`, `log10f()`, `log2f()`, `powf()`, `sqrtf()`, and `cbrtf()` are not supported on Series 300/400 systems.

Series 700/800

`log2()`, `cbrt()`, `expf()`, `logf()`, `log10f()`, `log2f()`, `powf()`, `sqrtf()`, and `cbrtf()` are not specified by any standard (however, the `float` functions are named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard). These functions are provided in the PA1.1 versions of the math library only. The `+DA1.1` option (default on Series 700 systems) links in a PA1.1 version automatically. A PA1.1 library can also be linked in explicitly. For more information, see the *HP-UX Floating-Point Guide*.

/lib/libm.a

`exp()` returns:

- +INFINITY when x is +INFINITY,
- 0.0 when x is –INFINITY.

`log()`, `log2()`, and `log10()` return +INFINITY when x is +INFINITY.

`pow()` returns +INFINITY when:

- Absolute value of x is greater than 1.0 and y is +INFINITY,
- Absolute value of x is less than 1.0 and y is –INFINITY,
- x is +INFINITY and y is greater than 0.0, or
- x is –INFINITY and y is an even integer.

`pow()` returns –INFINITY when x is –INFINITY and y is an odd integer.

`pow()` returns 0.0 when:

- Absolute value of x is greater than 1.0 and y is –INFINITY,
- Absolute value of x is less than 1.0 and y is +INFINITY,
- x is +INFINITY and y is less than 0.0.

`sqrt()` and `cbrt()` return +INFINITY when x is +INFINITY.

/lib/libM.a

`exp()` returns:

- +INFINITY when x is +INFINITY,
- 0.0 when x is –INFINITY.

`log()`, `log2()`, and `log10()` return +INFINITY when x is +INFINITY.

`pow()` returns 1.0 when x and y are both 0.0.

`pow()` returns +INFINITY when:

- Absolute value of x is greater than 1.0 and y is +INFINITY,
- Absolute value of x is less than 1.0 and y is –INFINITY,
- x is +INFINITY and y is greater than 0.0, or
- x is –INFINITY and y is an even integer.

`pow()` returns –INFINITY when x is –INFINITY and y is an odd integer.

`pow()` returns 0.0 when:

- Absolute value of x is greater than 1.0 and y is –INFINITY,
- Absolute value of x is less than 1.0 and y is +INFINITY,
- x is +INFINITY and y is less than 0.0.

`sqrt()` and `cbrt()` return +INFINITY when x is +INFINITY.

ERRORS

/lib/libm.a

`exp()` returns HUGE_VAL when the correct value would overflow, or 0.0 when the correct value would underflow, and sets `errno` to ERANGE. NaN is returned and `errno` is set to EDOM when x is NaN.

`log()`, `log2()`, and `log10()` return –HUGE_VAL and set `errno` to EDOM when x is non-positive. NaN is returned and `errno` is set to EDOM when x is NaN or –INFINITY. A message indicating DOMAIN error (or SING error when x is 0.0) is printed on the standard error output in these cases.

`pow()` returns 0.0 and sets `errno` to EDOM when x is 0.0 and y is negative, or when x is negative and y is not an integer. NaN is returned and `errno` is set to EDOM when x or y is NaN. In these cases a message indicating DOMAIN error is printed on the standard error output. When the correct value for `pow()` would overflow or underflow, `pow()` returns ±HUGE_VAL or 0.0 respectively, and sets `errno` to ERANGE.

`sqrt()` returns NaN and sets `errno` to EDOM when x is negative, NaN or -INFINITY. A message indicating DOMAIN error is printed on the standard error output.

`cbrt()` returns 0.0 and sets `errno` to EDOM when x is negative. NaN is returned and `errno` is set to EDOM when x is NaN. In these cases a message indicating DOMAIN error is printed on the standard error output. When the correct value for `cbrt()` would overflow or underflow, `cbrt()` returns ±HUGE_VAL or 0.0 respectively, and sets `errno` to ERANGE.

These error-handling procedures can be changed with the `matherr()` function (see *matherr*(3M)).

/lib/libM.a

No error messages are printed on the standard error output.

`exp()` returns HUGE_VAL when the correct value would overflow, or 0.0 when the correct value would underflow, and sets `errno` to ERANGE. NaN is returned and `errno` is set to EDOM when x is NaN.

`log()`, `log2()`, and `log10()` return NaN and set `errno` to EDOM when x is negative, -INFINITY, or NaN. -HUGE_VAL is returned and `errno` is set to EDOM when x is 0.0.

`pow()` returns -HUGE_VAL and sets `errno` to EDOM when x is 0.0 and y is negative. NaN is returned and `errno` is set to EDOM when x is negative and y is not an integer or when x or y is NaN. When the correct value for `pow()` would overflow or underflow, `pow()` returns ±HUGE_VAL or 0.0 respectively, and sets `errno` to ERANGE.

`sqrt()` returns NaN and sets `errno` to EDOM when x is negative, NaN or -INFINITY.

`cbrt()` returns NaN and sets `errno` to EDOM when x is negative or when x is NaN. When the correct value for `cbrt()` would overflow or underflow, `cbrt()` returns ±HUGE_VAL or 0.0 respectively, and sets `errno` to ERANGE.

These error-handling procedures can be changed by using the `_matherr()` function (see *matherr*(3M)). Note that `_matherr()` is provided in order to assist in migrating programs from `libm.a` to `libM.a` and is *not* a part of XPG3, ANSI C, or POSIX.

SEE ALSO

isinf(3M), isnan(3M), matherr(3M).

STANDARDS CONFORMANCE

`exp()` in libm.a: AES, SVID2, XPG2, XPG3, FIPS 151-2, POSIX.1
`exp()` in libM.a: AES, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`log()` in libm.a: AES, SVID2, XPG2, XPG3, FIPS 151-2, POSIX.1
`log()` in libM.a: AES, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`log10()` in libm.a: AES, SVID2, XPG2, XPG3, FIPS 151-2, POSIX.1
`log10()` in libM.a: AES, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`pow()` in libm.a: AES, SVID2, XPG2, XPG3, FIPS 151-2, POSIX.1
`pow()` in libM.a: AES, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`sqrt()` in libm.a: AES, SVID2, XPG2, XPG3, FIPS 151-2, POSIX.1
`sqrt()` in libM.a: AES, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

exportent(), getexportent(), setexportent(), addexportent(), remexportent(), endexportent(), getexportopt() - access exported file system information

SYNOPSIS

```
#include <stdio.h>
#include <exportent.h>

FILE *setexportent();

struct exportent *getexportent(FILE *fildep);

int addexportent(FILE *filep, char *dirname, char *options);

int remexportent(FILE *filep, char *dirname);

char *getexportopt(struct exportent *xent, char *opt);

void endexportent(FILE *filep);
```

DESCRIPTION

These routines access the exported filesystem information in `/etc/xtab`.

`setexportent()` Open the export information file and return a file pointer to use with `getexportent()`, `addexportent()`, `remexportent()`, and `endexportent()`. Returns NULL if the file is locked or if an error is encountered in opening the file.

`getexportent()` Read the next line from *filep* and return a pointer to an object with the following structure containing the broken-out fields of a line in file `/etc/xtab`. The fields have meanings described in *exports*(4).

```
#define ACCESS_OPT "access" /* machines that can mount fs */
#define ROOT_OPT   "root"   /* machines with root access to fs */
#define RO_OPT     "ro"     /* export read-only */
#define ANON_OPT   "anon"   /* uid for anonymous requests */
#define ASYNC_OPT  "async" /* all mounts will be aynchronous */

struct exportent {
       char *xent_dirname; /* directory (or file) to export */
       char *xent_options; /* options, as above */
       };
```

`getexportent()` returns NULL if it encounters end of file.

`addexportent()` Add the *exportent* to the end of the open file *filep*. It returns 0 if successful and –1 on failure.

`remexportent()` Remove the indicated entry from the list. Returns 0 on success and –1 on failure.

`getexportopt()` Scans the `xent_options` field of the `exportent` structure for a substring that matches *opt*. Returns the string value of *opt*, or NULL if the option is not found.

`endexportent()` Close the file.

RETURN VALUE

`setexportent()`, `getexportent()`, and `getexportopt()` return a NULL pointer on EOF or error.

`addexportent()` and `remexportent()` return –1 if they fail.

WARNINGS

The returned `exportent` structure points to static information that is overwritten in each call.

AUTHOR

`exportent`, `getexportent()`, `setexportent()`, `addexportent()`, `remexportent()`, `endexportent()`, and `getexportopt()` were developed by Sun Microsystems, Inc.

FILES

`/etc/exports /etc/xtab`

SEE ALSO
exportfs(1M), exports(4).

NAME

fclose(), fflush() - close or flush a stream

SYNOPSIS

```
#include <stdio.h>

int fclose(FILE *stream);

int fflush(FILE *stream);
```

DESCRIPTION

`fclose()` causes any buffered data for the named *stream* to be written out, and the *stream* to be closed. Buffers allocated by the standard input/output system may be freed.

`fclose()` is performed automatically for all open files upon calling *exit*(2).

If *stream* points to an output stream or an update stream in which the most recent operation was output, `fflush()` causes any buffered data for the *stream* to be written to that file; otherwise any buffered data is discarded. The *stream* remains open.

If *stream* is a null pointer, `fflush()` performs this flushing action on all currently open streams.

RETURN VALUE

Upon successful completion, `fclose()` and `fflush()` return 0. Otherwise, they return EOF and set `errno` to indicate the error.

ERRORS

`fclose()` and `fflush()` fail if:

[EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation.

[EBADF] The file descriptor underlying *stream* is not valid.

[EFBIG] An attempt was made to write a file that exceeds the process's file size limit or the maximum file size (see *ulimit*(2)).

[EINTR] `fclose()` or `fflush()` was interrupted by a signal.

[EIO] The process is in a background process group and is attempting to write to its controlling terminal, `TOSTOP` is set, the process is neither ignoring nor blocking the `SIGTTOU` signal, and the process group of the process is orphaned.

[ENOSPC] There was no free space remaining on the device containing the file.

[EPIPE] An attempt was made to write to a pipe that is not open for reading by any process. A `SIGPIPE` signal is also sent to the process.

Additional `errno` values may be set by the underlying `write()`, `lseek()`, and `close()` functions (see *write*(2), *lseek*(2) and *close*(2)).

SEE ALSO

close(2), exit(2), lseek(2), write(2), fopen(3S), setbuf(3S).

STANDARDS CONFORMANCE

`fclose()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`fflush()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

ferror(), feof(), clearerr() - stream status inquiries

SYNOPSIS

```
#include <stdio.h>

int ferror(FILE *stream);

int feof(FILE *stream);

void clearerr(FILE *stream);
```

DESCRIPTION

`ferror()` Returns non-zero when an I/O error has previously occurred reading from or writing to the named *stream*, otherwise zero. Unless cleared by `clearerr()`, or unless the specific *stdio* routine so indicates, the error indication lasts until the stream is closed.

`feof()` Returns non-zero when EOF has previously been detected reading the named input *stream*, otherwise zero.

`clearerr()` Resets the error indicator and EOF indicator on the named *stream* to zero.

WARNINGS

All these routines are implemented both as library functions and as macros. The macro versions, which are used by default, are defined in `<stdio.h>`. To obtain the library function, either use a `#undef` to remove the macro definition or, if compiling in ANSI-C mode, enclose the function name in parentheses or use the function address. The following example illustrates each of these methods :

```
#include <stdio.h>
#undef ferror
    ...
main()
{
      int (*find_error()) ();
          ...
      return_val=ferror(fd);
          ...
      return_val=(feof)(fd1);
          ...
      find_error = feof;
};
```

SEE ALSO

open(2), fopen(3S).

STANDARDS CONFORMANCE

`ferror()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`clearerr()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`feof()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

fgetpos(), fsetpos() - save and restore a file position indicator for a stream

SYNOPSIS

```
#include <stdio.h>

int fgetpos(FILE *stream, fpos_t *pos);

int fsetpos(FILE *stream, const fpos_t *pos);
```

DESCRIPTION

`fgetpos()` Store the current value of the file position indicator for the stream pointed to by *stream* in the object pointed to by *pos*. The value stored contains information usable by `fsetpos()` for repositioning the stream to its position at the time of the call to `fgetpos()`.

`fsetpos()` Set the file position indicator for the stream pointed to by *stream* according to the value of the object pointed to by *pos*, which must be a value set by an earlier call to `fgetpos()` on the same stream.

A successful call to `fsetpos()` clears the end-of-file indicator for the stream and undoes any effects of *ungetc*(3S) on the same stream. After a `fsetpos()` call, the next operation on a update stream can be either input or output.

RETURN VALUE

If successful, these functions return zero; otherwise non-zero.

WARNINGS

Failure can occur if these functions are used on a file that has not been opened via `fopen()`. In particular, they must not be used on a terminal or on a file opened via *popen*(3S).

`fsetpos()` has no effect on streams that are open for append (see *fopen*(3S)).

SEE ALSO

fseek(3S), fopen(3S), popen(3S), ungetc(3S).

STANDARDS CONFORMANCE

`fgetpos()`: AES, XPG4, ANSI C

`fsetpos()`: AES, XPG4, ANSI C

NAME

fgetws() - get a wide character string from a stream file

SYNOPSIS

```
#include <wchar.h>

wchar_t *fgetws(wchar_t *ws, int n, FILE *stream);
```

Remarks:

This function is compliant with the XPG4 Worldwide Portability Interface wide-character I/O functions. It parallels the 8-bit character I/O function defined in *gets(3S)*.

DESCRIPTION

`fgetws()` Reads characters from the *stream*, converts them into corresponding wide characters, and places them into the array pointed to by *ws*, until $n - 1$ characters are read, a new-line character is read and transferred to *ws*, or an end-of-file condition is encountered. The wide string is then terminated with a null wide character.

The definition for this functions and the type `wchar_t` are provided in the <`wchar.h`> header.

EXTERNAL INFLUENCES

Locale

The `LC_CTYPE` category determines how wide character conversions are done.

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUE

Upon successful completion, `fgetws()` returns *ws*. If the stream is at end-of-file, the end-of-file indicator for the stream is set and a null pointer is returned. If a read error occurs, the error indicator for the stream is set, `errno` is set to indicate the error, and a null pointer is returned.

`ferror()` and `feof()` can be used to distinguish between an error condition and an end-of-file condition.

ERRORS

`fgetws()` fails if data needs to be read into the *stream*'s buffer, and:

[EAGAIN] The `O_NONBLOCK` flag is set for the file descriptor underlying *stream* and the process would be delayed in the read operation.

[EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for reading.

[EINTR] The read operation was terminated due to the receipt of a signal, and either no data was transferred or the implementation does not report partial transfer for this file.

[EIO] The process is a member of a background process and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the `SIGTTIN` signal or the process group of the process is orphaned.

[EILSEQ] The data obtained from the input stream do not form a valid wide character string.

Additional `errno` values can be set by the underlying `read()` function (see *read*(2)).

SEE ALSO

ferror(3S), fopen(3S), fread(3S), getwc(3C), fputws(3C), scanf(3S).

STANDARDS CONFORMANCE

`fgetws()`: XPG4

NAME

fileno( ) - map stream pointer to file descriptor

SYNOPSIS

```
#include <stdio.h>

int fileno(FILE *stream);
```

DESCRIPTION

`fileno( )` returns the integer file descriptor associated with the named *stream*; see *open*(2).

The following symbolic values in <`unistd.h`> define the file descriptors associated with `stdin`, `stdout`, and `stderr` when a program is started :

`STDIN_FILENO`	Value of zero for standard input, `stdin`.
`STDOUT_FILENO`	Value of 1 for standard output, `stdout`.
`STDERR_FILENO`	Value of 2 for standard error, `stderr`.

RETURN VALUE

Upon error, `fileno( )` returns -1.

SEE ALSO

open(2), fopen(3S).

STANDARDS CONFORMANCE

`fileno( )`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

floor, ceil, fmod, fabs, rint, fabsf, fmodf - floor, ceiling, remainder, absolute value, and round-to-nearest functions

SYNOPSIS

```
#include <math.h>

double floor(double x);

double ceil(double x);

double fmod(double x, double y);

double fabs(double x);

double rint(double x);

float fabsf(float x);

float fmodf(float x, float y);
```

DESCRIPTION

`floor()` returns the largest integer (as a double-precision number) not greater than x.

`ceil()` returns the smallest integer not less than x.

`fmod()` returns the floating-point remainder (f) of the division of x by y, where f has the same sign as x, such that $x = iy + f$ for some integer i, and $|f| < |y|$.

`fabs()` returns the absolute value of x, $|x|$.

`rint()` returns the integer (represented as a double precision number) nearest x in the direction of the prevailing rounding mode.

`fabsf()` and `fmodf()` are `float` versions of `fabs()` and `fmod()`; they take `float` arguments and return `float` results. Their performance is significantly faster than that of the `double` versions. Programs must be compiled in ANSI mode (with the `-Aa` option) in order to use these functions; otherwise, the compiler promotes the `float` arguments to `double`, and the functions return incorrect results.

DEPENDENCIES

Series 300/400

`fabsf()`, `fmodf()`, and `rint()` are not supported on Series 300/400 systems.

Series 700/800

`fabsf()`, `fmodf()`, and `rint()` are not specified by any standard (`fabsf()` and `fmodf()` are, however, named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard). These functions are provided in the PA1.1 versions of the math library only. The `+DA1.1` linker option (default on Series 700 systems) links in a PA1.1 version automatically. A PA1.1 library can also be linked in explicitly. For more information, see the *HP-UX Floating-Point Guide*.

/lib/libm.a

When x is ±INFINITY, `floor()`, `ceil()`, and `rint()` return ±INFINITY respectively.

`fabs()` returns +INFINITY when x is ±INFINITY.

`fmod()` returns x if y is 0.0, if x/y would overflow, or if x/y would underflow (including when y is ±INFINITY).

/lib/libM.a

No error messages are printed on the standard error output.

When x is ±INFINITY, `floor()`, `ceil()`, and `rint()` return ±INFINITY respectively.

`fabs()` returns +INFINITY when x is ±INFINITY.

`fmod()` returns 0.0 if x/y would overflow, or x if x/y would underflow (including when y is ±INFINITY).

NOTES

In the default rounding mode (round to nearest), on a machine that conforms to the IEEE-754 standard, `rint(`x`)` is the integer nearest x with the additional stipulation that if $|$`rint(`x`)` $-x| = 1/2$, then `rint(`x`)` is even. Other rounding modes can make `rint()` act like `floor()`, or like `ceil()`, or round

toward 0.

Another way to obtain an integer near x is to declare (in C):

```
double x;  int k;  k = x;
```

The HP C compiler rounds `x` toward 0 to get the integer `k`. Note that if `x` is larger than `k` can accommodate, the value of `k` and the presence or absence of an integer overflow are hard to predict.

ERRORS

/lib/libm.a

`floor()` and `ceil()` return NaN and set `errno` to EDOM when x is NaN.

`fmod()` returns NaN and sets `errno` to EDOM when x or y is NaN, or when x is ±INFINITY.

`fabs()` returns NaN and sets `errno` to EDOM when x is NaN.

/lib/libM.a

`floor()` and `ceil()` return NaN and set `errno` to EDOM when x is NaN.

`fmod()` returns NaN and sets `errno` to EDOM when y is 0.0, when x or y is NaN, or when x is ±INFINITY.

`fabs()` returns NaN and sets `errno` to EDOM when x is NaN.

SEE ALSO

abs(3C), isinf(3M), isnan(3M), ieee(3M).

STANDARDS CONFORMANCE

`floor()` in libm.a: AES, SVID2, XPG2, XPG3, FIPS 151-2, POSIX.1
`floor()` in libM.a: AES, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`ceil()` in libm.a: AES, SVID2, XPG2, XPG3, FIPS 151-2, POSIX.1
`ceil()` in libM.a: AES, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`fabs()` in libm.a: AES, SVID2, XPG2, XPG3, FIPS 151-2, POSIX.1
`fabs()` in libM.a: AES, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`fmod()` in libm.a: AES, SVID2, XPG2, XPG3, FIPS 151-2, POSIX.1
`fmod()` in libM.a: AES, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

fnmatch() - match filename patterns

SYNOPSIS

```
#include <unistd.h>

int fnmatch(const char *pattern, const char *string, int flags);
```

DESCRIPTION

`fnmatch()` performs pattern matching as described in *regexp*(5) under *PATTERN MATCHING NOTATION*. By default, the rule qualifications for filename expansion do not apply; i.e., periods (dots) and slashes are matched as ordinary characters. This default behavior can be modified by using the flags described below.

The *flag* argument modifies the interpretation of *pattern* and *string*. If **FNM_PATHNAME**, which is defined in <unistd.h>, is set in *flag*, a slash character in *string* must be explicitly matched by a slash in *pattern;* it cannot be matched by either the asterisk or question mark special characters or by a bracket expression.

If **FNM_PERIOD** is set in *flag*, a leading period (.) must be explicitly matched. It will not be matched by a bracket expression, question mark or asterisk. By default, a period is leading if it is the first character in *string*. If **FNM_PATHNAME** is set in *flag*, a period is leading if it is the first character in *string* or immediately follows a slash.

If **FNM_NOESCAPE** is not set in *flag*, a backslash character (\) in *pattern* followed by any other character matches that second character in *string*. In particular, \\ matches a backslash in *string*. If **FNM_NOESCAPE** is set, a backslash character is treated as an ordinary character.

If *flag* is zero, the slash character and the period are treated as regular characters. If *flag* has any other value, the result is undefined.

RETURN VALUE

If *string* matches the pattern specified by *pattern*, `fnmatch()` returns zero. Otherwise, `fnmatch()` returns non-zero.

EXAMPLE

The following excerpt uses `fnmatch()` to check each file in a directory against the pattern `*.c`:

```
pattern = "*.c";

while(dp = readdir(dirp)){
    if((fnmatch(pattern, dp->d_name,0)) == 0){
        /* do processing for match */
              ...
    } }
```

SEE ALSO

sh(1), glob(3c).

STANDARDS CONFORMANCE

`fnmatch()`: XPG4, POSIX.2

NAME

fopen(), freopen(), fdopen() - open or re-open a stream file; convert file to stream

SYNOPSIS

```
#include <stdio.h>

FILE *fopen(const char *pathname, const char *type);

FILE *freopen(const char *pathname, const char *type, FILE *stream);

FILE *fdopen(int fildes, const char *type);
```

DESCRIPTION

fopen() Opens the file named by *pathname* and associates a *stream* with it. **fopen()** returns a pointer to the **FILE** structure associated with the *stream*.

freopen() substitutes the named file in place of the open *stream*. The original *stream* is closed, regardless of whether the open ultimately succeeds. **freopen()** returns a pointer to the **FILE** structure associated with *stream* and makes an implicit call to **clearerr()** (see *ferror*(3S)).

freopen() is typically used to attach the preopened *streams* associated with **stdin**, **stdout**, and **stderr** to other files.

fdopen() associates a stream with a file descriptor. File descriptors are obtained from **open()**, **dup()**, **creat()**, or **pipe()** (see *open*(2), *dup*(2), *creat*(2), and *pipe*(2)), which open files but do not return pointers to a **FILE** structure stream. Streams are necessary input for many of the Section (3S) library routines. The *type* of stream must agree with the mode of the open file. The meanings of *type* used in the **fdopen()** call are exactly as specified above, except that **w**, **w+**, **wb**, and **wb+** do not cause truncation of the file.

pathname Points to a character string containing the name of the file to be opened.

type Character string having one of the following values:

r	open for reading
w	truncate to zero length or create for writing
a	append; open for writing at end of file, or create for writing
rb	open binary file for reading
wb	truncate to zero length or create binary file for writing
ab	append; open binary file for writing at end-of-file, or create binary file
r+	open for update (reading and writing)
w+	truncate to zero length or create for update
a+	append; open or create for update at end-of-file
r+b *or* **rb+**	open binary file for update (reading and writing)
w+b *or* **wb+**	truncate to zero length or create binary file for update
a+b *or* **ab+**	append; open or create binary file for update at end-of-file

When a file is opened for update, both input and output can be done on the resulting *stream*. However, output cannot be directly followed by input without an intervening call to **fflush()** or to a file positioning function (**fseek()**, **fsetpos()**, or **rewind()**), and input cannot be directly followed by output without an intervening call to a file positioning function unless the input operation encounters end-of-file.

When a file is opened for append (i.e., when *type* is **a** or **a+**), it is impossible to overwrite information already in the file. All output is written at the end of the file, regardless of intervening calls to **fseek()**. If two separate processes open the same file for append, each process can write freely to the file without fear of destroying output being written by the other. Output from the two processes will be intermixed in the file in the order in which it is written.

RETURN VALUE

Upon successful completion, **fopen()**, **fdopen()**, and **freopen()** return a **FILE** * pointer to the

stream. Otherwise, a null pointer is returned and `errno` is set to indicate the error.

ERRORS

`fopen()`, `fdopen()`, and `freopen()` fail if:

[EINVAL] The *type* argument is not a valid mode.

[ENOMEM] There is insufficient space to allocate a buffer.

`fopen()` and `freopen()` fail if:

[EACCES] Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by *type* are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created.

[EINTR] A signal was caught during `fopen()` or `freopen()`. function.

[EISDIR] The named file is a directory and *type* requires write access.

[EMFILE] The calling process has attempted to exceed its open file limit.

[ENAMETOOLONG]
The length of the *pathname* string exceeds **PATH_MAX** or a pathname component is longer than **NAME_MAX** while POSIX_NO_TRUNC is in effect.

[ENFILE] The system file table is full.

[ENOENT] The named file does not exist or the *pathname* argument points to an empty string.

[ENOSPC] The directory or file system that would contain the new file cannot be expanded, the file does not exist, and it was to be created.

[ENOTDIR] A component of the path prefix is not a directory.

[ENXIO] The named file is a character special or block special file, and the device associated with the special file does not exist.

[EROFS] The named file resides on a read-only file system and *type* requires write access.

Additional `errno` values can be set by the underlying `open()` call made from the `fopen()` and `freopen()` functions (see *open(2)*).

NOTES

HP-UX binary file *types* are equivalent to their non-binary counterparts. For example, types `r` and `rb` are equivalent.

SEE ALSO

creat(2), dup(2), open(2), pipe(2), fclose(3S), fseek(3S), popen(3S), setvbuf(3S).

STANDARDS CONFORMANCE

`fopen()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`fdopen()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

`freopen()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

fpclassify(), fpclassifyf() - floating-point operand classification functions

SYNOPSIS

```
#include <math.h>

int fpclassify(double x);

int fpclassifyf(float x);
```

DESCRIPTION

`fpclassify()` and `fpclassifyf()` return a non-negative integer value that specifies the IEEE operand class to which the argument *x* belongs. The value returned is one of the following macros which are defined in `<math.h>`:

```
#define FP_PLUS_NORM      0  /* Positive normalized */
#define FP_MINUS_NORM     1  /* Negative normalized */
#define FP_PLUS_ZERO      2  /* Positive zero */
#define FP_MINUS_ZERO     3  /* Negative zero */
#define FP_PLUS_INF       4  /* Positive infinity */
#define FP_MINUS_INF      5  /* Negative infinity */
#define FP_PLUS_DENORM    6  /* Positive denormalized */
#define FP_MINUS_DENORM   7  /* Negative denormalized */
#define FP_SNAN           8  /* Signalling NaN */
#define FP_QNAN           9  /* Quiet NaN */
```

Every possible argument value falls into one of these ten categories, so these functions never result in an error.

`fpclassifyf()` is the `float` version of `fpclassify()`. Programs must be compiled in ANSI mode (with the `-Aa` option) in order to use this function; otherwise, the compiler promotes the `float` argument to `double`, and the function returns incorrect results.

DEPENDENCIES

Series 300/400

`fpclassify()` and `fpclassifyf()` are not supported on Series 300/400 systems.

Series 700/800

`fpclassify()` and `fpclassifyf()` are provided in the PA1.1 versions of the math library only. The `+DA1.1` option (default on Series 700 systems) links in a PA1.1 version automatically. A PA1.1 library can also be linked in explicitly. For more information, see the *HP-UX Floating-Point Guide*.

These functions are not specified by any standard. However, they implement the `class()` function suggested in the "Recommended Functions and Predicates" appendix of the IEEE-754 floating-point standard. Also, `fpclassifyf()` is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard.

SEE ALSO

isnan(3M), isinf(3M), ieee(3M).

NAME

fpgetround(), fpsetround(), fpgetmask(), fpsetmask(), fpgetsticky(), fpsetsticky(), fpsetdefaults(), fpgetcontrol(), fpsetcontrol(), fpgetfastmode(), fpsetfastmode() - floating-point mode-control functions

SYNOPSIS

```
#include <math.h>

fp_rnd fpgetround(void);

fp_rnd fpsetround(fp_rnd mode);

fp_except fpgetmask(void);

fp_except fpsetmask(fp_except value);

fp_except fpgetsticky(void);

fp_except fpsetsticky(fp_except value);

int fpgetfastmode(void);

int fpsetfastmode(int value);

void fpsetdefaults(void);

fp_control fpgetcontrol(void);

fp_control fpsetcontrol(fp_control value);
```

DESCRIPTION

The **fpgetround()** suite of functions allows programmers to manipulate the floating-point control register (also called the floating-point status register).

fpgetround() returns the current rounding mode. The type of the returned value, **fp_rnd**, is defined as follows in <**math.h**>:

```
typedef enum {
    FP_RZ=0,    /* Round toward zero */
    FP_RN,      /* Round to nearest */
    FP_RP,      /* Round toward positive infinity */
    FP_RM,      /* Round toward negative infinity */
    } fp_rnd;
```

The default value is **FP_RN**. Round-to-nearest mode rounds to the representable value closest to the true value. If two representable values are equally close to the true value, the system chooses the one whose least significant bit is zero.

fpsetround() sets the rounding mode to the specified value of type **fp_rnd** and returns the previous rounding mode.

There are five floating-point exceptions: divide-by-zero, overflow, underflow, imprecise (inexact) result, and invalid operation. If a floating-point exception occurs and the corresponding exception trap enable bit is set to 1, the trap takes place. If an exception occurs and the exception trap enable bit is set to 0, the corresponding exception flag is set to 1 and no trap takes place. The exception-trap-enable bits are sometimes called mask bits; the exception flags are sometimes called sticky bits. The routines **fpgetmask()** and **fpgetsticky()** return the current settings of these bits. To change the settings of these bits, use **fpsetmask()** and **fpsetsticky()**.

fpgetmask() returns the current exception trap enable bits. The type of the returned value, **fp_except**, is defined as **int** in <**math.h**>. The floating-point exception types are defined as follows in <**math.h**>:

```
#define FP_X_INV        0x10  /* invalid operation exception */
#define FP_X_DZ         0x08  /* divide-by-zero exception */
#define FP_X_OFL        0x04  /* overflow exception */
#define FP_X_UFL        0x02  /* underflow exception */
#define FP_X_IMP        0x01  /* imprecise (inexact result) */
#define FP_X_CLEAR      0x00  /* simply zero to clear all flags */
```

fpsetmask() sets or clears the exception trap enable bits and returns the previous setting. The argument is an expression of type **fp_except**. (To set or clear the exception trap enable bits at compile time, use the compiler option **+FP***string*).

fpgetsticky() returns the current exception flags.

fpsetsticky() sets or clears the exception flags and returns the previous setting. The argument is an expression of type **fp_except**.

fpgetfastmode() and **fpsetfastmode()** allow the programmer to change the way the system handles underflow. Fast underflow mode, also known as fastmode, is an alternative to IEEE-754-compliant underflow mode. On Series 700/800 systems, underflow involves a fault into the kernel, where the IEEE-mandated conversion of the result into a denormalized value or zero is accomplished by software emulation. On some PA1.1-based systems, fastmode causes the hardware to simply substitute a zero for the result of an operation, with no fault occurring. This may be a significant performance optimization for applications that underflow frequently. Fastmode also causes denormalized floating-point operands to be treated as if they were true zero operands.

fpgetfastmode() returns the current fastmode setting: 1 if fastmode is set, 0 if the default IEEE-754-compliant underflow mode is set. On systems that do not support fastmode, this function returns an undefined value.

On systems that support fastmode, **fpsetfastmode()** sets fastmode to either 1 (fastmode) or 0 (IEEE-754-compliant underflow mode) and returns the previous setting. On systems that do not support fastmode, this function has no effect.

fpsetdefaults() changes the default environment on Series 700 workstations, which is

Round to nearest (**FP_RN**)
All exception flags cleared (**FP_X_CLEAR**)
All exception traps disabled
Fast underflow mode disabled

fpsetdefaults() changes these defaults to more useful values. Specifically, it enables traps for the invalid operation, divide-by-zero, and overflow exceptions, while leaving the underflow and inexact-result exception traps disabled. It sets the environment as follows:

Round to nearest (**FP_RN**)
All exception flags cleared (**FP_X_CLEAR**)
All exception traps enabled except underflow and inexact result (**FP_X_INV+FP_X_DZ+FP_X_OFL**)
Fast underflow mode enabled (if the system supports it)

fpgetcontrol() and **fpsetcontrol()** access **fp0**, the floating-point unit's control register (also called the status register).

fpgetcontrol() returns the value of **fp0**. The type of the returned value, **fp_control**, is defined as **long** in <**math.h**>.

fpsetcontrol() sets the value of **fp0** and returns the previous value. For the format of **fp0**, see the *HP-UX Floating-Point Guide* or the *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*.

DEPENDENCIES

Series 300/400

These functions are not supported on Series 300/400 systems.

Series 700/800

All of these functions are provided in the PA1.1 versions of the math library only. The **+DA1.1** linker option (default on Series 700 systems) links in a PA1.1 version automatically. A PA1.1 library can be linked in explicitly. For more information, see the *HP-UX Floating-Point Guide*.

WARNINGS

fpsetsticky() modifies all exception flags. **fpsetmask()** modifies all exception trap enable bits.

Both C and FORTRAN require truncation (rounding to zero) for floating-point to integer conversions. The current rounding mode has no effect on these conversions.

NAME

fputws() - put a wide character string on a stream file

SYNOPSIS

```
#include <wchar.h>

int fputws(const wchar_t *ws, FILE *stream);
```

Remarks:

This function is compliant with the XPG4 Worldwide Portability Interface wide-character I/O functions. It parallels the 8 bit character I/O function defined in *puts(3S)*.

DESCRIPTION

`fputws()` writes a character string corresponding to the null-terminated wide-character string pointed to by *ws* to the named output *stream*, but does *not* append a new-line character or a terminating null character.

The definition for this function, the type `wchar_t` and the value `WEOF` are provided in the `<wchar.h>` header.

EXTERNAL INFLUENCES

Locale

The `LC_CTYPE` category determines how wide character conversions are done.

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUE

Upon successful completion, `fputws()` returns a non-negative number. Otherwise it returns `WEOF`, sets the error indicator for the stream, and sets `errno` to indicate the error.

ERRORS

`fputws()` fails if either the *stream* is unbuffered, or *stream*'s buffer needed to be flushed causing an underlying `write()` call to be invoked, and:

[EAGAIN]	The `O_NONBLOCK` flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation.
[EBADF]	The file descriptor underlying *stream* is not a valid file descriptor open for writing.
[EFBIG]	An attempt was made to write to a file that exceeds the process's file size limit or the maximum file size (see *ulimit*(2)).
[EINTR]	A signal was caught during the `write()` system call.
[EIO]	The process is in a background process group and is attempting to write to its controlling terminal, `TOSTOP` is set, the process is neither ignoring nor blocking the `SIGTTOU` signal, and the process group of the process is orphaned.
[ENOSPC]	There was no free space remaining on the device containing the file.
[EPIPE]	An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A `SIGPIPE` signal is also sent to the process.
[EILSEQ]	A wide character in *ws* does not correspond to a valid character.

Additional `errno` values may be set by the underlying `write()` function (see *write(2)*).

SEE ALSO

ferror(3S), fopen(3S), fread(3S), printf(3S), putwc(3C).

NOTES

`fputws()` does not append a new-line character.

STANDARDS CONFORMANCE

`fputws()`: XPG4

NAME

fread(), fwrite() - buffered binary input/output to a stream file

SYNOPSIS

```
#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream);

size_t fwrite(const void *ptr, size_t size, size_t nitems, FILE *stream);
```

DESCRIPTION

`fread()` copies, into an array pointed to by *ptr*, *nitems* items of data from the named input *stream*, where an item of data is a sequence of bytes (not necessarily terminated by a null byte) of length *size*. `fread()` stops appending bytes if an end-of-file or error condition is encountered while reading *stream*, or if *nitems* items have been read. `fread()` leaves the file pointer in *stream*, if defined, pointing to the byte following the last byte read if there is one. `fread()` does not change the contents of *stream*.

`fwrite()` appends at most *nitems* items of data from the array pointed to by *ptr* to the named output *stream*. `fwrite()` stops appending when it has appended *nitems* items of data or if an error condition is encountered on *stream*. `fwrite()` does not change the contents of the array pointed to by *ptr*.

The argument *size* is typically *sizeof*(**ptr*) where the pseudo-function *sizeof* specifies the length of an item pointed to by *ptr*. If *ptr* points to a data type other than `char` it should be cast into a pointer to `char`.

SEE ALSO

read(2), write(2), fopen(3S), getc(3S), gets(3S), printf(3S), putc(3S), puts(3S), scanf(3S).

RETURN VALUE

`fread()` and `fwrite()` return the number of items read or written. If *size* or *nitems* is non-positive, no characters are read or written and 0 is returned by both `fread()` and `fwrite()`.

STANDARDS CONFORMANCE

`fread()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`fwrite()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

frexp(), ldexp(), modf() - split floating-point into mantissa and exponent

SYNOPSIS

DESCRIPTION

Every non-zero number can be written uniquely as $x * 2^n$, where the "mantissa" (fraction) x is in the range $0.5 \leq | x | < 1.0$, and the "exponent" n is an integer.

frexp() returns the mantissa of a double *value*, and stores the exponent indirectly in the location pointed to by *eptr*. If *value* is zero, both results returned by *frexp* are zero.

ldexp() returns the quantity $value * 2^{exp}$.

modf() returns the signed fractional part of *value* and stores the integral part indirectly in the location pointed to by *iptr*.

DIAGNOSTICS

If **ldexp()** would cause overflow, ±**HUGE** is returned (according to the sign of *value*), and **errno** is set to ERANGE.

If **ldexp()** would cause underflow, zero is returned and **errno** is set to ERANGE.

STANDARDS CONFORMANCE

frexp(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

ldexp(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

modf(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

fseek(), rewind(), ftell() - reposition a file pointer in a stream

SYNOPSIS

```
#include <stdio.h>

int fseek(FILE *stream, long int offset, int whence);

void rewind(FILE *stream);

long int ftell(FILE *stream);
```

DESCRIPTION

fseek() sets the file-position indicator for *stream*. The new position, measured in bytes from the beginning of the file, is obtained by adding *offset* to the position specified by *whence*. The specified position is the beginning of the file for **SEEK_SET**, the current position for **SEEK_CUR**, or end-of-file for **SEEK_END**.

If the most recent operation, other than **ftell()** , on the *stream* is *fflush()*, the file offset in the underlying open file description is adjusted to reflect the location specified by the **fseek()** .

rewind(*stream***)** is equivalent to **fseek (stream, 0L, SEEK_SET)**, except that no value is returned.

fseek() and **rewind()** undo any effects of *ungetc*(3S).

After **fseek()** or **rewind()**, the next operation on a file opened for update can be either input or output. **fseek()** clears the EOF indicator for the *stream*. **rewind()** does an implicit **clearerr()** call (see *ferror*(3S)).

ftell() returns the offset of the current byte relative to the beginning of the file associated with the named *stream*.

RETURN VALUE

fseek() returns zero if it succeeds. Otherwise it returns –1 and sets **errno** to indicate the error.

ftell() returns the current value of the file position indicator for the stream measured in bytes from the beginning of the file. Otherwise, **ftell()** returns –1 and sets **errno** to indicate the error.

rewind() does not return a value. Therefore, any application that needs to detect errors should clear **errno** before calling **rewind()**. Then, upon completion, if **errno** is non-zero, it should assume an error has occurred.

ERRORS

fseek(), **ftell()**, and **rewind()** fail if the *stream* is unbuffered or the buffered data needs to be flushed, or if any of the following conditions are encountered:

[EAGAIN]	The O_NONBLOCK flag is set for the file descriptor and the process would be delayed in the write operation.
[EBADF]	The underlying file is not open for writing.
[EFBIG]	An attempt was made to write a file that exceeds the process's file size limit or the maximum file size. See *ulimit*(2).
[EINTR]	A signal was caught during the write operation.
[EIO]	The process is in a background process group and is attempting to write to its controlling terminal, **TOSTOP** is set, the process is neither ignoring nor blocking the **SIGTTOU** signal, and the process group of the process is orphaned.
[ENOSPC]	There was no free space remaining on the device containing the file.
[EPIPE]	An attempt was made to write to a pipe that is not open for reading by any process. A **SIGPIPE** signal is also sent to the process.
[ESPIPE]	A seek operation was attempted and the file descriptor underlying *stream* is associated with a pipe.

fseek() also fails if:

[EINVAL] The *whence* argument is invalid, or the file-position indicator would be set to a negative value.

Additional **errno** values may be set by the underlying **write()** and **lseek()** functions (see *write*(2) and *lseek*(2)).

WARNINGS

On HP-UX systems, the offset returned by **ftell()** is measured in bytes and it is permissible to seek to positions relative to that offset. However, when porting to non-HP-UX systems, **fseek()** should be used directly without relying on any offset obtained from **ftell()** because arithmetic cannot meaningfully be performed on such an offset if it is not measured in bytes on a particular operating system.

fseek() and **rewind()** have no effect on streams that have been opened in append mode (see *fopen*(3S)).

SEE ALSO

lseek(2), write(2), ferror(3S), fopen(3S), fgetpos(3S), ungetc(3S).

STANDARDS CONFORMANCE

fseek(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

ftell(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

rewind(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

ftw, ftwh, nftw, nftwh – walk a file tree

SYNOPSIS

```
#include <ftw.h>

int ftw (char *path, int (*fn)(),

int ftwh (char *path, int (*fn)(),

int nftw (char *path, int (*fn)(),

int nftwh (char *path, int (*fn)(),
```

DESCRIPTION

`ftw()` recursively descends the directory hierarchy rooted in *path*. For each object in the hierarchy, `ftw()` calls *fn*, passing it a pointer to a null-terminated character string containing the name of the object, a pointer to a `stat` structure containing information about the object (see *stat*(2)), and an integer. Possible values of the integer, defined in the <`ftw.h`> header file, are:

`FTW_F`	The object is a file.
`FTW_D`	The object is a directory,
`FTW_DNR`	The object is a directory without read permission. *fn* will not be called for any of its descendants.
`FTW_NS`	`stat()` could not successfully be executed on the object. The contents of the `stat` structure is undefined. If the `stat()` failure is because the object is in a directory without search permission, *fn* is called and the walk continues. If `stat()` fails for any other reason, `ftw()` does not call *fn*, sets `errno`, and returns –1.

Tree traversal continues until the tree is exhausted, an invocation of *fn* returns a nonzero value, or some error is detected within `ftw()` (such as an I/O error). If the tree is exhausted, `ftw()` returns zero. If *fn* returns a non-zero value, `ftw()` stops its tree traversal and returns whatever value was returned by *fn*. If `ftw()` detects an error, it returns –1 and sets the error type in `errno`.

`ftw()` visits a directory before visiting any of its descendants.

`ftw()`, `ftwh()`, `nftw()`, and `nftwh()` use one file descriptor for each level in the tree. The *depth* argument limits the number of file descriptors that can be used. If *depth* is 0 or negative, the effect is the same as if it were 1. *depth* must not be greater than the number of file descriptors currently available for use. For best performance, *depth* should be at least as large as the number of levels in the tree.

`ftwh()` is eqivalent to `ftw()` except that `ftwh()` also traverses hidden directories (context dependent files — see *cdf(4)*).

`nftw()` is similar to `ftw()` except that it takes the additional argument *flags*. The *flags* field is the inclusive OR of the following values, as defined in the <`ftw.h`> header file:

`FTW_PHYS`	`nftw()` does a physical walk. It does not follow symbolic links. `nftw()` follows hard links but does not walk down any path that crosses itself. If `FTW_PHYS` is not specified `nftw()` follows symbolic and hard links but does not walk a path that crosses itself.
`FTW_MOUNT`	The walk does not cross a mount point. This means the walk does not visit any files that reside on a device other than the one where the walk started. It does not cross NFS mount points.
`FTW_DEPTH`	`nftw()` performs a depth-first search. This means that a directory's contents are visited before the directory itself is visited.
`FTW_CHDIR`	The walk does a `chdir()` (see *chdir*(2)) to each directory before reading it.
`FTW_CDF`	The walk traverses hidden directories (context dependent files — see *cdf(4)*).
`FTW_SERR`	The walk normally terminates and returns –1 if `stat()` fails for any reason. If `FTW_SERR` is specified and a `stat()` failure is encountered, *fn* is called, and the walk continues.

`nftw()` calls *fn* with four arguments for each file and directory visited. The first argument is the pathname of the file or directory, the second is a pointer to a `stat` structure (see *stat(2)*) containing information about the object, and the third is an integer giving additional information as follows:

`FTW_F`	The object is a file.
`FTW_D`	The object is a directory.
`FTW_DP`	The object is a directory and subdirectories have been visited. This can be passed to *fn* only if `FTW_DEPTH` is specified.
`FTW_SL`	The object is a symbolic link. This can be passed to *fn* only if `FTW_PHYS` is specified.
`FTW_DNR`	The object is a directory that cannot be read. *fn* is not called for any of its descendants.
`FTW_NS`	`stat()` failed on the object. The contents of the `stat` structure passed to *fn* are undefined. If the `stat()` failure occured because the object is in a directory without search permission, `errno` is set, and `nftw()` returns –1 after calling *fn*. Note that this behavior differs from `ftw()`. If `stat()` fails for any other reason, `nftw()` does not call *fn*, sets `errno`, and returns –1. This behavior can be altered by specifing the `FTW_SERR` *flag*.

The fourth argument is a structure `FTW` which contains the following members:

```
int base;
int level;
```

The value of *base* is the offset from the first character in the pathname to where the basename of the object starts; this pathname is passed as the first argument to *fn*. The value of *level* indicates depth relative to the start of the walk, where the start level has a value of zero.

`nftwh()` is equivalent to `nftw()` except that `nftwh()` also traverses hidden directories (context dependent files — see *cdf*(4)). `nftwh()` is equivalent to calling `nftw()` with the `FTW_CDF` *flag* specified.

ERRORS

`ftw()`, `ftwh()`, `nftw()`, and `nftwh()` fail if any of the following conditions are encountered:

[EACCES]	If a component of the *path* prefix denies search permission or read permission is denied for *path,* and *fn* returns –1 and does not reset `errno`.
[ENAMETOOLONG]	The length of the specified path name exceeds `PATH_MAX` bytes, or the length of a component of the path name exceeds `NAME_MAX` bytes while `_POSIX_NO_TRUNC` is in effect.
[ENOENT]	*path* points to the name of a file that does not exist, or points to an empty string.
[ENOTDIR]	A component of *path* is not a directory.
[EINVAL]	The value of the *depth* argument is invalid.

In addition, if the function pointed to by *fn* encounters system errors, `errno` may be set accordingly.

WARNINGS

On Series 300, 400 and 700 systems, `ftw()` uses `lstat()` instead of `stat()` to get the structure containing information about the object (`ftw()` uses `stat()` on Series 800 systems). See *stat*(2).

Because these functions are recursive, it is possible for them to terminate with a memory fault when applied to very deep file structures.

`ftw()`, `ftwh()`, `nftw()`, and `nftwh()` use `malloc()` to allocate dynamic storage during their operation. If they are forcibly terminated (such as if `longjmp()` is executed by *fn* or an interrupt routine) the calling function will not have a chance to free that storage, causing it to remain allocated until the process terminates. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have *fn* return a nonzero value at its next invocation.

AUTHOR

`ftw()`, `ftwh()`, `nftw()`, and `nftwh` were developed by AT&T and HP.

SEE ALSO
stat(2), malloc(3C), cdf(4).

STANDARDS CONFORMANCE
`ftw()`: AES, SVID2, XPG2, XPG3, XPG4

NAME

gamma(), lgamma(), signgam() - log gamma function

SYNOPSIS

```
#include <math.h>

double gamma(double x);

double lgamma(double x);

extern int signgam;
```

DESCRIPTION

`gamma()` and `lgamma()` return `ln(|Γ(x)|)`, where $\Gamma(x)$ is defined as

$$\int_0^\infty e^{-t}t^{x-1}dt.$$

The sign of $\Gamma(x)$ is returned in the external integer `signgam`. The argument x must not be a non-positive integer. (`gamma()` is defined over the reals excluding the non-positive integers).

The following C program fragment can be used to calculate Γ:

```
if ((y = gamma(x)) > LN_MAXDOUBLE)
        error();
y = signgam * exp(y);
```

where if y is greater than `LN_MAXDOUBLE`, as defined in the `<values.h>` header file, `exp()` returns a range error (see *exp*(3M)).

ERRORS

/lib/libm.a

For non-positive integer arguments, `gamma()` and `lgamma()` return HUGE_VAL and set `errno` to EDOM. A message indicating SING error is printed on the standard error output.

If the correct value would overflow, `gamma()` and `lgamma()` return HUGE_VAL and set `errno` to ERANGE.

`gamma()` and `lgamma()` return NaN and set `errno` to EDOM when x is NaN, or return +INFINITY and set `errno` to EDOM when x is ±INFINITY. A message indicating DOMAIN error is printed on the standard error output.

These error-handling procedures can be changed by using the `matherr()` function (see *matherr*(3M)).

/lib/libM.a

No error messages are printed on the standard error output.

For non-positive integer arguments `gamma()` and `lgamma()` return HUGE_VAL and set `errno` to EDOM. A message indicating SING error is printed on the standard error output.

If the correct value would overflow, `gamma()` and `lgamma()` return HUGE_VAL and set `errno` to ERANGE.

`gamma()` and `lgamma()` return NaN and set `errno` to EDOM when x is NaN, or return +INFINITY and set `errno` to EDOM when x is ±INFINITY.

These error-handling procedures can be changed by using the `_matherr()` function (see *matherr*(3M)). Note that `_matherr()` is provided in order to assist in migrating programs from `libm.a` to `libM.a` and is *not* a part of XPG3, ANSI C, or POSIX.

SEE ALSO

exp(3M), isinf(3M), isnan(3M), matherr(3M), values(5).

STANDARDS CONFORMANCE

`gamma()` in libm.a: AES, SVID2, XPG2, XPG3
`gamma()` in libM.a: AES, XPG3, XPG4

`lgamma()` in libm.a: AES, XPG3
`lgamma()` in libM.a: AES, XPG3, XPG4

signgam in libm.a: AES, SVID2, XPG2, XPG3
signgam in libM.a: AES, XPG3, XPG4

NAME

getc(), getchar(), fgetc(), getw() - get character or word from a stream file

SYNOPSIS

```
#include <stdio.h>

int getc(FILE *stream);

int getchar(void);

int fgetc(FILE *stream);

int getw(FILE *stream);
```

DESCRIPTION

`getc()` Returns the next character (i.e., byte) from the named input *stream*, as an unsigned character converted to an integer. It also moves the file pointer, if defined, ahead one character in *stream*. `getchar()` is defined as `getc(stdin)`. `getc()` and `getchar()` are defined both as macros and as functions.

`fgetc()` Same as `getc()`, but is a function rather than a macro. `fgetc()` is slower than `getc()`, but it takes less space per invocation and its name can be passed as an argument to a function.

`getw()` returns the next word (i.e., `int` in C) from the named input *stream*. `getw()` increments the associated file pointer, if defined, to point to the next word. The size of a word is the size of an integer and varies from machine to machine. `getw()` assumes no special alignment in the file.

RETURN VALUE

Upon sucessful completion, `getc()`, `getchar()`, and `fgetc()` return the next byte from the input stream pointed to by *stream* (`stdin` for `getchar()`). If the stream is at end-of-file, the end-of-file indicator for the stream is set and EOF is returned. If a read error occurs, the error indicator for the stream is set, `errno` is set to indicate the error, and EOF is returned.

Upon sucessful completion, `getw()` returns the next word from the input stream pointed to by *stream*. If the stream is at end-of-file, the end-of-file indicator for the stream is set and `getw()` returns EOF. If a read error occurs, the error indicator for the stream is set, and `getw()` returns EOF and sets `errno` to indicate the error.

`ferror()` and `feof()` can be used to distinguish between an error condition and an end-of-file condition.

ERRORS

`getc()`, `getchar()`, `getw()`, and `fgetc()` fail if data needs to be read into the *stream*'s buffer, and:

[EAGAIN] The `O_NONBLOCK` flag is set for the file descriptor underlying *stream* and the process would be delayed in the read operation.

[EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for reading.

[EINTR] The read operation was terminated due to the receipt of a signal, and either no data was transferred or the implementation does not report partial transfer for this file.

[EIO] The process is a member of a background process and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the `SIGTTIN` signal or the process group of the process is orphaned.

Additional `errno` values may be set by the underlying `read()` function (see *read*(2)).

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), fread(3S), gets(3S), putc(3S), read(2), scanf(3S).

WARNING

`getc()` and `getchar()` are implemented both as library functions and macros. The macro versions, which are used by default, are defined in `<stdio.h>`. To obtain the library function either use a `#undef` to remove the macro definition or, if compiling in ANSI-C mode, enclose the function name in parenthesis or use the function address. The following example illustrates each of these methods :

```
#include <stdio.h>
#undef getc
    ...
main()
{
    int (*get_char()) ();
                ...
    return_val=getc(c,fd);
                ...
    return_val=(getc)(c,fd1);
                ...
    get_char = getchar;
};
```

If the integer value returned by `getc()`, `getchar()`, or `fgetc()` is stored into a character variable then compared against the integer constant **EOF**, the comparison may never succeed because sign-extension of a character on widening to integer is machine-dependent.

The macro version of `getc()` incorrectly treats a *stream* argument with side effects. In particular, `getc(*f++)` does not work sensibly. The function version of `getc()` or `fgetc()` should be used instead.

Because of possible differences in word length and byte ordering, files written using `putw()` are machine-dependent, and may be unreadable by `getw()` on a different processor.

STANDARDS CONFORMANCE

`getc()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`fgetc()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`getchar()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`getw()`: AES, SVID2, XPG2, XPG3, XPG4

NAME

getccent(), getcccid(), getccnam(), setccent(), endccent(), fgetccent() - get HP Cluster configuration entry

SYNOPSIS

```
#include <cluster.h>

struct cct_entry *getccent(void);

struct cct_entry *getcccid(cnode_t cid);

struct cct_entry *getccnam(const char *name);

void setccent(void);

void endccent(void);

struct cct_entry *fgetccent(FILE *stream);
```

DESCRIPTION

`getccent()`, `getcccid()`, and `getccnam()` each return a pointer to an object with the following structure containing the broken-out fields in the `/etc/clusterconf` file. The file contains a list of `cct_entry` structures, defined in the `<cluster.h>` header file. The `cct_entry` structure includes the following fields:

```
u_char machine_id[M_IDLEN]; /* Unique machine ID */
cnode_t cnode_id;           /* cnode ID */
char cnode_name[15];        /* cnode name */
char cnode_type;            /* 'r'=cluster server
                               'c'=all other cluster nodes */
cnode_t swap_serving_cnode; /* swap cnode */
int kcsp;                   /* default number of CSPs
                               to create see csp(1M) */
```

The constant `M_IDLEN` is defined in `<cluster.h>`.

`getccent()` When first called, `getccent()` opens the cluster configuration file `/etc/clusterconf` and returns a pointer to the first `cct_entry` structure in the file. Thereafter, it returns a pointer to the next `cct_entry` structure in the file. Successive calls can be used to search the entire file.

`getcccid()` Searches from the beginning of the file until an entry whose cnode ID matches *cid* is found, and returns a pointer to the particular structure in which it was found.

`getccnam()` Searches from the beginning of the file until a cnode name matching *name* is found and returns a pointer to the particular structure in which it was found. If an EOF or an error is encountered on reading, these functions return a NULL pointer.

`setccent()` Has the effect of rewinding the cluster configuration file to the beginning of the file to allow repeated searches.

`endccent()` Can be called to close the cluster configuration file when processing is complete.

`fgetccent()` Returns a pointer to the next `cct_entry` structure in the stream *stream*, which matches the format of `/etc/clusterconf`.

RETURN VALUE

A NULL pointer is returned on EOF or error.

WARNINGS

The above routines use `<stdio.h>`, which causes them to increase the size of programs not otherwise using standard I/O, more than might be expected.

All information is contained in a static area that is overwritten with each call; thus information must be copied if it is to be saved.

AUTHOR

`getccent()` was developed by HP.

FILES

`/etc/clusterconf`

SEE ALSO
csp(1M), clusterconf(4).

NAME

getcdf(), hidecdf() - manipulate CDF path names

SYNOPSIS

```
#include <unistd.h>

char *getcdf(const char *path, char *buf, size_t size);

char *hidecdf(const char *path, char *buf, size_t size);
```

DESCRIPTION

`getcdf()` and `hidecdf()` manipulate path names possibly containing CDF (hidden directory) components.

`getcdf()` Returns a pointer to the expanded path matching the path name in *path*. The *path* argument can be a context dependent file (CDF) in which case a path name with all hidden directories expanded is returned. If *path* is not a CDF, a copy of the original path name is returned.

`hidecdf()` Returns a pointer to the simplified path corresponding to *path*. Any context-dependent components in the original *path* that match the current context (see *context*(5)) are removed from the resulting path.

The value of *size* must be at least one greater than the length of the path name to be returned.

If *buf* is not a NULL pointer, `getcdf()` and `hidecdf()` copy the expanded path name into array *buf*. If *buf* is a NULL pointer, `getcdf()` and `hidecdf()` obtain *size* bytes of space using `malloc()` (see *malloc*(3C)). In this case, the pointer returned by `getcdf()` and `hidecdf()` can be used as an argument in a subsequent call to `free()` (see *malloc*(3C)).

RETURN VALUE

Upon successful completion, `getcdf()` and `hidecdf()` return a pointer to the resulting path name. Otherwise, a value of NULL is returned and `errno` is set to indicate the error.

ERRORS

If either `getcdf()` or `hidecdf()` fails, it will set `errno` to one of the following values:

[ENOENT]	A component of *path* does not exist.
[EACCES]	Read or search permission is denied for one of the directories given in *path*.
[ENAMETOOLONG]	*size* is not large enough to hold the resulting path.

EXAMPLES

```
#include <stdio.h>

char *path, *cdf, *getcdf();
int size;
        . . .
if ((cdf = getcdf(path, NULL, size)) == NULL) {
    perror("getcdf");
    exit(1);
}
printf("%s\n", cdf);
free(cdf);
```

AUTHOR

`getcdf()` and `hidecdf()` were developed by HP.

SEE ALSO

showcdf(1), malloc(3C), cdf(4), context(5).

NAME

getclock - get current value of system-wide clock

SYNOPSIS

```
#include <sys/timers.h>

int getclock(int clock_type, struct timespec *tp);
```

DESCRIPTION

`getclock()` gets the current value *tp* of the specified system-wide clock, *clock_type*.

`getclock()` supports a *clock_type* of `TIMEOFDAY`, defined in `<sys/timers.h>` which clock represents the time-of-day clock for the system. For this clock, the values returned by `getclock()` represent the amount of time since the Epoch.

RETURN VALUE

`getclock()` returns a value of zero if successful; otherwise it returns a value of −1 and sets `errno` to indicate the error.

ERRORS

getclock() fails if any of the following conditions are encountered:

[EINVAL]	*clock_type* does not specify a known system-wide clock.
[EIO]	An error occurred while accessing the clock device.

SEE ALSO

gettimer(3C), setclock(3C), <sys/timers.h>.

STANDARDS CONFORMANCE

`getclock()`: AES

NAME

getcwd(), gethcwd() - get pathname of current working directory

SYNOPSIS

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);

char *gethcwd(char *buf, size_t size);
```

DESCRIPTION

`getcwd()` places the absolute pathname of the current working directory in the array pointed to by *buf*, and returns *buf*. The value of *size* must be at least one greater than the length of the pathname to be returned.

If *buf* is a NULL pointer, `getcwd()` obtains *size* bytes of space using `malloc()` (see *malloc*(3C)). In this case, the pointer returned by `getcwd()` can be used as the argument in a subsequent call to `free()` (see *malloc*(3C)). Invoking `getcwd()` with *buf* as a null pointer is not recommended because this functionality may be removed from the HP-UX operating system in a future release.

`gethcwd()` works the same as `getcwd()` except the returned directory pathname lists all hidden directories (context dependent files (see *cdf*(4)).

RETURN VALUE

Upon successful completion, `getcwd()` returns a pointer to the current directory pathname. Otherwise, it returns NULL with `errno` set if *size* is not large enough, or if an error occurs in a lower-level function.

ERRORS

`getcwd()` fails if any of the following conditions are encountered:

[EINVAL]	The *size* argument is zero or negative.
[ERANGE]	The *size* argument is greater than zero, but is smaller than the length of the pathname.
[ENAMETOOLONG]	The length of the specified path name exceeds **PATH_MAX** bytes, or the length of a component of the path name exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect.

`getcwd()` may fail if any of the following conditions are encountered:

[EACCES]	Read or search permission is denied for a component of pathname.
[EFAULT]	*buf* points outside the allocated address space of the process. `getcwd()` may not always detect this error.
[ENOMEM]	`malloc()` failed to provide *size* bytes of memory.

EXAMPLES

```
char *cwd, *getcwd();
char buf[PATH_MAX+1];
                . . .
if ((cwd = getcwd((buf *)NULL, PATH_MAX+1)) == NULL) {
    perror("pwd");
    exit(1);
}
puts(cwd);
```

AUTHOR

`getcwd()` was developed by AT&T. `gethcwd()` was developed by HP.

SEE ALSO

pwd(1), malloc(3C), cdf(4).

STANDARDS CONFORMANCE

`getcwd()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

getdate() - convert user format date and time

SYNOPSIS

```
#include <time.h>

struct tm *getdate(const char *string);

extern int getdate_err;
```

DESCRIPTION

getdate() converts user definable date and/or time specifications pointed to by *string* into a **struct tm**. The structure declaration is in the **<time.h>** header file (see *ctime*(3C)).

User-supplied templates are used to parse and interpret the input string. The templates are text files created by the user and identified via the environment variable **DATEMSK**. **DATEMSK** should be set to indicate the full pathname of the template file. The first line in the template that matches the input specification is used for interpretation and conversion into the internal time format. Upon successful completion, **getdate()** returns a pointer to a **struct tm**; otherwise, it returns NULL and the external variable **getdate_err** is set to indicate the error.

The following field descriptors are supported:

%%	same as **%**
%a	abbreviated weekday name
%A	full weekday name
%b	abbreviated month name
%B	full month name
%c	locale's appropriate date and time representation
%d	day of the month (01 through 31; the leading 0 is optional)
%e	same as **%d**
%D	date as **%m/%d/%y**
%h	abbreviated month name
%H	hour (00 through 23)
%I	hour (01 through 12)
%m	month number (01 through 12)
%M	minute (00 through 59)
%n	same as **\n**
%p	locale's equivalent of either AM or PM
%r	time as **%I:%M:%S %p**
%R	time as **%H:%M**
%S	seconds (00 through 59)
%t	insert a tab
%T	time as **%H:%M:%S**
%w	weekday number (Sunday = 0 through Saturday = 6)
%x	locale's appropriate date representation
%X	locale's appropriate time representation
%y	year without century (00 through 99)
%Y	year as *ccyy* (e.g., 1986)
%Z	time zone name or no characters if no time zone exists

Month and weekday names can consist of any combination of uppercase and lowercase letters. The user can request that the input date or time specification be in a specific language by setting the **LC_TIME** category (see *setlocale*(3C)).

For descriptors that allow leading zeros, leading zeros are optional; not required. However, the number of digits used for those descriptors must not exceed two, including leading zeros. Extra whitespace in either the template file or in *string* is ignored.

The field descriptors **%c**, **%x**, and **%X** are not supported if they include unsupported field descriptors.

The following example shows the possible contents of a template:

```
%m
%A %B %d, %Y, %H:%M:%S
%A
```

```
%B
%m/%d/%y %I %p
%d,%m,%Y %H:%M
at %A the %dst of %B in %Y
run job at %I %p, %B %dnd
%A den %d. %B %Y %H.%M Uhr
```

The following are examples of valid input specifications for the above template:

```
getdate("10/1/87 4 PM");
getdate("Friday");
getdate("Friday September 18, 1987, 10:30:30");
getdate("24,9,1986 10:30");
getdate("at monday the 1st of december in 1986");
getdate("run job at 3 PM, december 2nd");
```

If the `LC_TIME` category is set to a German locale that includes `freitag` as a weekday name and `oktober` as a month name, the following would be valid:

```
getdate("freitag den 10. oktober 1986 10.30 Uhr");
```

This example shows how local date and time specification can be defined in the template:

Invocation	Line in Template
getdate("11/27/86")	**%m/%d/%y**
`getdate("27.11.86")`	`%d.%m.%y`
`getdate("86-11-27")`	`%y-%m-%d`
`getdate("Friday 12:00:00")`	`%A %H:%M:%S`

The following rules apply when converting the input specification into the internal format:

- If only the weekday is given, today is assumed if the given day is equal to the current day, and next week if it is less.
- If only the month is given, the current month is assumed if the given month is equal to the current month, and next year if it is less and no year is given (the first day of the month is assumed if no day is given).
- If no hour, minute and second are given, the current hour, minute and second are assumed.
- If no date is given, today is assumed if the given hour is greater than the current hour and tomorrow is assumed if it is less.

The following examples help to illustrate the above rules assuming that the current date is `Mon Sep 22 12:19:47 EDT 1986`, and the `LC_TIME` category is set to the default "C" locale.

Input	Line in Template	Date
`Mon`	`%a`	Mon Sep 22 12:19:47 EDT 1986
`Sun`	`%a`	Sun Sep 28 12:19:47 EDT 1986
`Fri`	`%a`	Fri Sep 26 12:19:47 EDT 1986
`September`	`%B`	Mon Sep 1 12:19:47 EDT 1986
`January`	`%B`	Thu Jan 1 12:19:47 EST 1987
`December`	`%B`	Mon Dec 1 12:19:47 EST 1986
`Sep Mon`	`%b %a`	Mon Sep 1 12:19:47 EDT 1986
`Jan Fri`	`%b %a`	Fri Jan 2 12:19:47 EST 1987
`Dec Mon`	`%b %a`	Mon Dec 1 12:19:47 EST 1986
`Jan Wed 1989`	`%b %a %Y`	Wed Jan 4 12:19:47 EST 1989
`Fri 9`	`%a %H`	Fri Sep 26 09:00:00 EDT 1986
`Feb 10:30`	`%b %H:%S`	Sun Feb 1 10:30:00 EST 1987
`10:30`	`%H:%M`	Tue Sep 23 10:30:00 EDT 1986
`13:30`	`%H:%M`	Mon Sep 22 13:30:00 EDT 1986

ERRORS

Upon failure, `getdate()` returns NULL and the variable `getdate_err` is set to indicate the error.

The following is a complete list of the `getdate_err` settings and their interpretation:

1 the **DATEMSK** environment variable is null or undefined,
2 the template file cannot be opened for reading,
3 failed to get file status information,
4 the template file is not a regular file,
5 an error is encountered while reading the template file,
6 memory allocation failed (not enough memory available),
7 there is no line in the template that matches the input,
8 invalid input specification (example: February 31).

SEE ALSO

ctime(3C), ctype(3C), setlocale(3C), strftime(3C).

NAME

getdiskbyname() - get disk description by its name

SYNOPSIS

```
#include <disktab.h>

struct disktab *getdiskbyname(const char *name);
```

DESCRIPTION

getdiskbyname() takes a disk name (such as hp7959B) and returns a pointer to a structure that describes its geometry information and the standard disk partition tables. All information is obtained from the disktab database file (see *disktab*(4)).

The contents of the structure **disktab** include the following members. Note that there is not necessarily any correlation between the placement in this list and the order in the structure.

```
char    *d_name;            /* drive name */
char    *d_type;            /* drive type */
int     d_secsize;          /* sector size in bytes */
int     d_ntracks;          /* # tracks/cylinder */
int     d_nsectors;         /* # sectors/track */
int     d_ncylinders;       /* # cylinders */
int     d_rpm;              /* revolutions/minute */
struct  partition {

        int     p_size;     /* #sectors in partition */
        short   p_bsize;    /* block size in bytes */
        short   p_fsize;    /* frag size in bytes */

} d_partitions[NSECTIONS];
```

The constant **NSECTIONS** is defined in <**disktab.h**>.

DIAGNOSTICS

A NULL pointer is returned in case of an error, or if *name* is not found in the disktab database file.

AUTHOR

getdiskbyname() was developed by HP and the University of California, Berkeley.

SEE ALSO

disktab(4)

NAME

getenv( ) - return value for environment name

SYNOPSIS

```
#include <stdlib.h>

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

DESCRIPTION

`getenv( )` searches the environment list (see *environ*(5)) for a string of the form *name=value*, and returns a pointer to the *value* in the current environment if such a string is present, otherwise a NULL pointer. *name* can be either the desired name, null-terminated, or of the form *name=value*, in which case `getenv( )` uses the portion to the left of the = as the search key.

WARNINGS

`getenv( )` returns a pointer to static data which can be overwritten by subsequent calls.

SEE ALSO

exec(2), putenv(3C), environ(5).

EXTERNAL INFLUENCES

Locale

The `LC_CTYPE` category determines the interpretation of characters in *name* as single- and/or multi-byte characters.

International Code Set Support

Single- and multi-byte character code sets are supported.

STANDARDS CONFORMANCE

`getenv( )`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

getfsent(), getfsspec(), getfsfile(), getfstype(), setfsent(), endfsent() - get file system descriptor file entry

SYNOPSIS

```
#include <checklist.h>

struct checklist *getfsent(void);

struct checklist *getfsspec(const char *spec);

struct checklist *getfsfile(const char *file);

struct checklist *getfstype(const char *type);

int setfsent(void);

int endfsent(void);
```

Remarks:

These routines are included only for compatibility with 4.2 BSD. For maximum portability and improved functionality, new applications should use the *getmntent*(3X) library routines.

DESCRIPTION

`getfsent()`, `getfsspec()`, `getfsfile()`, and `getfstype()` each returns a pointer to an object with the following structure containing the broken-out fields of a line in the `/etc/checklist` file. The structure is declared in the `<checklist.h>` header file:

```
struct checklist {
    char    *fs_spec;     /* special file name */
    char    *fs_bspec;    /* block special file name */
    char    *fs_dir;      /* file sys directory name */
    char    *fs_type;     /* type: ro, rw, sw, xx */
    int     fs_passno;    /* fsck pass number */
    int     fs_freq;      /* backup frequency */
};
```

The fields have meanings described in *checklist*(4). If the block special file name, the file system directory name, and the type are not all defined on the associated line in `/etc/checklist`, these routines return pointers to NULL in the `fs_bspec`, `fs_dir`, and `fs_type` fields. If the pass number or the backup frequency field are not present on the line, these routines return –1 in the corresponding structure member. `fs_freq` is reserved for future use.

`getfsent()` Reads the next line of the file, opening the file if necessary.

`setfsent()` Opens and rewinds the file.

`endfsent()` Closes the file.

`getfsspec()` Sequentially searches from beginning of file until a matching special file name is found, or until EOF is encountered.

`getfsfile()` Sequentially searches from the beginning of the file until a matching file system file name is found, or until EOF is encountered. `getfstype()` Sequentially searches from the beginning of the file until a matching file system type field is found, or until EOF is encountered.

DIAGNOSTICS

A null pointer is returned on EOF, invalid entry, or error.

WARNINGS

Since all information is contained in a static area, it must be copied to be saved.

AUTHOR

`getfsent()` was developed by HP and the University of California, Berkeley.

FILES

`/etc/checklist`

SEE ALSO

checklist(4).

NAME

getgrent(), getgrgid(), getgrnam(), setgrent(), endgrent(), fgetgrent() - get group file entry

SYNOPSIS

```
#include <grp.h>

struct group *getgrent(void);

struct group *getgrgid(gid_t gid);

struct group *getgrnam(const char *name);

void setgrent(void);

void endgrent(void);

struct group *fgetgrent(FILE *stream);
```

DESCRIPTION

`getgrent()`, `getgrgid()`, and `getgrnam()` locate an entry in the `/etc/group` file, and return a pointer to an object of type `struct group`.

The `group` structure is defined in `<grp.h>` and includes the following members:

```
char  *gr_name;     /* the name of the group */
char  *gr_passwd;   /* the encrypted group password */
gid_t gr_gid;       /* the numerical group ID */
char  **gr_mem;     /* null-terminated array of pointers
                       to member names */
```

`getgrent()` When first called, `getgrent()` returns a pointer to the first `group` structure in the file; thereafter, it returns a pointer to the next `group` structure in the file. In this way, successive calls can be used to search the entire file. `getgrent()` opens the `/etc/group` file prior to doing its work and leaves the file open afterward;

`setgrent()` Has the effect of rewinding this file to allow repeated searches;

`endgrent()` Can be called to close the file when processing is complete.

`getgrgid()` Searches from the beginning of the file until a numeric group ID matching *gid* is found, and returns a pointer to the particular structure in which it was found.

`getgrnam()` Searches from the beginning of the file until a group name matching *name* is found, and returns a pointer to the particular structure in which it was found.

`fgetgrent()` Returns a pointer to the next `group` structure in the standard I/O stream *stream*, which should be open for reading, and its contents should match the format of `/etc/group`.

NETWORKING FEATURES

NFS

If an entry beginning with a plus sign (+) or a minus sign (−) is found, these routines try to use the Network Information Service network database for data. See *group*(4) for proper syntax and operation.

RETURN VALUE

`getgrent()`, `getgrgid()`, `getgrnam()`, and `fgetgrent()` return a NULL pointer if an end-of-file or error is encountered on reading. Otherwise, the return value points to an internal static area containing a valid `group` structure.

WARNINGS

The above routines use `<stdio.h>` and the Network Information Service library. This causes them to increase the size of programs that do not otherwise use standard I/O and Network Information Service more than might ordinarily be expected.

The value returned by these functions points to a single static area that is overwritten by each call to any of the functions. It must be copied if it is to be saved.

DEPENDENCIES

NFS:

FILES

/etc/yp/domainname/group.byname
/etc/yp/domainname/group.bygid

SEE ALSO:
ypcat(1).

FILES
/etc/group

SEE ALSO
getgroups(2), getpwent(3C), stdio(3S), group(4).

STANDARDS CONFORMANCE
getgrent(): SVID2, XPG2
endgrent(): SVID2, XPG2

fgetgrent(): SVID2, XPG2
getgrgid(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

getgrnam(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1
setgrent(): SVID2, XPG2

NAME

gethostent(), gethostbyaddr(), gethostbyname(), sethostent(), endhostent() - get network host entry

SYNOPSIS

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

extern int h_errno;

struct hostent *gethostent(void);

struct hostent *gethostbyname(const char *name);

struct hostent *gethostbyaddr(
    const char *addr,
    int len,
    int type);

int sethostent(int stayopen);

int endhostent(void);
```

DESCRIPTION

`gethostent()`, `gethostbyname()`, and `gethostbyaddr()` each return a pointer to a structure of type `hostent`, defined as follows in `<netdb.h>`:

```
struct hostent {
    char    *h_name;
    char    **h_aliases;
    int     h_addrtype;
    int     h_length;
    char    **h_addr_list;
};
#define h_addr  h_addr_list[0]
```

The members of this structure are:

`h_name`	The official name of the host.
`h_aliases`	A null-terminated array of alternate names for the host.
`h_addrtype`	The type of address being returned; always **AF_INET**.
`h_length`	The length, in bytes, of the address.
`h_addr_list`	A null-terminated array of network addresses for the host.
`h_addr`	The first address in h_addr_list; this is for compatibility with previous HP-UX implementations where a *struct hostent* contains only one network address per host.

Name Server Operation

If the local system is configured to use the **named** name server (see *named*(1M)):

`gethostent()`	Always returns a NULL pointer.
`sethostent()`,	Requests the use of a connected stream socket for queries to the name server if the *stayopen* flag is non-zero. The connection is retained after each call to `gethostbyname()` or `gethostbyaddr()`.
`endhostent()`	Closes the stream socket connection.
`gethostbyname()` `gethostbyaddr()`	Each retrieves host information from the name server. Names are matched without respect to uppercase or lowercase. For example, `berkeley.edu`, `Berkeley.EDU` , and `BERKELEY.EDU` all match the entry for `berkeley.edu`.

NIS Server Operation

If the local system is not configured to use the name server, but is running **ypserv**, the Network

Information Service server:

`gethostent()`	Returns the next entry in the NIS database.
`sethostent()`	Initializes an internal key for the NIS database. If the *stayopen* flag is non-zero, the internal key is not cleared after calls to `endhostent()`.
`endhostent()`	Clears the internal NIS database key.
`gethostbyname()` `gethostbyaddr()`	Each retrieves host information from the NIS database. Names are matched without respect to uppercase or lowercase. For example, `berkeley.edu`, `Berkeley.EDU` , and `BERKELEY.EDU` all match the entry for `berkeley.edu`.

Non-Server Operation

If the local system is using neither the local name server nor the Network Information Service server:

`gethostent()`	Reads the next line of `/etc/hosts`, opening the file if necessary.
`sethostent()`	opens and rewinds the file. If the *stayopen* flag is non-zero, the host data base is not closed after each call to `gethostent()` (either directly or indirectly through one of the other `gethost` calls).
`endhostent()`	Closes the file.
`gethostbyname()`	Sequentially searches from the beginning of the file until a host name (among either the official names or the aliases) matching its *name* parameter is found, or until EOF is encountered. Names are matched without respect to uppercase or lowercase, as described above in the name server case.
`gethostbyaddr()`	Sequentially searches from the beginning of the file until an Internet address matching its *addr* parameter is found, or until EOF is encountered.

In calls to `gethostbyaddr()`, the parameter *addr* must point to an Internet address in network order (see *byteorder*(3N)). The parameter *len* must be the number of bytes in an Internet address; that is, `sizeof (struct in_addr)`. The parameter *type* must be the constant `AF_INET`.

RETURN VALUE

If successful, `gethostbyname()`, `gethostbyaddr()` and `gethostent()` return a pointer to the requested hostent struct. `gethostbyname()` and `gethostbyaddr()` return NULL if their *host* or *addr* parameters, respectively, cannot be found in the database. If `/etc/hosts` is being used, they also return NULL if they are unable to open `/etc/hosts`. `gethostbyaddr()` also returns NULL if either its *addr* or *len* parameter is invalid. `gethostent()` always returns NULL if the name server is being used.

ERRORS

If the name server is being used and `gethostbyname()` or `gethostbyaddr()` returns a NULL pointer, the external integer `h_errno` contains one of the following values:

`HOST_NOT_FOUND`	No such host is known.
`TRY_AGAIN`	This is usually a temporary error. The local server did not receive a response from an authoritative server. A retry at some later time may succeed.
`NO_RECOVERY`	This is a non-recoverable error.
`NO_ADDRESS`	The requested name is valid but does not have an IP address; this is not a temporary error. This means another type of request to the name server will result in an answer.

If the name server is not being used, the value of `h_errno` may not be meaningful.

WARNINGS

All information is contained in a static area so it must be copied if it is to be saved.

AUTHOR

`gethostent()` was developed by the University of California, Berkeley.

FILES

`/etc/hosts`

SEE ALSO

named(1M), ypserv(1M), resolver(3N), ypclnt(3C), hosts(4), ypfiles(4).

NAME

getlogin() - get login name

SYNOPSIS

```
#include <unistd.h>

char *getlogin(void);
```

DESCRIPTION

`getlogin()` returns a pointer to the login name as found in `/etc/utmp`. It can be used in conjunction with `getpwnam()` to locate the correct password file entry when the same user ID is shared by several login names.

If `getlogin()` is called within a process that is not attached to a terminal, it returns a NULL pointer. The recommended procedure to obtain the user name associated with the real user ID of the calling process is to call `getlogin()`, and if that fails to call `getpwuid()`. The function `cuserid()` can be used to obtain the user name associated with the effective user ID of the calling process.

ERRORS

`getlogin()` fails if any of the following is true:

[EBADF]	An invalid file descriptor was obtained.
[EMFILE]	Too many file descriptors are in use by this process.
[ENFILE]	The system file table is full.

FILES

`/etc/utmp`

SEE ALSO

getgrent(3C), getpwent(3C), cuserid(3S), utmp(4).

DIAGNOSTICS

`getlogin()` returns the NULL pointer if *name* is not found.

WARNINGS

Return values point to static data whose content is overwritten by each call.

STANDARDS CONFORMANCE

`getlogin()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

getmntent(), setmntent(), addmntent(), endmntent(), hasmntopt() - get file system descriptor file entry

SYNOPSIS

```
#include <mntent.h>

FILE *setmntent(const char *path, char *type);

struct mntent *getmntent(FILE *stream);

int addmntent(FILE *stream, struct mntent *mnt);

char *hasmntopt(struct mntent *mnt, const char *opt);

int endmntent(FILE *stream);
```

DESCRIPTION

These routines replace the obsolete `getfsent()` routines (see *getfsent*(3X)) for accessing the file system description file `/etc/checklist`. They are also used to access the mounted file system description file `/etc/mnttab`.

`setmntent()` Opens a file system description file and returns a file pointer which can then be used with `getmntent()`, `addmntent()`, or `endmntent()`. The *type* argument is the same as in *fopen*(3C).

`getmntent()` Reads the next line from *stream* and returns a pointer to an object with the following structure containing the broken-out fields of a line in the file-system description file, `<mntent.h>`. The fields have meanings described in *checklist*(4).

```
struct mntent {
    char    *mnt_fsname; /* file system name */
    char    *mnt_dir;    /* file system path prefix */
    char    *mnt_type;   /* hfs, nfs, swap, or xx */
    char    *mnt_opts;   /* ro, suid, etc. */
    int     mnt_freq;    /* dump frequency, in days */
    int     mnt_passno;  /* pass number on parallel fsck */
    long    mnt_time;    /* When file system was mounted; */
                         /* see mnttab(4). */
    cnode_t mnt_cnode;   /* Cnode id from stat of mnt_fsname */
                         /* (0 for NFS) */
};
```

In the HP Clustered environment, the `mnt_cnode` field contains the cnode ID associated with the file system name named in the `mnt_fsname` field unless the specified file system is of NFS type in which case the `mnt_cnode` field is set to 0. `getmntent()` obtains the `mnt_cnode` field for non-NFS type file systems by executing the `stat()` system call and using the `st_rcnode` field of the stat structure (see *stat*(2)).

`addmntent()` Adds the `mntent` structure *mnt* to the end of the open file *stream*. Note that *stream* must be opened for writing.

`hasmntopt()` Scans the `mnt_opts` field of the `mntent` structure *mnt* for a substring that matches *opt*. It returns the address of the substring if a match is found; `0` otherwise.

`endmntent()` Closes the file.

The following definitions are provided in `<mntent.h>`:

```
#define MNT_CHECKLIST   "/etc/checklist"
#define MNT_MNTTAB      "/etc/mnttab"

#define MNTMAXSTR       128         /* Max size string in mntent */

#define MNTTYPE_HFS     "hfs"       /* HFS file system */
#define MNTTYPE_CDFS    "hfs"       /* CD-ROM file system */
#define MNTTYPE_NFS     "nfs"       /* Network file system */
```

```
#define MNTTYPE_SWAP    "swap"      /* Swap device */
#define MNTTYPE_SWAPFS  "swapfs"    /* File system swap */
#define MNTTYPE_IGNORE  "ignore"    /* Ignore this entry */

#define MNTOPT_DEFAULTS "defaults"  /* Use all default options */
#define MNTOPT_RO       "ro"        /* Read only */
#define MNTOPT_RW       "rw"        /* Read/write */
#define MNTOPT_SUID     "suid"      /* Set uid allowed */
#define MNTOPT_NOSUID   "nosuid"    /* No set uid allowed */
#define MNTOPT_QUOTA    "quota"     /* Enable disk quotas */
#define MNTOPT_NOQUOTA  "noquota"   /* Disable disk quotas */
```

The following definition is provided for device swap in <mntent.h>:

```
#define MNTOPT_END    "end"      /* swap after end of file system,
                                    Series 300/400/700 only */
```

The following definitions are provided for file system swap in <mntent.h>:

```
#define MNTOPT_MIN    "min"      /* minimum file system swap */
#define MNTOPT_LIM    "lim"      /* maximum file system swap */
#define MNTOPT_RES    "res"      /* reserve space for file system */
#define MNTOPT_PRI    "pri"      /* file system swap priority */
```

NETWORKING FEATURES

NFS

The following definitions are provided in <mntent.h>:

```
#define MNTOPT_BG       "bg"        /* Retry mount in background */
#define MNTOPT_FG       "fg"        /* Retry mount in foreground */
#define MNTOPT_RETRY    "retry"     /* Number of retries allowed */
#define MNTOPT_RSIZE    "rsize"     /* Read buffer size in bytes */
#define MNTOPT_WSIZE    "wsize"     /* Write buffer size in bytes*/
#define MNTOPT_TIMEO    "timeo"     /* Timeout in 1/10 seconds */
#define MNTOPT_RETRANS  "retrans"   /* Number of retransmissions */
#define MNTOPT_PORT     "port"      /* Server's IP NFS port */
#define MNTOPT_SOFT     "soft"      /* Soft mount */
#define MNTOPT_HARD     "hard"      /* Hard mount */
#define MNTOPT_INTR     "intr"      /* Interruptable hard mounts */
#define MNTOPT_NOINTR   "nointr"    /* Uninterruptable hard mounts*/
#define MNTOPT_DEVS     "devs"      /* Device file access allowed */
#define MNTOPT_NODEVS   "nodevs"    /* No device file access allowed */
```

RETURN VALUE

`setmntent()` Returns a null pointer on error.

`getmntent()` Returns a null pointer on error or EOF. Otherwise, `getmntent()` returns a pointer to a mntent structure. Some of the fields comprising a mntent structure are optional in `/etc/checklist` and `/etc/mnttab`. In the supplied structure, such missing character pointer fields are set to NULL and missing integer fields are set to –1.

`addmntent()` Returns `1` on error.

`endmntent()` Returns `1`.

WARNINGS

The returned `mntent` structure points to static information that is overwritten in each call.

AUTHOR

`addmntent()`, `endmntent()`, `getmntent()`, `hasmntopt()`, and `setmntent()` were developed by The University of California, Berkeley, Sun Microsystems, Inc., and HP.

FILES

`/etc/checklist`
`/etc/mnttab`

SEE ALSO
checklist(4), getfsent(3X), mnttab(4).

NAME

getnetent(), getnetbyaddr(), getnetbyname(), setnetent(), endnetent() - get network entry

SYNOPSIS

```
#include <sys/socket.h>
#include <netdb.h>

struct netent *getnetent(void);

struct netent *getnetbyname(const char *name);

struct netent *getnetbyaddr(int net, int type);

int setnetent(int stayopen);

int endnetent(void);
```

DESCRIPTION

`getnetent()`, `getnetbyname()`, and `getnetbyaddr()` each return a pointer to a structure of type *netent* containing the broken-out fields of a line in the network data base, `/etc/networks`.

The members of this structure are:

`n_name`	The official name of the network.
`n_aliases`	A null-terminated list of alternate names for the network.
`n_addrtype`	The type of the network number returned; always AF_INET.
`n_net`	The network number.

Functions behave as follows:

`getnetent()`	Reads the next line of the file, opening the file if necessary.
`setnetent()`	opens and rewinds the file. If the *stayopen* flag is non-zero, the network data base is not closed after each call to `getnetent()` (either directly or indirectly through one of the other `getnet*` calls).
`endnetent()`	Closes the file.
`getnetbyname()`	Sequentially searches from the beginning of the file until a network name (among either the official names or the aliases) matching its parameter *name* is found, or until EOF is encountered.
`getnetbyaddr()`	Sequentially searches from the beginning of the file until a network number matching its parameter *net* is found, or until EOF is encountered. The parameter *net* must be in network order. The parameter *type* must be the constant `AF_INET`. Network numbers are supplied in host order (see *byteorder*(3N)).

If the system is running Network Information Service (NFS), `getnetbyname()` and `getnetbyaddr()` obtain their network information from the NIS server (see *ypserv*(1M) and *ypfiles*(4)).

RETURN VALUE

`getnetent()`, `getnetbyname()`, and `getnetbyaddr()` return a null pointer (0) on EOF or when they are unable to open `/etc/networks`. `getnetbyaddr()` also returns a null pointer if its *type* parameter is invalid.

WARNINGS

All information is contained in a static area so it must be copied if it is to be saved.

AUTHOR

`getnetent()` was developed by the University of California, Berkeley.

FILES

`/etc/networks`

SEE ALSO

ypserv(1M), networks(4), ypfiles(4).

NAME

getnetgrent(), setnetgrent(), endnetgrent(), innetgr() - get network group entry

SYNOPSIS

```
int innetgr(
      char *netgroup,
      char *machine,
      char *user,
      char *domain
);

int setnetgrent(char *netgroup);

int endnetgrent();

int getnetgrent(
      char **machinep,
      char **userp,
      char **domainp
);
```

DESCRIPTION

`innetgr()` Returns 1 if `netgroup` contains the machine, user, and domain triple as a member. Otherwise, it returns 0. If *machine*, *user*, or *domain* is NULL, `innetgr` interprets NULL to mean, any machine, user, or domain respectively. Refer to *netgroup*(4) for a description of `netgroup` membership criteria.

`getnetgrent()` Returns the next member of a network group. After the call, *machinep* contains a pointer to a string containing the name of the machine part of the network group member. Pointers *userp* and *domainp* behave in a manner similar to *machinep*. If any of these pointers are returned with a NULL value, they are interpreted as wild cards. `getnetgrent()` allocates space for the names. This space is released when an `endnetgrent()` call is made. `getnetgrent()` returns 1 if it succeeded in obtaining another network group member, 0 if it reached the end of the group.

`setnetgrent()` Establishes the network group from which `getnetgrent()` obtains members, and also restarts calls to `getnetgrent()` from the beginning of the list. If the previous `setnetgrent()` call was to a different network group, a `endnetgrent()` call is implied.

`endnetgrent()` Frees the space allocated during `getnetgrent()` calls.

AUTHOR

`getnetgrent()` was developed by Sun Microsystems, Inc.

FILES

`/etc/netgroup`

SEE ALSO

netgroup(4).

NAME

getopt(), optarg, optind, opterr - get option letter from argument vector

SYNOPSIS

```
#include <unistd.h>

int getopt(int argc, char * const argv[], const char *optstring);

extern char *optarg;

extern int optind, opterr, optopt;
```

DESCRIPTION

`getopt()` returns the next option letter in *argv* (starting from *argv*[1]) that matches a letter in *optstring*. *argc* and *argv* are the argument count and argument array as passed to *main*(). *optstring* is a string of recognized option characters; if a character is followed by a colon, the option takes an argument which may or may not be separated from it by white space.

`optind` is the index of the next element of the *argv*[ ] vector to be processed. It is initialized to 1 by the system, and `getopt()` updates it when it finishes with each element of *argv*[].

`getopt()` returns the next option character from *argv* that matches a character in *optstring*, if there is one that matches. If the option takes an argument, `getopt()` sets the variable `optarg` to point to the option-argument as follows:

- If the option was the last character in the string pointed to by an element of *argv*, then `optarg` contains the next element of *argv*, and `optind` is incremented by 2. If the resulting value of `optind` is greater than or equal to *argc*, this indicates a missing option argument, and `getopt()` returns an error indication.
- Otherwise, `optarg` points to the string following the option character in that element of *argv*, and `optind` is incremented by 1.

If, when `getopt()` is called, *argv*[*optind*] is NULL, or the string pointed to by *argv*[*optind*] either does not begin with the character `-` or consists only of the character `-`, `getopt()` returns –1 without changing *optind*. If *argv*[*optind*] points to the string `--`, `getopt()` returns –1 after incrementing *optind*.

If `getopt()` encounters an option character that is not contained in *optstring*, it returns the question-mark (`?`) character. If it detects a missing option argument, it returns the colon character (`:`) if the first character of *optstring* was a colon, or a question-mark character otherwise. In either case, `getopt()` sets the variable *optopt* to the option character that caused the error. If the application has not set the variable `opterr` to zero and the first character of *optstring* is not a colon, `getopt()` also prints a diagnostic message to standard error.

The special option `--` can be used to delimit the end of the options; –1 is returned, and `--` is skipped.

RETURN VALUE

`getopt()` returns the next option character specified on the command line. A colon (`:`) is returned if `getopt()` detects a missing argument and the first character of *optstring* was a colon (`:`).

A question-mark (`?`) is returned if `getopt()` encounters an option character not in *optstring* or detects a missing argument and the first character of *optstring* was not a colon (`:`).

Otherwise, `getopt()` returns –1 when all command line options have been parsed.

EXTERNAL INFLUENCES

Locale

The `LC_CTYPE` category determines the interpretation of option letters as single and/or multi-byte characters.

International Code Set Support

Single- and multi-byte character code sets are supported with the exception of multi-byte character file names.

EXAMPLES

The following code fragment shows to process arguments for a command that can take the mutually exclusive options `a` and `b`, and the options `f` and `o`, both of which require arguments:

```
#include <stdio.h>
#include <unistd.h>
main (int argc, char *argv[])
{
    int c;
    int bflg, aflg, errflg;
    extern char *optarg;
    extern int optind, optopt;
    .
    .
    .
    while ((c = getopt(argc, argv, ":abf:o:")) != -1)
        switch (c) {
        case 'a':
            if (bflg)
                errflg++;
            else
                aflg++;
            break;
        case 'b':
            if (aflg)
                errflg++;
            else {
                bflg++;
                bproc( );
            }
            break;
        case 'f':
            ifile = optarg;
            break;
        case 'o':
            ofile = optarg;
            break;
        case ':':       /* -f or -o without arguments */
            fprintf(stderr, "Option -%c requires an argument\n",
                optopt);
            errflg++;
            break;
        case '?':
            fprintf(stderr, "Unrecognized option: - %c\n",
                optopt);
            errflg++;
        }
    if (errflg) {
        fprintf(stderr, "usage: . . . ");
        exit (2);
    }
    for ( ; optind < argc; optind++) {
        if (access(argv[optind], 4)) {
    .
    .
    .
}
```

WARNINGS

Options can be any ASCII characters except colon (`:`), question mark (`?`), or null (`\0`). It is impossible to distinguish between a `?` used as a legal option, and the character that `getopt()` returns when it encounters an invalid option character in the input.

SEE ALSO
getopt(1).

STANDARDS CONFORMANCE
`getopt()`: AES, SVID2, XPG2, XPG3, XPG4, POSIX.2

`optarg`: AES, SVID2, XPG2, XPG3, XPG4, POSIX.2

`opterr`: AES, SVID2, XPG2, XPG3, XPG4, POSIX.2

`optind`: AES, SVID2, XPG2, XPG3, XPG4, POSIX.2

NAME

getpass() - read a password

SYNOPSIS

```
#include <unistd.h>

char *getpass(const char *prompt);
```

DESCRIPTION

`getpass()` reads up to a newline or EOF from the file `/dev/tty`, after prompting on the standard error output with the null-terminated string *prompt* and disabling echoing. A pointer is returned to a null-terminated string of at most 8 characters. If `/dev/tty` cannot be opened, a NULL pointer is returned. An interrupt terminates input and sends an interrupt signal to the calling program before returning.

FILES

`/dev/tty`

SEE ALSO

crypt(3C).

WARNING

The above routine uses `<stdio.h>`, which causes it to increase, more than might be expected, the size of programs not otherwise using standard I/O.

WARNINGS

The return value points to static data whose content is overwritten by each call.

STANDARDS CONFORMANCE

`getpass()`: AES, SVID2, XPG2, XPG3, XPG4

NAME

getprotoent(), getprotobynumber(), getprotobyname(), setprotoent(), endprotoent() - get protocol entry

SYNOPSIS

```
#include <netdb.h>

struct protoent *getprotoent(void);

struct protoent *getprotobyname(const char *name);

struct protoent *getprotobynumber(int proto);

int setprotoent(int stayopen);

int endprotoent(void);
```

DESCRIPTION

`getprotoent()`, `getprotobyname()`, and `getprotobynumber()` each return a pointer to a structure of type *protoent* containing the broken-out fields of a line in the network protocol data base, `/etc/protocols`.

The members of this structure are:

`p_name`	The official name of the protocol.
`p_aliases`	A null-terminated list of alternate names for the protocol.
`p_proto`	The protocol number.

Functions behave as follows:

`getprotoent()`	Reads the next line of the file, opening the file if necessary.
`setprotoent()`	Opens and rewinds the file. If the *stayopen* flag is non-zero, the protocol data base is not closed after each call to `getprotoent()` (either directly or indirectly through one of the other `getproto*` calls).
`endprotoent()`	Closes the file.
`getprotobyname()` `getprotobynumber()`	Sequentially search from the beginning of the file until a matching protocol name (among either the official names or the aliases) or protocol number is found, or until EOF is encountered.

If the system is running Network Information Service (NFS) services, `getprotobyname()` and `getprotobynumber()` get the host information from the NIS server (see *ypserv*(1M) and *ypfiles*(4)).

RETURN VALUE

`getprotoent()`, `getprotobyname()`, and `getprotobynumber()` return a null pointer (0) on EOF or when they are unable to open `/etc/protocols`.

WARNINGS

All information is contained in a static area so it must be copied if it is to be saved.

AUTHOR

`getprotoent()` was developed by the University of California, Berkeley.

FILES

`/etc/protocols`

SEE ALSO

ypserv(1M), protocols(4), ypfiles(4).

NAME

getpw() - get name from UID

SYNOPSIS

```
#include <pwd.h>

int getpw(uid_t uid, char *buf);
```

DESCRIPTION

`getpw()` searches the password file for a user ID number that equals *uid*, copies the line of the password file in which *uid* was found into the array pointed to by *buf*, and returns 0. `getpw()` returns non-zero if *uid* cannot be found. The line is null-terminated.

This routine is included only for compatibility with prior systems, and should not be used; see *getpwent*(3C) for routines to use instead.

NETWORKING FEATURES

NFS

This routine is implemented using `getpwuid()` (see *getpwuid*(3C)) and therefore uses the Network Information Service network database as described in *passwd*(4).

RETURN VALUE

`getpw()` returns non-zero on error.

WARNINGS

The above routine uses `<stdio.h>`, which causes it to increase, more than might be expected, the size of programs not otherwise using standard I/O.

AUTHOR

`getpw()` was developed by AT&T and HP.

FILES

/etc/passwd

SEE ALSO

getpwent(3C), passwd(4).

STANDARDS CONFORMANCE

`getpw()`: XPG2

NAME

getpwent(), getpwuid(), getpwnam(), setpwent(), endpwent(), fgetpwent() - get password file entry

SYNOPSIS

```
#include <pwd.h>

struct passwd *getpwent(void);

struct passwd *getpwuid(uid_t uid);

struct passwd *getpwnam(const char *name);

void setpwent(void);

void endpwent(void);

struct passwd *fgetpwent(FILE *stream);
```

DESCRIPTION

`getpwent()`, `getpwuid()`, and `getpwnam()` locate an entry in the `/etc/passwd` file, and return a pointer to an object of type `struct passwd`.

The `passwd` structure is defined in <`pwd.h`> and includes the following members:

```
char    *pw_name;
char    *pw_passwd;
int     pw_uid;
int     pw_gid;
char    *pw_age;
char    *pw_comment;
char    *pw_gecos;
char    *pw_dir;
char    *pw_shell;
long    pw_audid;
int     pw_audflg;
```

The `pw_comment` field is unused; the others have meanings described in *passwd*(4).

`getpwent()` When first called, `getpwent()` returns a pointer to the first `passwd` structure in the file. Thereafter, it returns a pointer to the next `passwd` structure in the file. In this way, successive calls can be used to search the entire file. `getpwent()` opens the `/etc/passwd` file prior to doing its work and leaves the file open afterward;

`setpwent()` Has the effect of rewinding this file to allow repeated searches; `endpwent()` Can be called to close the file when processing is complete.

`getpwuid()` Searches from the beginning of the file until a numeric user ID matching *uid* is found, and returns a pointer to the particular structure in which it was found.

`getpwnam()` searches from the beginning of the file until a login name matching *name* is found, and returns a pointer to the particular structure in which it was found.

`fgetpwent()` returns a pointer to the next `passwd` structure in the standard I/O stream *stream*, which should be open for reading, and its contents should match the format of `/etc/passwd`.

SECURITY FEATURES

If the secure password file (`/.secure/etc/passwd`) exists on the system and the calling process has permission to access it, the `getpwent()` routines fill in the encrypted password, audit ID, and audit flag from the corresponding entry in that file.

If the secure password file exists but the caller does not have permission to read it, the encrypted password field is set to * and the audit ID and audit flag are set to –1.

If the secure password file does not exist, the encrypted password in `/etc/passwd` is returned and the audit ID and audit flag are set to –1.

In situations where it is not necessary to get information from the regular password file, `getspwent()` is significantly faster because it avoids unnecessary searches of the regular password file (see *getspwent*(3C)),

and does not use the Network Information Service database.

`putpwent()` affects only `/etc/passwd`; the audit ID and audit flag in the password structure are ignored. `putspwent()` must be used to modify `/.secure/etc/passwd` (see *putspwent*(3C)).

NETWORKING FEATURES

NFS

If an entry beginning with a plus sign (+) or a minus sign (–) is found, these routines try to use the Network Information Service network database for data. See *passwd*(4) for proper syntax and operation.

RETURN VALUE

`getpwent()`, `getpwuid()`, `getpwnam()`, and `fgetpwent()` return a NULL pointer if an end-of-file or error is encountered on reading. Otherwise, the return value points to an internal static area containing a valid `passwd` structure.

WARNINGS

The above routines use `<stdio.h>` and the Network Information Service library, which causes them to increase the size of programs, not otherwise using standard I/O and Network Information Service, more than might be expected.

The value returned by these functions points to a single static area that is overwritten by each call to any of the functions, so it must be copied if it is to be saved.

The following fields have numerical limitations as noted:

- The user ID is an integer value between -2 and UID_MAX inclusive.
- The group ID is an integer value between 0 and UID_MAX inclusive.

If either of these values are out of range, the *getpwent*(3C) functions will reset the associated ID value to (UID_MAX+1).

DEPENDENCIES

NFS

Files:

`/etc/yp/domainname/passwd.byname`
`/etc/yp/domainname/passwd.byuid`

See Also:

ypcat(1).

AUTHOR

`getpwent()`, `getpwuid()`, `getpwnam()`, `setpwent()`, `endpwent()`, and `fgetpwent()` were developed by AT&T and HP.

FILES

`/etc/passwd`

SEE ALSO

ypcat(1), cuserid(3S), getgrent(3C), getlogin(3C), getspwent(3C), stdio(3S), putspwent(3C), passwd(4), spasswd(4), limits(5).

STANDARDS CONFORMANCE

`getpwent()`: SVID2, XPG2
`endpwent()`: SVID2, XPG2

`fgetpwent()`: SVID2, XPG2
`getpwnam()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

`getpwuid()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1
`setpwent()`: SVID2, XPG2

NAME

getrpcent(), getrpcbyname(), getrpcbynumber() - get rpc entry

SYNOPSIS

```
#include <netdb.h>

struct rpcent *getrpcent();

struct rpcent *getrpcbyname(char *name);

struct rpcent *getrpcbynumber(int number);

int setrpcent(int stayopen);

int endrpcent();
```

DESCRIPTION

`getrpcent()`, `getrpcbyname()`, and `getrpcbynumber()` each return a pointer to an object with the following structure containing the broken-out fields of a line in the rpc program number data base, `/etc/rpc`.

```
struct rpcent {
    char *r_name;       /* name of server for this rpc program */
    char **r_aliases;   /* NULL terminated list of aliases */
    int  r_number;      /* rpc program number for this service */
};
```

Functions

`getrpcent()`	Read the next line of the file, opening the file if necessary.
`setrpcent()`	Open and rewind the file. If the *stayopen* flag is non-zero, the rpc database is not closed after each call to `getrpcent()` (either directly or indirectly through one of the other `getrpc*()` calls).
`endrpcent()`	Close the file.
`getrpcbyname()`	Sequentially search from the beginning of the file until a matching rpc program name is found, or until EOF is encountered.
`getrpcbynumber()`	Sequentially search from the beginning of the file until a matching rpc program number is found, or until EOF is encountered.

RETURN VALUE

`getrpcent()`, `getrpcbyname()`, and `getrpcbynumber()` return a null pointer (0) on EOF or when unable to access the information in `/etc/rpc` either directly or through a Network Information Service database.

WARNINGS

All information is contained in a static area so it must be copied if it is to be saved.

AUTHOR

`getrpcent()` was developed by Sun Microsystems, Inc.

FILES

`/etc/rpc`

SEE ALSO

rpcinfo(1M), rpc(4).

NAME

getrpcport() - get RPC port number

SYNOPSIS

```
int getrpcport(
      char *host,
      int prognum,
      int versnum,
      int proto
);"
```

DESCRIPTION

`getrpcport()` returns the port number for version *versnum* of the RPC program *prognum* running on *host* and using protocol *proto*. It returns 0 if it cannot contact *portmap* or if *prognum* is not registered. If *prognum* is registered but not with version *versnum*, it returns the port number of the last registered (prognum, *proto*) pair.

WARNING

User applications that call this routine must be linked with `/usr/include/librpcsvc.a`. For example,

```
cc my_source.c -lrpcsvc
```

AUTHOR

`getrpcport()` was developed by Sun Microsystems, Inc.

SEE ALSO

portmap(1M).

NAME

gets(), fgets() - get a string from a stream

SYNOPSIS

```
#include <stdio.h>

char *gets(char *s);

char *fgets(char *s, int n, FILE *stream);
```

DESCRIPTION

`gets()` Reads characters from the standard input stream, `stdin`, into the array pointed to by *s*, until a new-line character is read or an end-of-file condition is encountered. The new-line character is discarded and the string is terminated with a null character.

`fgets()` Reads characters from the *stream* into the array pointed to by *s*, until *n–1* characters are read, a new-line character is read and transferred to *s*, or an end-of-file condition is encountered. The string is then terminated with a null character.

RETURN VALUE

Upon sucessful completion, `fgets()` and `gets()` return *s*. If the stream is at end-of-file, the end-of-file indicator for the stream is set and a null pointer is returned. If a read error occurs, the error indicator for the stream is set, `errno` is set to indicate the error, and a null pointer is returned.

`ferror()` and `feof()` can be used to distinguish between an error condition and an end-of-file condition.

ERRORS

`fgets()` and `gets()` fail if data needs to be read into the *stream*'s buffer, and:

[EAGAIN] The `O_NONBLOCK` flag is set for the file descriptor underlying *stream* and the process would be delayed in the read operation.

[EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for reading.

[EINTR] The read operation was terminated due to the receipt of a signal, and either no data was transferred or the implementation does not report partial transfer for this file.

[EIO] The process is a member of a background process and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the `SIGTTIN` signal or the process group of the process is orphaned.

Additional `errno` values can be set by the underlying `read()` function (see *read*(2)).

SEE ALSO

ferror(3S), fopen(3S), fread(3S), getc(3S), puts(3S), scanf(3S).

STANDARDS CONFORMANCE

`gets()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`fgets()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

getservent(), getservbyport(), getservbyname(), setservent(), endservent() - get service entry

SYNOPSIS

```
#include <netdb.h>

struct servent *getservent(void);

struct servent *getservbyname(
    const char *name,
    const char *proto);

struct servent *getservbyport(int port, const char *proto);

int setservent(int stayopen);

int endservent(void);
```

DESCRIPTION

`getservent()`, `getservbyname()`, and `getservbyport()` each return a pointer to a structure of type *servent* containing the broken-out fields of a line in the network services data base, `/etc/services`.

The members of this structure are:

`s_name`	The official name of the service.
`s_aliases`	A null-terminated list of alternate names for the service.
`s_port`	The port number at which the service resides.
`s_proto`	The name of the protocol to use when contacting the service.

Functions behave as follows:

`getservent()`	Reads the next line of the file, opening the file if necessary.
`setservent()`	Opens and rewinds the file. If the *stayopen* flag is non-zero, the services data base is not closed after each call to `getservent()` (either directly or indirectly through one of the other `getserv*` calls).
`endservent()`	Closes the file.
`getservbyname()` `getservbyport()`	Sequentially search from the beginning of the file until a matching service name (among either the official names or the aliases) or port number is found, or until EOF is encountered. If a non-NULL protocol name is also supplied (such as `tcp` or `udp`), searches must also match the protocol. If the system is running Network Information Service (NFS), `getservbyname()` gets the service information from the NIS server (see *ypserv*(1M) and *ypfiles*(4)).

RETURN VALUE

`getservent()`, `getservbyname()`, and `getservbyport()` return a null pointer (0) on EOF or when they are unable to open `/etc/services`.

WARNINGS

All information is contained in a static area so it must be copied if it is to be saved.

AUTHOR

`getservent()` was developed by the University of California, Berkeley.

FILES

`/etc/services`

SEE ALSO

ypserv(1M), services(4), ypfiles(4).

NAME

getspwent(), getspwuid(), getspwaid(), getspwnam(), setspwent(), endspwent(), fgetspwent() - get secure password file entry

SYNOPSIS

```
#include <pwd.h>

struct s_passwd *getspwent(void);

struct s_passwd *getspwuid(uid_t uid);

struct s_passwd *getspwaid(aid_t aid);

struct s_passwd *getspwnam(const char *name);

void setspwent(void);

void endspwent(void);

struct s_passwd *fgetspwent(FILE *stream);
```

DESCRIPTION

These privileged routines provide access to the secure password file in a manner similar to the way *getpwent*(3C) routines handle the regular password file, (`/etc/passwd`).

These routines are particularly useful in situations where it is not necessary to get information from the regular password file. *getspwent*(3C) routines run significantly faster than *getpwent*(3C) routines because they avoid unnecessary scanning of the password file and use of Network Information Service.

`getspwent()`, `getspwuid()`, `getspwaid()`, and `getspwnam()` each return a pointer to an object. The `s_passwd` structure is written in the `/.secure/etc/passwd` file, and consists of five fields per line, as follows:

```
struct s_passwd {
    char  *pw_name;     /* login name */
    char  *pw_passwd;   /* encrypted password */
    char  *pw_age;      /* password age */
    int   pw_audid;     /* audit ID   */
    int   pw_audflg;    /* audit flag 1=on, 0=off */
};
```

Since the `s_passwd` structure is declared in the <pwd.h> header file, it is unnecessary to redeclare it.

`getspwent()` When first called, `getspwent()` returns a pointer to each `s_passwd` structure in `/.secure/etc/passwd` in sequence. Subsequent calls can be used to search the entire file.

`getspwuid()` Searches each entry from the beginning of the file until it finds a numerical user ID matching *uid*. It then returns a pointer to the particular structure in which *uid* is found. `getspwaid()` Similarly searches for a numerical audit ID matching *aid* and returns a pointer to the particular structure in which *aid* is found (see *spasswd*(4) for details on this field).

`getspwnam()` Searches from the beginning of the file until a login name matching *name* is found. Returns a pointer to the particular structure in which *name* is found.

`setspwent()` Resets the file pointer to the beginning of the `/.secure/etc/passwd` file to allow repeated searches.

`endspwent` Can be called to close the secure password file when processing is complete.

`fgetspwent` Returns a pointer to the next `s_passwd` structure in the stream *stream*, which matches the format of `/.secure/etc/passwd`.

RETURN VALUE

`getspwent()` returns a NULL pointer if any of these routines encounter an end-of-file or error while searching, or if the effective user ID of the calling process is not zero.

WARNINGS

The above routines use <stdio.h>, which causes them to increase the size of programs by more than

might otherwise be expected.

Since all information is contained in a static area, it must be copied to be saved.

AUTHOR

`getspwent()` was developed by HP.

FILES

`/.secure/etc/passwd`

SEE ALSO

ypcat(1), getgrent(3C), getlogin(3C), getpwent(3C), putspwent(3C), passwd(4), spasswd(4).

NAME

getsubopt() - parse suboptions from a string.

SYNOPSIS

```
#include <unistd.h>

int getsubopt(char **optionp, char *tokens[], char **valuep);
```

DESCRIPTION

`getsubopt()` parses suboptions in a flag argument that were initially parsed by `getopt()` (see getopt(3C)). These suboptions are separated by commas, and may consist of either a single token, or a token-value pair separated by an equal sign. Because commas delimit suboptions in the option string, they are not allowed to be part of the suboption or the value of a suboption. Similarly, because the equal sign separates a token from its value, a token must not contain an equals sign. An example command that uses this syntax is `mount`. `mount` allows parameters to be specified with the `-` switch as follows:

```
mount -o rw,hard,bg,wsize=1024 speed:/usr /usr
```

In this example there are four suboptions: `rw`, `hard`, `bg`, and `wsize`, the last of which has an associated value of 1024.

`getsubopt()` takes the address of a pointer to the option string, a vector of possible tokens, and the address of a value string pointer. It returns the index of the token that matched the suboption in the input string or –1 if there was no match. If the option string at **optionp* contains only one suboption, `getsubopt()` updates **optionp* to point to the null at the end of the string, otherwise it isolates the suboption by replacing the comma separator with a null, and updates **optionp* to point to the start of the next suboption. If the suboption has an associated value, `getsubopt()` updates **valuep* to point to the value's first character. Otherwise it sets **valuep* to NULL.

The token vector is organized as a series of pointers to NULL-terminated strings. The end of the token vector is identified by NULL.

When `getsubopt()` returns, if **valuep* is not NULL then the suboption processed included a value. The calling program can use this information to determine if the presence or lack of a value for this suboption is an error.

Additionally, when `getsubopt()` fails to match the suboption with the tokens in the *tokens* array, the calling program should decide if this is an error, or if the unrecognized option should be passed on to another program.

EXAMPLES

The following code fragment shows how options can be processed to the `mount` command by using `getsubopt()`.

```
char *myopts[] = {
#define READONLY        0
                        "ro",
#define READWRITE       1
                        "rw",
#define WRITESIZE       2
                        "wsize",
#define READSIZE        3
                        "rsize",
                        NULL};

main (int argc, char **argv)
{
    int sc, c, errflag;
    char *options, *value;
    extern char *optarg;
    extern int optind;
    .
    .
    .
```

```
        while ((c = getopt(argc, argv, "abf:o:")) != EOF)
            switch (c) {
            case 'a':  /* process 'a' option */
                break;
            case 'b':  /* process 'b' option */
                break;
            case 'f':
                ofile = optarg;
                break;
            case '?':
                errflag++;
                break;
            case 'o':
                options = optarg;
                while (*options != '\0') {
                    switch(getsubopt(&options, myopts, &value)) {
                    case READONLY: /* process ro option */
                        break;
                    case READWRITE: /* process rw option */
                        break;
                    case WRITESIZE: /* process wsize option */
                        if (value == NULL) {
                            error_no_arg();
                            errflag++;
                        }
                        else
                            write_size = atoi(value);
                        break;
                    case READSIZE: /* process rsize option */
                        if (value == NULL) {
                            error_no_arg();
                            errflag++;
                        }
                        else
                            read_size = atoi(value);
                        break;
                    default:
                        /* process unknown token */
                        error_bad_token(value);
                        errflag++;
                        break;
                    }
                }
                break;
            }
        }
        if (errflg) {
            fprintf(stderr, "usage: . . . ");
            exit (2);
        }
        for ( ; optind < argc; optind++) {
            /* process remaining arguments */
             .
             .
             .
    }
```

EXTERNAL INFLUENCES

Locale

The `LC_CTYPE` category determines the interpretation of option letters as single and/or multi-byte

characters.

International Code Set Support
Single- and multi-byte character code sets are supported with the exception of multi-byte-character file names.

SEE ALSO
getopt(3C).

STANDARDS CONFORMANCE
`getsubopt()`: SVID3

NAME

gettimer - get value of a per-process timer

SYNOPSIS

```
#include <sys/timers.h>

int gettimer(timer_t timerid, struct itimerspec *value);
```

DESCRIPTION

`gettimer()` returns an *itimerspec* structure value to the *value* argument. The *it_value* member of the structure represents the amount of time in the current interval before the timer expires for the timer specified in *timerid*, or zero if the timer is disabled. The *it_interval* member has the value last set by `reltimer()` (see *reltimer*(3C)). The members of *value* are subject to the resolution of the timer (see *mktimer*(3C)).

The behavior of this function is undefined if *value* is NULL.

RETURN VALUE

Upon successful completion, `gettimer()` returns zero; otherwise, it returns –1 and set `errno` to indicate the error.

ERRORS

gettimer() fails if any of the following conditions are encountered:

[EINVAL] *timerid* does not correspond to an ID returned by `mktimer()`.

[EIO] An error occurred while accessing the clock device.

SEE ALSO

reltimer(3C), mktimer(3C), <sys/timers.h>.

STANDARDS CONFORMANCE

`gettimer()`: AES

NAME

getusershell(), setusershell(), endusershell() - get legal user shells

SYNOPSIS

```
#include <unistd.h>

char *getusershell(void);

void setusershell(void);

void endusershell(void);
```

DESCRIPTION

`getusershell()` Returns a pointer to the first legal user shell as defined in the file `/etc/shells` (see *shells*(4)). If `/etc/shells` does not exist or is not readable, `getusershell()` returns the following standard system shells:

```
/bin/sh
/bin/rsh
/bin/ksh
/bin/rksh
/bin/csh
/bin/pam
/usr/bin/keysh
/bin/posix/sh
```

as if they were contained in `/etc/shells`. The file is left open so that the next call returns the next shell. A null pointer (0) is returned on EOF or error.

`setusershell()`
Rewinds the file.

`endusershell()`
Closes the file.

WARNINGS

All information is contained in a static area and therefore must be copied if it is to be saved.

AUTHOR

`getusershell()` was developed by HP and the University of California, Berkeley.

FILES

`/etc/shells`

SEE ALSO

shells(4).

NAME

getutent(), getutid(), getutline(), pututline(), _pututline(), setutent(), endutent(), utmpname() - access utmp file entry

SYNOPSIS

```
#include <utmp.h>

struct utmp *getutent(void);

struct utmp *getutid(struct utmp *id);

struct utmp *getutline(struct utmp *line);

struct utmp *_pututline(const struct utmp *utmp);

void pututline(const struct utmp *utmp);

void setutent(void);

void endutent(void);

void utmpname(const char *file);
```

DESCRIPTION

`getutent()`, `getutid()`, and `getutline()` each return a pointer to a structure of the following type:

```
struct utmp {
    char ut_user[8];           /* User login name */
    char ut_id[4];             /* /etc/inittab id (usually line #) */
    char ut_line[12];          /* device name (console, lnxx) */
    pid_t ut_pid;              /* process id */
    short ut_type;             /* type of entry */
    struct exit_status {
        short e_termination;   /* Process termination status */
        short e_exit;          /* Process exit status */
        } ut_exit;             /* The exit status of a process */
                               /* marked as DEAD_PROCESS. */
    unsigned short ut_reserved1;  /* Reserved for future use */
    time_t ut_time;            /* time entry was made */
    char ut_host[16];          /* host name, if remote; NOT SUPPORTED */
    unsigned long ut_addr;     /* Internet addr of host, if remote */
};
```

`getutent()` Reads in the next entry from a `utmp`-like file. If the file is not already open, `getutent()` opens it. If it reaches the end of the file, `getutent()` fails.

`getutid()` Searches forward from the current point in the *utmp* file until it finds an entry with a `ut_type` matching *id->ut_type* if the type specified is `RUN_LVL`, `BOOT_TIME`, `OLD_TIME`, or `NEW_TIME`. If the type specified in *id* is INIT_PROCESS, LOGIN_PROCESS, USER_PROCESS, or DEAD_PROCESS, `getutid()` returns a pointer to the first entry whose type is one of these four, and whose `ut_id` field matches *id->ut_id*. If end-of-file is reached without a match, `getutid()` fails.

`getutline()` Searches forward from the current point in the *utmp* file until it finds an entry of type `LOGIN_PROCESS` or `USER_PROCESS` that also has a `ut_line` string matching the *line->ut_line* string. If end-of-file is reached without a match, `getutline()` fails.

`pututline()` Writes out the supplied `utmp` structure into the *utmp* file. `pututline()` uses `getutid()` to search forward for the proper location if it is not already there. It is normally expected that the application program has already searched for the proper entry by using one of the `getut()` routines before calling `pututline()`. If the search as already been made, `pututline()` does not repeat it. If `pututline()` does not find a matching slot for the new entry, it adds a new entry to the end of the file.

`_pututline()` Performs the same actions as `pututline()`, except that it returns a value useful for error checking.

`setutent()` Resets the input stream to the beginning of the file. This should be done before each search for a new entry if it is desired that the entire file be examined.

`endutent()` Closes the currently open file.

`utmpname()` Allows the user to change the name of the file being examined from `/etc/utmp` to any other file. The other file is usually `/etc/wtmp`. If the file does not exist, its absence is not discovered until the first subsequent attempt to reference the file. `utmpname()` does not open the file — it merely closes the old file if it is currently open, and saves the new file name.

The most current entry is saved in a static structure. Multiple accesses require that the structure be copied before further accesses are made. During each call to either `getutid()` or `getutline()`, the static structure is examined before performing more I/O. If the contents of the static structure match what the routine is searching for, no additional searching is done. Therefore, if using `getutline()` to search for multiple occurrences, it is necessary to zero out the static structure after each success; otherwise `getutline()` simply returns the same pointer over and over again. There is one exception to the rule about removing the structure before a new read: The implicit read done by `pututline()` (if it finds that it is not already at the correct place in the file) does not alter the contents of the static structure returned by `getutent()`, `getutid()`, or `getutline()` if the user has just modified those contents and passed the pointer back to `pututline()`.

RETURN VALUE

These functions return a NULL pointer upon failure to read (whether for permissions or having reached end-of-file), or upon failure to write. They also return a NULL pointer if the size of the file is not an integral multiple of `sizeof(struct utmp)`.

`_pututline()` behaves the same as `pututline()`, except that it returns a pointer to a static location containing the most current utmp entry if the `_pututline()` call succeeds. The contents of this structure is identical to the contents of the supplied `utmp` structure if successful. If `_pututline()` fails, it returns a NULL pointer.

WARNINGS

Some vendors' versions of `getutent()` erase the *utmp* file if the file exists but is not an integral multiple of `sizeof(struct utmp)`. Given the possiblity of user error in providing a name to *utmpname* (such as giving improper arguments to *who*(1)), HP-UX does not do this, but instead returns an error indication.

FILES

`/etc/utmp`
`/etc/wtmp`

SEE ALSO

ttyslot(3C), utmp(4).

STANDARDS CONFORMANCE

`endutent()`: SVID2, XPG2

`getutent()`: SVID2, XPG2

`getutid()`: SVID2, XPG2

`getutline()`: SVID2, XPG2

`pututline()`: SVID2, XPG2

`setutent()`: SVID2, XPG2

`utmpname()`: SVID2, XPG2

NAME

getwc(), getwchar(), fgetwc() - get a wide character from a stream file

SYNOPSIS

```
#include <wchar.h>

wint_t getwc(FILE *stream);

wint_t getwchar(void);

wint_t fgetwc(FILE *stream);
```

Remarks:

These functions are compliant with the XPG4 Worldwide Portability Interface wide-character I/O functions. They parallel the 8 bit character I/O functions defined in *getc(3S)*.

DESCRIPTION

`getwc()` Returns the next character from the named input *stream*, converts that to the corresponding wide character and moves the file pointer ahead one character in *stream*. `getwchar()` is defined as `getwc(stdin)`. `getwc()` and `getwchar()` are defined both as macros and as functions.

`fgetwc()` Behaves like `getwc()`, but is a function rather than a macro.

Definitions for these functions, the types `wint_t`, `wchar_t` and the value `WEOF` are provided in header file <wchar.h>.

RETURN VALUE

Upon successful completion, `getwc()`, `getwchar()`, and `fgetwc()` return the next wide character read from *stream* (`stdin` for `getwchar()`) converted to a type `wint_t`. If the stream is at end-of-file, the end-of-file indicator for the stream is set and `WEOF` is returned. If a read error occurs, the error indicator for the stream is set, `errno` is set to indicate the error, and `WEOF` is returned.

`ferror()` and `feof()` can be used to distinguish between an error condition and an end-of-file condition.

ERRORS

`getwc()`, `getwchar()`, and `fgetwc()` fail if data needs to be read into the *stream*'s buffer, and:

[EAGAIN] The `O_NONBLOCK` flag is set for the file descriptor underlying *stream* and the process would be delayed in the read operation.

[EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for reading.

[EINTR] The read operation was terminated due to the receipt of a signal, and either no data was transferred or the implementation does not report partial transfer for this file.

[EIO] The process is a member of a background process and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the `SIGTTIN` signal or the process group of the process is orphaned.

[EILSEQ] The data obtained from the input stream does not form a valid wide character.

Additional `errno` values may be set by the underlying `read()` function (see *read*(2)).

EXTERNAL INFLUENCES

Locale

The `LC_CTYPE` category determines how wide character conversions are done.

International Code Set Support

Single- and multi-byte character code sets are supported.

WARNINGS

`getwc()` and `getwchar()` are implemented both as library functions and macros. The macro versions, which are used by default, are defined in <wchar.h>. To obtain the library function, either use a `#undef` to remove the macro definition or, if compiling in ANSI-C mode, enclose the function name in parenthesis or use the function address. The following example illustrates each of these methods :

```
#include <wchar.h>
#undef getwc
```

```
        ...
main()
{
    wint_t (*get_wchar()) ();
                ...
    return_val=getwc(c,fd);
                ...
    return_val=(getwc)(c,fd1);
                ...
    get_wchar = getwchar;
};
```

If the value returned by **getwc()**, **getwchar()**, or **fgetwc()** is stored into a type **wchar_t** variable then compared against the constant **WEOF**, the comparison may never succeed because extension of a **wchar_t** to a **wint_t** is machine-dependent.

The macro version of **getwc()** incorrectly treats a *stream* argument with side effects. In particular, **getwc(*f++)** does not work sensibly. The function version of **getwc()** or **fgetwc()** should be used instead.

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), fread(3S), fgetws(3C), putwc(3C), read(2), scanf(3S).

STANDARDS CONFORMANCE

getwc(): XPG4

fgetwc(): XPG4

getwchar(): XPG4

NAME

glob(), globfree() - file name generation function

SYNOPSIS

```
#include <glob.h>

int glob(
    const char *pattern,
    int flags,
    int (*errfunc)(const char *, int),
    glob_t *pglob
);

void globfree(glob_t *pglob);
```

DESCRIPTION

`glob()` is a pathname generator. *pattern* is a pointer to a pathname pattern to be expanded. If *pattern* contains any of the special characters `*`, `?`, or `[`, *pattern* is matched against all accessible pathnames. In order to have access to a pathname, `glob()` requires:

- Search permission on every component of a path except the last.
- Read permission on each directory of any filename component of *pattern* that contains any of the above special characters.

`glob`() stores the number of matched pathnames in *pglob* -> *gl_pathc* and a pointer to a sorted list of pathnames in *pglob* ->*gl_pathv*. The first pointer after the last pathname is a NULL pointer.

It is the caller's responsibility to allocate space for the structure pointed to by *pglob*. `glob()` allocates other space as needed, including the memory pointed to by *gl_pathv*. `globfree()` frees any space associated with *pglob* from a previous call to `glob()`.

The *flags* argument is used to control the behavior of `glob()`. The value of *flags* is the bit-wise inclusive OR of the following constants defined in <`glob.h`>:

`GLOB_NOESCAPE`	Disable backslash escaping.
`GLOB_ERR`	Causes `glob()` to return when it first encounters a directory that it cannot open or read. Ordinarily, `glob()` continues to find matches.
`GLOB_MARK`	Each pathname that matches *pattern* and is a directory has a `/` appended.
`GLOB_NOSORT`	Ordinarily, `glob()` sorts the matching pathnames according to the currently active *collation sequence* as defined by the `LC_COLLATE` category. When this flag is used, the order of pathnames returned is unspecified.
`GLOB_NOCHECK`	If *pattern* does not match any pathname, `glob()` returns a list consisting of only *pattern*, and the number of matched pathnames is 1.
`GLOB_DOOFFS`	Make use of *pglob* ->*gl_offs*. If this flag is set, *pglob* ->*gl_offs* is used to specify how many NULL pointers to add to the beginning of *pglob* ->*gl_pathv*. In other words, *pglob* ->*gl_pathv* points to *pglob* ->*gl_offs* NULL pointers, followed by *pglob* ->*gl_pathc* pathname pointers, followed by a NULL pointer.
`GLOB_APPEND`	Append pathnames generated to the ones from a previous call to `glob()`.

If `GLOB_APPEND` is specified in *flags*, the following rules apply:

- If the application set `GLOB_DOOFFS` in the first call to `glob()`, then it also sets it in all subsequent calls to `glob()`, as long as the same `glob_t` structure is used for appending.
- If the application did not set `GLOB_DOOFFS` in the first call to `glob()`, then it does not set it in any subsequent calls to `glob()`, as long as the same glob_t structure is used for appending.
- If `GLOB_DOOFFS` is set, the value of *pglob* ->*gl_offs* must not be modified between calls to `glob()`.
- After the second call, *pglob* ->*gl_pathv* points to a list containing the following:

- Zero or more NULLs, as specified by GLOB_DOOFS and *pglob ->gl_offs*.
- Pointers to the pathnames that were in the *pglob ->gl_pathv* list before the call, in the same order as before.
- Pointers to the new pathnames generated by the second call, in the specified order.

- The count returned in *pglob ->gl_pathc* is the sum of the number of pathnames matched in the previous and current calls to `glob()`.
- The application does not modify *pglob ->gl_pathc* or *pglob ->gl_pathv* between calls to `glob()`.

If, during the search, a directory is encountered that cannot be opened or read and *errfunc* is not NULL, `glob()` calls (**errfunc*)() with two arguments:

- A pointer to the path that failed.
- The value of `errno` from the failure.

If *errfunc* is called and returns non-zero, or if the GLOB_ERR flag is set in *flags*, `glob()` stops the scan and returns GLOB_ABORTED after setting `gl_pathc` and `gl_pathv` in *pglob* to reflect the paths already scanned. If GLOB_ERR is not set and either *errfunc* is NULL or (**errfunc*)() returns zero, the error is ignored.

Pattern Matching Notation

The form of the patterns is the Pattern Matching Notation as qualified for Filename Expansion (see *regexp*(5)) with the following exceptions:

- Tilde (~) expansion is not performed.
- Variable expansion is not performed.

If a filename component ends with a plus sign (+) (indicating a context-dependent file), the plus sign must be explicitly matched by a plus sign in the pattern; it cannot be matched by either the asterisk or question mark special characters, or by bracket expressions.

EXTERNAL INFLUENCES

Locale

The LC_COLLATE category determines the collating sequence used in compiling and executing regular expressions, and also the order of the returned paths if GLOB_NOSORT is not selected.

The LC_CTYPE category determines the interpretation of text as single and/or multi-byte characters, and which characters are matched by character class expressions in regular expressions.

International Code Set Support

Single- and multi-bye character code sets are supported.

RETURN VALUE

If `glob()` terminates due to an error, it returns one of the following constants (defined in <glob.h>); otherwise, it returns zero.

GLOB_NOSPACE	An attempt to allocate memory failed.
GLOB_ABORTED	The scan was stopped because GLOB_ERR was set or (**errfunc*)() returned non-zero.
GLOB_NOMATCH	The *pattern* does not match any existing pathname, and GLOB_NOCHECK was not set in *flags*.

In any case, the argument *pglob ->gl_pathc* returns the number of matched pathnames and the argument *pglob ->gl_pathv* contains a pointer to a null-terminated list of matched and sorted pathnames.

However, if *pglob ->gl_pathc* is zero, the content of *pglob ->gl_pathv* is undefined.

If the *pattern* argument passed to `glob()` is badly constructed, `glob()` returns zero and sets `gl_pathc` to zero unless GLOB_NOCHECK was set, in which case *pattern* is returned and `gl_pathc` is set to 1.

WARNINGS

GLOB_APPEND must not be set in an initial call to `glob()`.

AUTHOR

`glob()` and `globfree()` were developed by HP.

SEE ALSO

ed(1), grep(1), sh(1), fnmatch(3C), malloc(3C), malloc(3X), regexp(5).

STANDARDS CONFORMANCE

`glob()`: XPG4, POSIX.2

`globfree()`: XPG4, POSIX.2

NAME

gpio_get_status - return status lines of GPIO card

SYNOPSIS

```
#include <dvio.h>

int gpio_get_status(int eid);
```

DESCRIPTION

`gpio_get_status()` reads the status register of the GPIO interface associated with the device file identified by *eid*. *eid* is an entity identifier of an open GPIO device file obtained from an `open()`, `dup()`, `fcntl()`, or `creat()` call (see *open*(2), *dup*(2), *fcntl*(2), or *creat*(2)). The current state of each status line on the interface card is mapped to the value returned, with `STS0` mapped to the least significant bit. Only *x* least-significant bits are used, where *x* is the number of status lines available on the hardware interface being used.

DEPENDENCIES

Series 300/400

For the HP98622A, *x* is 2.

Series 800

For the HP27114A, *x* is 2.

For the HP27114B, *x* is 6.

For the HP28651A, *x* is 5.

RETURN VALUE

`gpio_get_status()` returns the value of the status register of the GPIO interface associated with *eid*, and -1 if an error was encountered.

ERRORS

`gpio_get_status()` fails if any of the following conditions are encountered and sets `errno` accordingly:

[EBADF] *eid* does not refer to an open file.

[ENOTTY] *eid* does not refer to a GPIO device file.

NAME

gpio_set_ctl - set control lines on GPIO card

SYNOPSIS

```
#include <dvio.h>

int gpio_set_ctl(int eid, int value);
```

DESCRIPTION

`gpio_set_ctl()` sets the control register of a GPIO interface. *eid* is an entity identifier of an open GPIO device file obtained from an `open()`, `dup()`, `fcntl()`, or `creat()` call (see *open*(2), *dup*(2), *fcntl*(2), and *creat*(2)). *value* is the value to be written into the control register of the GPIO interface associated with *eid*.

value is mapped onto the control lines on the interface card, with the least significant bit mapped to `CTL0`. Only the *x* least significant bits are used, where *x* is the number of control lines available on the hardware interface being used.

DEPENDENCIES

Series 300/400

For the HP 98622A, *x* is 2.

Series 800

For the HP 27114A, *x* is 3.

For the HP 27114B, *x* is 6.

For the HP 28651A, *x* is 5.

RETURN VALUE

`gpio_set_ctl()` returns 0 if successful and -1 if an error was encountered.

ERRORS

`gpio_set_ctl()` fails if any of the following conditions are encountered, and sets `errno` accordingly:

[EBADF] *eid* does not refer to an open file.

[ENOTTY] *eid* does not refer to a GPIO device file.

NAME

hpib_abort() - stop activity on specified HP-IB bus

SYNOPSIS

```
#include <dvio.h>

int hpib_abort(int eid);
```

DESCRIPTION

`hpib_abort()` terminates activity on the addressed HP-IB bus by pulsing the IFC line. *eid* is an entity identifier of an open HP-IB raw bus device file obtained from an `open()`, `dup()`, `fcntl()`, or `creat()` call.

`hpib_abort()` also sets the REN line and clears the ATN line. The status of the SRQ line is not affected. The interface must be the system controller of the bus.

RETURN VALUE

`hpib_abort()` returns 0 (zero) if successful, or –1 if an error was encountered.

ERRORS

`hpib_abort()` fails under the following circumstances, and sets `errno` (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EIO]	the specified interface is not the system controller (see DEPENDENCIES below).
[ETIMEDOUT]	a timeout occurred.
[EACCES]	The interface associated with this *eid* is locked by another process and `O_NDELAY` is set for this *eid* (see io_lock(3I)).

DEPENDENCIES

Series 300/400:

The HP 98625A/B and HP 25560A HP-IB interfaces do not clear the ATN line. EIO is returned if a timeout occurs.

Series 800:

If the interface is not currently the system controller, `hpib_abort()` sets `errno` to [EACCES] instead of to [EIO].

AUTHOR

`hpib_abort()` was developed by HP.

SEE ALSO

dup(2), creat(2), fcntl(2), open(2).

NAME

hpib_address_ctl() - set the HP-IB bus address for an interface

SYNOPSIS

```
#include <dvio.h>

int hpib_address_ctl(int eid, int ba);
```

DESCRIPTION

`hpib_address_ctl()` sets the HP-IB bus address of the interface associated with *eid* to *ba*. *eid* is an entity identifier of an open HP-IB raw bus device file obtained from an `open()`, `dup()`, `fcntl()`, or `creat()` call. *ba* is an integer and must be in the range of [0-30].

The new bus address remains in effect until a reboot, an `io_reset()` call, or another `hpib_address_ctl()` call occurs. When a `reboot()` or `io_reset()` call occurs, the HP-IB bus address reverts to its powerup value.

RETURN VALUE

`hpib_address_ctl()` returns 0 (zero) if successful or –1 if an error was encountered.

ERRORS

`hpib_address_ctl()` fails under the following circumstances and sets `errno` (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EIO]	a timeout occurred.
[EINTR]	the request was interrupted by a signal.
[EINVAL]	*ba* is not in the range of 0-30.

AUTHOR

`hpib_address_ctl()` was developed by HP.

SEE ALSO

dup(2), creat(2), fcntl(2), open(2), io_reset(3I).

NAME

hpib_atn_ctl() - control the Attention line on HP-IB

SYNOPSIS

```
#include <dvio.h>

int hpib_atn_ctl(int eid, int flag);
```

DESCRIPTION

`hpib_atn_ctl()` enables/disables the Attention (ATN) line, depending on the value of *flag*. *eid* is an entity identifier of an open HP-IB raw bus device file obtained from an `open()`, `dup()`, `fcntl()`, or `creat()` call. *flag* is an integer which, if non-zero, enables the ATN line, and otherwise disables it.

RETURN VALUE

`hpib_atn_ctl()` returns 0 (zero) if successful or –1 if an error was encountered.

ERRORS

`hpib_atn_ctl()` fails under the following circumstances, and sets `errno` (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EIO]	the interface is not the active controller or a timeout occurred.

AUTHOR

`hpib_atn_ctl()` was developed by HP.

NAME

hpib_bus_status() - return status of HP-IB interface

SYNOPSIS

```
#include <dvio.h>

int hpib_bus_status(int eid, int status);
```

DESCRIPTION

`hpib_bus_status()` obtains status information about an HP-IB channel. *eid* is an entity identifier of an open HP-IB raw bus device file obtained from an `open()`, `dup()`, `fcntl()`, or `creat()` call. *status* is an integer determining what status information is returned for a particular call. The values defined for *status* and their associated meanings are:

`REMOTE_STATUS`	Is the channel currently in remote state?
`SRQ_STATUS`	What is the current state of the SRQ line?
`NDAC_STATUS`	What is the current state of the NDAC line?
`SYS_CONT_STATUS`	Is the channel currently system controller?
`ACT_CONT_STATUS`	Is the channel currently active controller?
`TALKER_STATUS`	Is the channel currently addressed as talker?
`LISTENER_STATUS`	Is the channel currently addressed as listener?
`CURRENT_BUS_ADDRESS`	What is the channel's bus address?

The remote-state status is not defined when the interface is the active controller, although reading remote-state status in such a situation is not an error. Determining the status of the NDAC line is not available on all machines, and its use is therefore discouraged to ensure compatibility among various systems. Machines that do not support sensing the NDAC line return an error.

RETURN VALUE

The value returned by `hpib_bus_status()` depends upon the value of *status*. If *status* is `CURRENT_BUS_ADDRESS`, the return value is either the HP-IB bus address or –1 if an error occurred. If *status* is any of the other values, the return value is 0 if the condition is false (the line is clear), 1 if the condition is true (the line is set), or –1 if an error occurred.

ERRORS

`hpib_bus_status()` fails under the following conditions, and sets `errno` (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EINVAL]	*status* is not one of the values specified above.

DEPENDENCIES

Series 300/400:

The status of signal lines being driven by the interface is undefined, although reading them in such a situation is not an error. Non-active controllers cannot sense the SRQ line. Active listeners cannot sense the NDAC line.

The HP 98625A/B HP-IB interface cannot determine the current state of the NDAC line. Attempts to read this line fail and set `errno` (see *errno*(2)) to EINVAL.

AUTHOR

`hpib_bus_status()` was developed by HP.

NAME

hpib_card_ppoll_resp() - control response to parallel poll on HP-IB

SYNOPSIS

```
#include <dvio.h>

int hpib_card_ppoll_resp(int eid, int flag);
```

DESCRIPTION

`hpib_card_ppoll_resp()` enables or disables an interface for parallel polls. It also controls the sense, and determines the line on which the response is sent. This provides a means for the interface to ignore or respond to a parallel poll according to whether it is enabled to respond.

eid is an entity identifier of an open HP-IB raw bus device file obtained from an `open()`, `dup()`, `fcntl()`, or `creat()` call. *flag* is an integer having one of the following bit patterns:

Bit Pattern	Meaning
10000	Disable parallel poll response.
0SPPP	Enable parallel poll response, where
	S = sense of the response, and
	PPP = 3-bit binary number specifying the line on which the response is sent where the octal values 0 through 7 correspond to lines DIO1 through DIO8.

RETURN VALUE

`hpib_card_ppoll_resp()` returns 0 (zero) if successful, or –1 if an error was encountered.

ERRORS

`hpib_card_ppoll_resp()` fails under the following circumstances, and sets `errno` (see *errno*(2)) to the value in square brackets:

[EACCES]	The interface associated with this *eid* is locked by another process and `O_NDELAY` is set for this *eid* (see *io_lock*(3I)).
[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EINVAL]	the device cannot respond on the line number specified by *flag*.
[ETIMEDOUT]	a timeout occurred.

DEPENDENCIES

Series 300/400:

The HP 98625A/B and HP 25560A HP-IB interfaces support only enabling and disabling the parallel poll response (bit 4 of *flag*). The sense and response line number are not programmable on this card.

EIO is returned if a timeout occurs.

Series 800:

Since the sense and response line number are not programmable on the HP27110B HP-IB interface, the equivalent parallel poll configuration commands are sent over the HP-IB to the interface. Therefore, this function fails if the interface is not active controller.

AUTHOR

`hpib_card_ppoll_resp()` was developed by HP.

SEE ALSO

dup(2), creat(2), fcntl(2), open(2), hpib_ppoll(3I), hpib_ppoll_resp_ctl(3I).

NAME

hpib_eoi_ctl() - control EOI mode for HP-IB file

SYNOPSIS

```
#include <dvio.h>

int hpib_eoi_ctl(int eid, int flag);
```

DESCRIPTION

`hpib_eoi_ctl()` enables you to turn EOI mode on or off. *eid* is an entity identifier of an open HP-IB raw device file obtained from an `open()`, `dup()`, `fcntl()`, or `creat()` call. *flag* is an integer which, if non-zero, enables EOI mode, and otherwise disables it.

EOI mode causes the last byte of all subsequent write operations to be written out with the EOI line asserted, signifying the end of the data transmission. By default, EOI mode is disabled when the device file is opened.

Entity identifiers for the same device file obtained by separate `open()` requests have their own EOI modes associated with them. Entity identifiers for the same device file obtained by `dup()` or inherited by a `fork()` request share the same EOI mode. In the latter case, if one process enables EOI mode, then EOI mode is in effect for all such entity identifiers.

RETURN VALUE

`hpib_eoi_ctl()` returns a 0 (zero) if successful, or –1 if an error was encountered.

ERRORS

`hpib_eoi_ctl()` fails under any of the following circumstances and sets `errno` (see *errno*(2)) to the value in square brackets:

[EBADF] *eid* does not refer to an open file.

[ENOTTY] *eid* does not refer to an HP-IB device file.

DEPENDENCIES

Series 800:

EOI mode is enabled when the device file is first opened.

AUTHOR

`hpib_eoi_ctl()` was developed by HP.

NAME

hpib_io() - perform I/O with an HP-IB channel from buffers

SYNOPSIS

```
#include <dvio.h>

int hpib_io(int eid, struct iodetail *iovec, size_t iolen);
```

DESCRIPTION

`hpib_io()` performs and controls read and/or write operations on the specified HP-IB bus. *eid* is an entity identifier of an open HP-IB raw bus device file obtained from an `open()`, `dup()`, `fcntl()`, or `creat()` call.

Parameters are as follows:

iovec Pointer to an array of structures of the form:

```
struct iodetail {
    char  mode;
    char  terminator;
    int   count;
    char  *buf;
};
```

The *iodetail* structure is defined in the include file <`dvio.h`>.

iolen
Specifies the number of structures in *iovec*.

iodetail Structure

Elements in the `iodetail` structure are:

mode Describes what is to be done during I/O on the buffer pointed to by *buf*. *mode* is constructed by OR-ing flags from the following list:

One and only one of the following two flags *must* be specified:

`HPIBREAD` Perform a read of the HP-IB bus, placing data into the accompanying buffer.

`HPIBWRITE` Perform a write to the HP-IB bus, using data from the accompanying buffer.

The following flags can be used in most combinations (not all combinations are valid), or not at all:

`HPIBATN` Data is written with ATN enabled.

`HPIBEOI` Data written is terminated with EOI (this flag is ignored when `HPIBATN` is enabled).

`HPIBCHAR` Data read is terminated with the character given in the *terminator* element of the *iodetail* structure.

terminator
Describes the termination character, if any, that should be checked for on input. *count* is an integer specifying the maximum number of bytes to be transferred.

A read operation terminates when either *count* is matched, an EOI is detected, or the designated *terminator* is detected (if `HPIBCHAR` is set in *mode*).

A write operation terminates when *count* is matched, and the final byte is sent with EOI asserted (if `HPIBEOI` is set in *mode*).

If `HPIBATN` is set in *mode*, write operations occur with ATN enabled. Setting `HPIBATN` for a read operation is ignored and has no effect.

The members of the `iovec` array are accessed in order.

RETURN VALUE

If all transactions are successful, `hpib_io()` returns a zero and updates the *count* element in each

structure in the `iovec` array to reflect the actual number of bytes read or written.

If an error is encountered during a transaction defined by an element of `iovec`, `hpib_io()` returns without completing any transactions that might follow. In particular, if an error occurs, `hpib_io()` returns a –1, and the *count* element of the transaction that caused the error is set to –1.

ERRORS

`hpib_io()` fails under any of the following circumstances, and sets `errno` (see *errno*(2)) to the value indicated:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[ETIMEDOUT]	a timeout occurred.
[EIO]	*eid* is not the active controller.

DEPENDENCIES

Series 300/400:

EIO is returned if a timeout occurs.

Series 800:

If the interface is not currently the active controller, `hpib_io()` sets `errno` to [EACCES] instead of to [EIO].

AUTHOR

`hpib_io()` was developed by HP.

NAME

hpib_parity_ctl() - enable/disable odd parity on ATN commands

SYNOPSIS

```
#include <dvio.h>

int hpib_parity_ctl(int eid, int flag);
```

DESCRIPTION

`hpib_parity_ctl()` enables/disables the sending of odd parity for ATN command sequences depending upon the value of *flag*. *eid* is an entity identifier of an open HP-IB raw bus device file obtained from an `open()`, `dup()`, `fcntl()`, or `creat()` call. *flag* is an integer which, if non-zero, enables odd parity and otherwise disables it.

Entity identifiers for the same device file obtained by separate `open()` requests have their own parity flags associated with them. Entity identifiers for the same device file obtained by `dup()` or inherited by a `fork()` request share the same parity flag. In the latter case, if one process changes the parity flag, the new parity flag is in effect for all such entity identifiers.

RETURN VALUE

`hpib_parity_ctl()` returns 0 (zero) if successful, or –1 if an error was encountered.

ERRORS

`hpib_parity_ctl()` fails under the following circumstances, and sets `errno` (see *errno*(2)) to the value in square brackets:

[EBADF] *eid* does not refer to an open file.

[ENOTTY] *eid* does not refer to an HP-IB raw bus device file.

AUTHOR

`hpib_parity_ctl()` was developed by HP.

NAME

hpib_pass_ctl() - change active controllers on HP-IB

SYNOPSIS

```
#include <dvio.h>

int hpib_pass_ctl(int eid, int ba);
```

DESCRIPTION

`hpib_pass_ctl()` passes control of a bus to an inactive controller on that bus. The inactive controller becomes the active controller of that bus. *eid* is an entity identifier of an open HP-IB raw bus device file obtained from an `open()`, `dup()`, `fcntl()`, or `creat()` call. *ba* is the bus address of the intended device.

Not all devices can accept control. Pass control passes only active control of the bus; it cannot pass system control of the bus. The specified interface must be the current active controller but need not be the system controller. The pass control operation does not suspend program execution if the inactive controller does not take active control of the bus. However, the interface is no longer active controller.

RETURN VALUE

`hpib_pass_ctl()` returns 0 (zero) if successful, or –1 if an error was encountered.

ERRORS

`hpib_pass_ctl()` fails under any of the following circumstances, and sets `errno` (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EIO]	the interface is not the active controller.
[ETIMEDOUT]	a timeout occurred.
[EINVAL]	*ba* is not a valid HP-IB bus address.
[EACCES]	The interface associated with this *eid* is locked by another process and `O_NDELAY` is set for this *eid* (see *io_lock*(3I)).

DEPENDENCIES

Series 300/400:

EIO is returned if a timeout occurs.

Series 800:

If the interface is not currently the active controller, `hpib_pass_ctl()` sets `errno` to [EACCES] instead of to [EIO].

AUTHOR

`hpib_pass_ctl()` was developed by HP.

NAME

hpib_ppoll() - conduct parallel poll on HP-IB bus

SYNOPSIS

```
#include <dvio.h>

int hpib_ppol(int eid);
```

DESCRIPTION

`hpib_ppoll()` conducts a parallel poll on an HP-IB bus. *eid* is an entity identifier of an open HP-IB raw bus device file obtained from an `open()`, `dup()`, `fcntl()`, or `creat()` call.

Devices enabled to respond that are in need of service can then assert the appropriate DIO line. This enables the controller to determine which devices, if any, need service at a given time. `hpib_ppoll()` delays for 25 microseconds before returning with the response. The interface must be the active controller to conduct a parallel poll.

RETURN VALUE

`hpib_ppoll()` returns an integer value whose least significant byte corresponds to the byte formed by the eight data input/output (DIO) lines. Devices enabled to respond to a parallel poll do so on the appropriate DIO line. DIO line 1 corresponds to the least significant bit in the response byte; line 8 to the most significant bit. A return value of −1 indicates that an error occurred.

ERRORS

`hpib_ppoll()` fails under the following situations, and sets `errno` (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EIO]	the interface is not current the active controller.

AUTHOR

`hpib_ppoll()` was developed by the Hewlett-Packard Company.

NAME

hpib_ppoll_resp_ctl() - define interface parallel poll response

SYNOPSIS

```
#include <dvio.h>

int hpib_ppoll_resp_ctl(int eid, int response);
```

DESCRIPTION

`hpib_ppoll_resp_ctl()` defines a response to be sent when an active controller performs a parallel poll on an HP-IB interface. *eid* is an entity identifier of an open HP-IB raw bus device file, obtained from an `open()`, `dup()`, `fcntl()`, or `creat()` call.

The value of *response* indicates whether this computer does or does not need service. A non-zero *response* value indicates that service is required. This statement only sets up a potential response; no actual response is generated when the statement is executed. The sense of the response and the line number to respond on are set by `hpib_card_ppoll_resp()` (see *hpib_card_ppoll_resp*(3I)) or by the active controller.

RETURN VALUE

`hpib_ppoll_resp_ctl()` returns 0 if the response is successfully set, or –1 if an error has occured.

ERRORS

`hpib_ppoll_resp_ctl()` fails under the following situations, and sets `errno` (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to a raw HP-IB device file.
[EACCES]	The interface associated with this *eid* is locked by another process and `O_NDELAY` is set for this *eid* (see io_lock(3I)).

AUTHOR

`hpib_ppoll_resp_ctl()` was developed by HP.

SEE ALSO

hpib_ppoll(3I), hpib_card_ppoll_resp(3I)

NAME

hpib_ren_ctl() - control the Remote Enable line on HP-IB

SYNOPSIS

```
#include <dvio.h>

int hpib_ren_ctl(int eid, int flag);
```

DESCRIPTION

`hpib_ren_ctl()` enables/disables the Remote Enable (REN) line depending upon the value of *flag*. *eid* is an entity identifier of an open HP-IB raw bus device file obtained from an `open()`, `dup()`, `fcntl()`, or `creat()` call. *flag* is an integer which, if non-zero, enables the REN line, and otherwise disables it.

`hpib_ren_ctl()` can be used in conjunction with `hpib_send_cmnd()` to place devices into the remote state or local state. The REN line is normally enabled at all times, and is in this state at power-up. Only the system controller can enable or disable the REN line.

RETURN VALUE

`hpib_ren_ctl()` returns 0 (zero) if successful, or –1 if an error was encountered.

ERRORS

`hpib_ren_ctl()` fails under the following circumstances, and sets `errno` (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EIO]	the interface is not the system controller.

AUTHOR

`hpib_ren_ctl()` was developed by HP.

NAME

hpib_rqst_srvce() - allow interface to enable SRQ line on HP-IB

SYNOPSIS

```
#include <dvio.h>

int hpib_rqst_srvce(int eid, int cv);
```

DESCRIPTION

hpib_rqst_srvce() specifies a response byte to be sent by the interface when it is serially polled by the active controller. *eid* is an entity identifier of an open HP-IB raw bus device file obtained from an **open()**, **dup()**, **fcntl()**, or **creat()** call. *cv* is an integer control value representation of the desired response byte.

hpib_rqst_srvce() optionally enables the SRQ line depending upon the response byte. If bit 6 of the response byte is set, the SRQ line is enabled. It remains enabled until the active controller conducts a serial poll or until the computer executes the request function with bit 6 cleared. However, the SRQ line is not enabled as long as the interface is active controller. If bit 6 is set, the interface remembers its response byte, and enables the SRQ line when control is passed to another device on the bus.

The response byte is structured as follows:

Bit	Meaning
0	SPOLL bit (least significant bit of response byte)
1	SPOLL bit
2	SPOLL bit
3	SPOLL bit
4	SPOLL bit
5	SPOLL bit
6	SRQ line
7	SPOLL bit (most significant bit of response byte)

RETURN VALUE

hpib_rqst_srvce() returns 0 (zero) if successful, or −1 if an error was encountered.

ERRORS

hpib_rqst_srvce() fails under the following circumstances, and sets **errno** (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[ETIMEDOUT]	a timeout occurred.
[EACCES]	The interface associated with this *eid* is locked by another process and **O_NDELAY** is set for this *eid* (see io_lock(3I)).

DEPENDENCIES

Series 300/400:

The HP98625A/B and HP25560A HP-IB interface cards allow only bit 6 to be set. All other bits are cleared.

EIO is returned if a timeout occurs.

Series 800:

The HP27110B HP-IB interface card allows only bit 6 to be set. All other bits are cleared.

AUTHOR

hpib_rqst_srvce() was developed by HP.

NAME

hpib_send_cmnd( ) - send command bytes over HP-IB

SYNOPSIS

```
#include <dvio.h>

int hpib_send_cmnd(int eid, const char *ca, int length);
```

DESCRIPTION

`hpib_send_cmnd()` sends specified arbitrary bytes of information on the HP-IB with the ATN line asserted, providing a means to configure and control the bus. *eid* is an entity identifier of an open HP-IB raw bus device file obtained from an `open()`, `dup()`, `fcntl()`, or `creat()` call. *ca* is a character pointer to a string of bytes to be written to the HP-IB bus as commands. *length* is an integer specifying the number of bytes in the string pointed to by *ca*.

The interface must currently be the active controller in order to send commands over the bus.

Note that for all HP-IB interfaces, both built-in and plug-in, the most significant bit of each byte is overwritten with a parity bit. All commands are written with odd parity.

RETURN VALUE

`hpib_send_cmnd()` returns 0 (zero) if successful, or –1 if an error was encountered.

ERRORS

`hpib_send_cmnd()` fails under the following circumstances, and sets `errno` (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EIO]	the interface is not currently the active controller.
[ETIMEDOUT]	a timeout occurred.
[EACCES]	The interface associated with this *eid* is locked by another process and `O_NDELAY` is set for this *eid* (see *io_lock*(3I)).
[EINVAL]	The value specified for *length* is invalid, either less than or equal to 0 or greater than `MAX_HPIB_COMMANDS` as defined in `<dvio.h>`.

DEPENDENCIES

Series 300/400:

EIO is returned if a timeout occurs.

Series 800:

If the interface is not currently the active controller, `hpib_send_cmnd()` sets `errno` to [EACCES] instead of [EIO].

AUTHOR

`hpib_send_cmnd()` was developed by HP.

SEE ALSO

dup(2), creat(2), fcntl(2), open(2), hpib_parity_ctl(3I).

NAME

hpib_spoll() - conduct a serial poll on HP-IB bus

SYNOPSIS

```
#include <dvio.h>

int hpib_spoll(int eid, int ba);
```

DESCRIPTION

`hpib_spoll()` conducts a serial poll of the specified device. *eid* is an entity identifier of an open HP-IB raw bus device file obtained from an `open()`, `dup()`, `fcntl()`, or `creat()` call. *ba* is the bus address of the intended device.

`hpib_spoll()` Polls a single device for its response byte. The information stored in the response byte is device specific with the exception of bit 6. If bit 6 of the response byte is set, the addressed device has asserted the SRQ line, and is requesting service (note that the least significant bit of the response byte is bit 0).

Not all devices respond to the serial poll function. Consult device documentation. Specifying a device that does not support serial polling may cause a timeout error or suspend your program indefinitely. The interface cannot serial poll itself. The interface must be the active controller.

RETURN VALUE

If `hpib_spoll()` is successful, the device response byte is returned in the least significant byte of the return value. Otherwise, –1 is returned, indicating an error.

ERRORS

`hpib_spoll()` fails under the following circumstances, and sets `errno` (see *errno*(2)) to the value in square brackets:

[EBAD]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[EIO]	the interface is not the active controller.
[ETIMEDOUT]	the device polled did not respond before timeout.
[EINVAL]	*ba* is the address of the polling interface itself.
[EACCES]	The interface associated with this *eid* is locked by another process and `O_NDELAY` is set for this *eid* (see io_lock(3I)).

DEPENDENCIES

Series 300/400:

EIO is returned if a timeout occurs.

Series 800:

If the interface is not currently the active controller, `hpib_spoll()` sets `errno` to [EACCES] instead of to [EIO].

AUTHOR

`hpib_spoll()` was developed by HP.

SEE ALSO

dup(2), creat(2), fcntl(2), open(2), hpib_rqst_srvce(3I).

NAME

hpib_status_wait() - wait until the requested status condition becomes true

SYNOPSIS

```
#include <dvio.h>

int hpib_status_wait(int eid, int status); #include <dvio.h>
```

DESCRIPTION

`hpib_status_wait()` waits until a specific condition has occurred before returning. *eid* is an entity identifier of an open HP-IB raw bus device file obtained from an `open()`, `dup()`, `fcntl()`, or `creat()` call. *status* is an integer specifying what information is returned. The possible values for *status* and their associated meanings are:

`WAIT_FOR_SRQ`	Wait until SRQ line is enabled.
`WAIT_FOR_CONTROL`	Wait until this channel is active controller.
`WAIT_FOR_TALKER`	Wait until this channel is addressed as talker.
`WAIT_FOR_LISTENER`	Wait until this channel is addressed as listener.

The wait is subject to the current timeout in effect. If a timeout occurs before the desired condition occurs, the function returns with an error.

RETURN VALUE

`hpib_status_wait()` returns zero when the condition requested becomes true. A value of –1 is returned if an error occurs. A –1 is also returned if a timeout occurs before the desired condition becomes true.

ERRORS

`hpib_status_wait()` fails under the following circumstances, and sets `errno` (see *errno*(2)) to the value in square brackets:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB raw bus device file.
[ETIMEDOUT]	a timeout occurred.
[EINVAL]	*status* contains an invalid value.
[EACCES]	the interface associated with this *eid* is locked by another process and `O_NDELAY` is set for this *eid* (see io_lock(3I)).

DEPENDENCIES

Series 300/400:

EIO is returned if a timeout occurs.

The following error is also defined:

[EIO]	the device is active controller and *status* specifies `WAIT_FOR_TALKER` or `WAIT_FOR_LISTENER`.

AUTHOR

`hpib_status_wait()` was developed by HP.

NAME

hpib_wait_on_ppoll() - wait until a particular parallel poll value occurs

SYNOPSIS

```
#include <dvio.h>

int hpib_wait_on_ppoll(int eid, int mask, int sense);
```

DESCRIPTION

`hpib_wait_on_ppoll()` waits for a parallel poll response to occur on one or more lines. *eid* is an entity identifier of an open HP-IB raw bus device file.

The *mask* argument specifies on which lines the parallel poll response is expected. The value of *mask* is treated as an eight-bit binary number where the least significant bit corresponds to line DIO1; the most significant bit to DIO8. For example, if you want to wait for a response on lines DIO2 and DIO6, the corresponding binary number is 00010010, so a hexadecimal value of 12 should be passed as the *mask* argument.

The *sense* argument specifies what response is expected on the selected lines. The value of *sense* is constructed in the same way as *mask*; eight bits for eight lines. If a bit in *sense* is set, the function returns when the line corresponding to that bit is *cleared*. If a bit in *sense* is clear, the function returns when the corresponding line is *set*. Using the previous example, if *mask* is 0x12 and *sense* is 00000010 (0x02 hexadecimal), the function returns when line DIO5 is set, or when line DIO2 is clear.

RETURN VALUE

`hpib_wait_on_ppoll()` returns a value of −1 if an error or timeout condition occurs. Upon successful completion, the function returns the response byte XOR-ed with the *sense* value and AND-ed with the *mask*.

ERRORS

`hpib_wait_on_ppoll()` fails and sets `errno` to indicate the error if any of the following is true:

[EACCES]	The interface associated with this *eid* is locked by another process and *O_NDELAY* is set for this *eid* (see *io_lock*(3I)).
[EBADF]	The *eid* argument is not a valid open entity identifier.
[ENOTTY]	The *eid* argument does not refer to an HP-IB raw bus device file.
[EINVAL]	An invalid mask is received.
[EIO]	The interface is not currently the active controller.
[EIO]	A timeout occurred (Series 300/400 only).
[ETIMEDOUT]	A timeout occurred (Series 800 only).

DEPENDENCIES

Series 300/400:

[EIO] is returned if a timeout occurs.

Series 800:

If the interface is not currently the active controller, `hpib_wait_on_ppoll()` sets `errno` to [EACCES] instead of to [EIO].

AUTHOR

`hpib_wait_on_ppoll()` was developed by HP.

NAME

HPPACADDD, HPPACCMPD, HPPACCVAD, HPPACCVBD, HPPACCVDA, HPPACCVDB, HPPACDIVD, HPPACLONGDIVD, HPPACMPYD, HPPACNSLD, HPPACSLD, HPPACSRD, HPPACSUBD - 3000-mode packed-decimal library

SYNOPSIS

```
#include <hppac.h>

void HPPACADDD(
    unsigned char *operand2,
    int op2digs,
    unsigned char *operand1,
    int op1digs,
    enum HPPAC_CC *compcode,
    int *pacstatus
);

void HPPACCMPD(
    unsigned char *operand1,
    int op1digs,
    unsigned char *operand2,
    int op2digs,
    enum HPPAC_CC *compcode,
    int *pacstatus
);

void HPPACCVAD(
    unsigned char *target,
    int targetdigs,
    unsigned char *source,
    int sourcedigs,
    enum HPPAC_CC *compcode,
    int *pacstatus
);

void HPPACCVBD(
    unsigned char *target,
    int targetdigs,
    unsigned short *source,
    int sourcewords,
    enum HPPAC_CC *compcode,
    int *pacstatus
);

void HPPACCVDA(
    unsigned char *target,
    int targetdigs,
    unsigned char *source,
    int sign_control,
    enum HPPAC_CC *compcode,
    int *pacstatus
);

void HPPACCVDB(
    unsigned short *target,
    unsigned char *source,
    int sourcedigs,
    enum HPPAC_CC *compcode,
    int *pacstatus
);

void HPPACDIVD(
    unsigned char *operand2,
    int op2digs,
```

```
    unsigned char *operand1,
    int op1digs,
    enum HPPAC_CC *compcode,
    int *pacstatus
);
void HPPACLONGDIVD(
    unsigned char *operand2,
    int op2digs,
    unsigned char *operand1,
    int op1digs,
    enum HPPAC_CC *compcode,
    int *pacstatus
);
void HPPACMPYD(
    unsigned char *operand2,
    int op2digs,
    unsigned char *operand1,
    int op1digs,
    enum HPPAC_CC *compcode,
    int *pacstatus
);
void HPPACNSLD(
    unsigned char *operand2,
    int op2digs,
    unsigned char *operand1,
    int op1digs,
    int *shift_amt,
    enum HPPAC_CC *compcode,
    int *pacstatus,
    int *carry
);
void HPPACSLD(
    unsigned char *operand2,
    int op2digs,
    unsigned char *operand1,
    int op1digs,
    int shift_amt,
    enum HPPAC_CC *compcode,
    int *pacstatus,
    int *carry
);
void HPPACSRD(
    unsigned char *operand2,
    int op2digs,
    unsigned char *operand1,
    int op1digs,
    int shift_amt,
    enum HPPAC_CC *compcode,
    int *pacstatus
);
void HPPACSUBD(
    unsigned char *operand2,
    int op2digs,
    unsigned char *operand1,
    int op1digs,
    enum HPPAC_CC *compcode,
```

```
    int *pacstatus
);
```

DESCRIPTION

This set of calls invokes the library functions for emulating 3000-mode (MPE V/E) packed-decimal operations. These functions are in library `libcl` which is searched when the option `-lcl` is used with *cc*(1) or *ld*(1).

`HPPACADDD()`
: Performs packed-decimal addition.

`HPPACCMPD()`
: Compares two packed-decimal numbers.

`HPPACCVAD()`
: Converts an ASCII representation to packed-decimal.

`HPPACCVBD()`
: Converts a binary representation to packed-decimal.

`HPPACCVDA()`
: Converts a packed-decimal number to ASCII.

`HPPACCVDB()`
: Converts a packed-decimal number to binary.

`HPPACDIVD()`
: Performs packed-decimal division.

`HPPACLONGDIVD()`
: Performs packed-decimal division (alternate routine).

`HPPACMPYD()`
: Performs packed-decimal multiplication.

`HPPACNSLD()`
: Performs a packed-decimal normalizing left shift.

`HPPACSLD()` Performs a packed-decimal left shift.

`HPPACSRD()` Performs a packed-decimal right shift.

`HPPACSUBD()`
: Performs packed-decimal subtraction.

For all operations, the value returned in the variable to which the *compcode* argument points is one of the following values of type `enum HPPAC_CC`:

`HPPAC_CCG` Result > 0 or operand1 > operand2

`HPPAC_CCL` Result < 0 or operand1 < operand2

`HPPAC_CCE` Result == 0 or operand1 == operand2

For all operations, the value returned in the variable to which the *pacstatus* argument points is one of the following values of type `enum HPPAC_STATUS`. Their meanings are intended to be obvious:

`HPPAC_NO_ERROR`

`HPPAC_DECIMAL_OVERFLOW`

`HPPAC_INVALID_ASCII_DIGIT`

`HPPAC_INVALID_PACKED_DECIMAL_DIGIT`

`HPPAC_INVALID_SOURCE_WORD_COUNT`

`HPPAC_INVALID_DECIMAL_OPERAND_LENGTH`

`HPPAC_DECIMAL_DIVIDE_BY_ZERO`

AUTHOR

The HPPAC library was developed by HP.

SEE ALSO
Compiler Library /XL Reference Manual

NAME

hsearch(), hcreate(), hdestroy() - manage hash search tables

SYNOPSIS

```
#include <search.h>

ENTRY *hsearch(ENTRY item, ACTION action);

int hcreate(unsigned nel);

void hdestroy(void);
```

DESCRIPTION

`hsearch()` is a hash-table search routine generalized from Knuth (6.4) Algorithm D. It returns a pointer into a hash table indicating the location at which an entry can be found. *item* is a structure of type **ENTRY** (defined in the `<search.h>` header file) containing two pointers: points to the comparison key, and points to any other data to be associated with that key. (Pointers to types other than character should be cast to pointer-to-character.) *action* is a member of an enumeration type **ACTION** indicating the disposition of the entry if it cannot be found in the table. **ENTER** indicates that the item should be inserted in the table at an appropriate point. **FIND** indicates that no entry should be made. Unsuccessful resolution is indicated by the return of a NULL pointer.

`hcreate()` allocates sufficient space for the table, and must be called before `hsearch()` is used. *nel* is an estimate of the maximum number of entries that the table will contain. This number can be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

`hdestroy()` destroys the search table, and can be followed by another call to `hcreate()`.

EXAMPLE

The following example reads in strings followed by two numbers and stores them in a hash table, discarding duplicates. It then reads in strings and finds the matching entry in the hash table and prints it out.

```
#include <stdio.h>
#include <search.h>

struct info {         /* this is the info stored in the table */
     int age, room;   /* other than the key. */
};
#define NUM_EMPL    5000    /* # of elements in search table */

main( )
{
     /* space to store strings */
     char string_space[NUM_EMPL*20];

     /* space to store employee info */
     struct info info_space[NUM_EMPL];

     /* next avail space in string_space */
     char *str_ptr = string_space;

     /* next avail space in info_space */
     struct info *info_ptr = info_space;
     ENTRY item, *found_item, *hsearch( );
     /* name to look for in table */

     char name_to_find[30];
     int i = 0;

     /* create table */
     (void) hcreate(NUM_EMPL);
     while (scanf("%s%d%d", str_ptr, &info_ptr->age,
            &info_ptr->room) != EOF && i++ < NUM_EMPL) {
```

```
                /* put info in structure, and structure in item */
                item.key = str_ptr;
                item.data = (char *)info_ptr;
                str_ptr += strlen(str_ptr) + 1;
                info_ptr++;

                /* put item into table */
                (void) hsearch(item, ENTER);
            }

            /* access table */
            item.key = name_to_find;
            while (scanf("%s", item.key) != EOF) {
                if ((found_item = hsearch(item, FIND)) != NULL) {

                /* if item is in the table */
                (void)printf("found %s, age = %d, room = %d\n",
                        found_item->key,
                        ((struct info *)found_item->data)->age,
                        ((struct info *)found_item->data)->room);
                } else {
                (void)printf("no such employee %s\n",
                        name_to_find);
                }
            }
        }
```

RETURN VALUE

`hsearch()` returns a NULL pointer if either the action is **FIND** and the item could not be found or the action is **ENTER** and the table is full.

`hcreate()` returns zero if it cannot allocate sufficient space for the table.

WARNINGS

`hsearch()` and `hcreate()` use `malloc()` to allocate space (see *malloc*(3C)).

Only one hash search table can be active at any given time.

SEE ALSO

bsearch(3C), lsearch(3C), malloc(3C), string(3C), tsearch(3C).

STANDARDS CONFORMANCE

`hsearch()`: AES, SVID2, XPG2, XPG3, XPG4

`hcreate()`: AES, SVID2, XPG2, XPG3, XPG4

`hdestroy()`: AES, SVID2, XPG2, XPG3, XPG4

NAME

hypot(), cabs() - Euclidean distance function, complex absolute value

SYNOPSIS

```
#include <math.h>

double hypot(double x, double y);

double cabs(struct {double x, y;} z);
```

DESCRIPTION

`hypot()` and `cabs()` return `sqrt` (*x***x*+*y***y*), taking precautions against unwarranted overflows.

`hypot()` and `cabs()` return +INFINITY when *x* or *y* is ±INFINITY.

ERRORS

/lib/libm.a

When the correct value would overflow, `hypot()` and `cabs()` return HUGE_VAL and set `errno` to ERANGE.

`hypot()` and `cabs()` return NaN and set `errno` to EDOM when *x* or *y* is NaN.

These error-handling procedures can be changed with the `matherr()` function (see *matherr*(3M)).

/lib/libM.a

No error messages are printed on the standard error output.

When the correct value would overflow, `hypot()` and `cabs()` return HUGE_VAL and set `errno` to ERANGE.

`hypot()` and `cabs()` return NaN and set `errno` to EDOM when *x* or *y* is NaN.

These error-handling procedures can be changed by using the `_matherr()` function (see *matherr*(3M)). Note that `_matherr()` is provided in order to assist in migrating programs from `libm.a` to `libM.a` and is *not* a part of XPG3, ANSI C, or POSIX.

SEE ALSO

isinf(3M), isnan(3M), matherr(3M).

STANDARDS CONFORMANCE

`hypot()` in libm.a: AES, SVID2, XPG2, XPG3
`hypot()` in libM.a: AES, XPG3, XPG4

NAME

iconvsize, iconvopen, iconvclose, iconvlock, ICONV, ICONV1, ICONV2 - code set conversion routines

SYNOPSIS

```
#include <iconv.h>

int iconvsize(const char *tocode, const char *fromcode);

iconvd iconvopen(const char *tocode, const char *fromcode,
      unsigned char *table, int d1, int d2);

int iconvclose(iconvd cd);

int iconvlock(iconvd cd, int direction, int lock, const char *s);

int ICONV(iconvd cd, const unsigned char **inchar, size_t
      *inbytesleft, unsigned char **outchar, size_t *outbytesleft);

int ICONV1(iconvd cd, unsigned char *to,
      unsigned char *from, size_t buflen);

int ICONV2(iconvd cd, unsigned char *to,
      unsigned char *from, size_t buflen);
```

Remarks

For conformance to standards currently under development, the interfaces described in this manual entry may be replaced with others in a future release. To make migration easier, application writers should take care to isolate use of these functions.

DESCRIPTION

`iconvsize()` Find the size of a table if one is needed to convert characters from the code set specified by the *fromcode* argument to the code set specified by the *tocode* argument. If a conversion table is needed and the table exists, the size of the table in bytes is returned. If a table is needed and the table does not exist, -1 is returned. If a conversion table is not needed, 0 is returned.

`iconvopen()` Perform all initializations that have to be done to convert characters from the code set specified by the *fromcode* argument to the code set specified by the *tocode* argument and return a conversion descriptor of type `iconvd` that identifies the conversion. Up to `MAX_CD` conversions can be open simultaneously. See *iconv*(1) for HP-supplied *fromcode* and *tocode* names and their corresponding code sets. For conversions that require a table, the *table* argument is a pointer to the start of the conversion table. It is the caller's responsibility to allocate sufficient memory for the table which is given by `iconvsize()`. For conversions that do not require a table, the *table* argument must be a NULL pointer. `iconvsize()` can be used to determine whether a table is needed. For multi-byte code sets, a "converted from" character is mapped to a default character (*d1* or *d2*) if it does not have an equivalent in the "converted to" code set. Currently supported multi-byte code sets can have character lengths of one or two bytes. If a one-byte character is unmapped, the default one-byte character *d1* is used. Similarly, if a two-byte character is unmapped, the default two-byte character *d2* is used. Default characters are used since different multi-byte code sets typically do not have the same number of characters which makes a one-to-one mapping difficult. Also, unused sections in multi-byte code sets are usually reserved for future use. A different approach is taken with single-byte code sets. For single-byte code sets, it is assumed that the translation table forces a one-to-one mapping between the "from" and "to" characters. No default characters are used with single-byte code sets. This one-to-one mapping guarantees that the conversion is reversible. For example, if the output of a ROMAN8-to-ISO 8859/1 conversion is converted back to ROMAN8, the result of this double conversion is the same as the original data.

`iconvclose()` Close the conversion descriptor *cd* freeing it up for a subsequent `iconvopen()`. It is the caller's responsibility to de-allocate any table associated with the *cd* conversion descriptor.

If needed, code set lock-shift information for the conversion identified by *cd* can be initialized by `iconvlock()`. If *direction* is 0, string *s* is used as a lock-shift sequence for

the "converted from" or input data. If *direction* is 1, string *s* is used as a lock-shift sequence for the "converted to" or output data. Currently, three lock-shift sequences can be used in a conversion: lock-shift 0, lock-shift 1 and lock-shift 2. These are identified by the *lock* parameter values 0, 1 and 2. `iconvlock()` also resets any state information to the initial shift state.

`ICONV()` Fetch a character in the "converted from" code set from an input buffer, convert the character to the "converted to" code set and place it plus any lock-shift information into an output buffer. The descriptor *cd* identifies the conversion. The contents of *inchar* points to a single- or multi-byte character in the input buffer and *inbytesleft* points to the number of bytes from the input character to the end of the buffer. The contents of *outchar* points to the next available space in the output buffer and *outbytesleft* points to the number of the bytes from the next available space to the end of the buffer. While conversions are done from the input buffer to the output buffer, the contents of *inchar*, *inbytesleft*, *outchar*, and *outbytesleft* are incremented or decremented to reflect the current status of the input and output buffers.

`ICONV1()` and `ICONV2()` are used where it is more efficient to handle single- and multi-byte characters separately. These routines do not check for lock-shift information.

`ICONV1()` Convert single-byte characters in *from* according to the conversion identified by *cd* and return the converted value in *to*. `ICONV1()` assumes *from* contains only single-byte characters.

`ICONV2()` Similarly convert double-byte characters in *from* according to the conversion identified by *cd* and return the converted value in *to*. `ICONV2()` assumes *from* contains only double-byte characters.

The *buflen* argument in both `ICONV1()` and `ICONV2()` specifies the number of bytes that will be converted.

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUES

`iconvsize()` Returns the size of the conversion table in bytes if a table is needed and it exists. The function returns -1 if a table is needed and it does not exist. The function returns 0 if a table is not needed.

`iconvopen()` Returns a conversion descriptor if successful. Otherwise, a (*iconvd*) -1 is returned.

`iconvclose()` Returns a non-negative number if successful. Otherwise a -1 is returned.

`ICONV()` returns 0 if all characters from the input buffer are successfully converted and placed into the output buffer. `ICONV()` returns 1 if a multi-byte input character or a lock-shift sequence spans the input buffer boundary. No conversion is attempted on the character and the contents of *inchar* points to the start of the truncated character sequence. `ICONV()` returns 2 if an input character does not belong to the "converted from" character set. No conversion is attempted on the character and the contents of *inchar* points to the start of the unidentified input character. `ICONV()` returns 3 if there is no room in the output buffer to place the converted character. The converted character is not placed in the output buffer and the contents of *inchar* points to the start of the character sequence that caused the output buffer overflow.

`ICONV1()`
`ICONV2()` Both return the number of bytes converted if successful. Otherwise they return -1.

EXAMPLES

```
int
convert(tocode, fromcode, d1, d2)
char *tocode;                          /* tocode name */
char *fromcode;                        /* fromcode name */
int d1;                                /* one-byte default character */
int d2;                                /* two-byte default character */
```

```
{
    extern void error();            /* local error message */

    iconvd cd;                      /* conversion descriptor */
    int size;                       /* size of translation table */
    unsigned char *table;           /* ptr to translation table */
    int bytesread;                  /* num bytes read into input buffer */
    unsigned char inbuf[BUFSIZ];    /* input buffer */
    unsigned char *inchar;          /* ptr to input character */
    int inbytesleft;                /* num bytes left in input buffer */
    unsigned char outbuf[BUFSIZ];   /* output buffer */
    unsigned char *outchar;         /* ptr to output character */
    int outbytesleft;               /* num bytes left in output buffer */

    /* create conversion table */
    if ((size = iconvsize(tocode, fromcode)) == BAD) {
        error(FATAL, BAD_SIZE);
    }
    else if (size == 0) {
        table = (unsigned char *) NULL;
    }
    else if ((table=(unsigned char *)malloc((unsigned int)size))==(unsigned char *)NULL)
        error(FATAL, BAD_CREATE);
    }

    /* start up a conversion */
    if ((cd = iconvopen(tocode, fromcode, table, d1, d2)) == (iconvd) BAD) {
        error(FATAL, BAD_OPEN);
    }

    inchar = inbuf;
    inbytesleft = 0;
    outchar = outbuf;
    outbytesleft = BUFSIZ;

    /* translate the characters */
    for ( ;; ) {
        switch (ICONV(cd, &inchar, &inbytesleft, &outchar, &outbytesleft)) {
        case 0:
        case 1:
                /*
                ** Done with buffer, empty buffer or character spans
                ** input buffer boundary.  Move any remaining stuff
                ** to start of buffer, get more characters and
                ** reinitialize input variables.  If at EOF, flush
                ** output buffer and leave; otherwise, continue to
                ** convert the characters.
                */
                strncpy(inbuf, inchar, inbytesleft);
                if ((bytesread=read(Input, inbuf+inbytesleft, BUFSIZ-inbytesleft)) < 0) {
                        perror("prog");
                        return BAD;
                }
                if (! (inbytesleft += bytesread)) {
                        if (write(1, outbuf, BUFSIZ - outbytesleft) < 0) {
                                perror("prog");
                                return BAD;
                        }
                        goto END_CONVERSION;
```

```
                }
                inchar = inbuf;
                break;
        case 2:
                error(FATAL, BAD_CONVERSION);
        case 3:
                /*
                ** Full buffer or output character spans output buffer
                ** boundary.  Send the output buffer to stdout,
                ** reinitialize the output variables.
                */
                if (write(1, outbuf, BUFSIZ - outbytesleft) < 0) {
                        perror("prog");
                        return BAD;
                }
                outchar = outbuf;
                outbytesleft = BUFSIZ;
        }
    }
END_CONVERSION:

    /* end conversion & get rid of the conversion table */
    if (iconvclose(cd) == BAD) {
        error(FATAL, BAD_CLOSE);
    }
    if (size) {
        free((char *) table);
    }
    return GOOD;
}
```

AUTHOR

`iconv()` was developed by HP.

SEE ALSO

iconv(1).

NAME

copysign(), copysignf(), drem(), finite(), finitef(), logb(), scalb() - exponent manipulations

SYNOPSIS

```
#include <math.h>

double copysign(double x, double y);

double drem(double x, double y);

int finite(double x);

double logb(double x);

double scalb(double x, int n);

float copysignf(float x, float y);

int finitef(float x);
```

DESCRIPTION

These functions are required for, or recommended by, the IEEE-754 standard for floating-point arithmetic.

copysign() returns x with its sign changed to y's.

drem() returns the remainder $r=x-n*y$ where n is the integer nearest the exact value of x/y; moreover, if $|n-x/y| = 1/2$, then n is even. Consequently the remainder is computed exactly and $|r| \leq |y|/2$. But **drem(x,0)** is exceptional; see below under ERRORS.

finite() returns 1 only when -INFINITY < x < +INFINITY. Otherwise it returns 0 (i.e., when $|x|$ = INFINITY or x is NaN).

logb() returns x's exponent n, a signed integer converted to double-precision floating point and chosen such that $1 \leq |x|/2**n < 2$ unless $x = 0$ or (only on machines that conform to the IEEE-754 standard) $|x|$ = INFINITY or x lies between 0 and the underflow threshold.

scalb() returns $x*(2**n)$ computed, for integer n, without first computing $2**n$.

copysignf() and **finitef()** are **float** versions of **copysign()** and **finite()**. They are named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard. Programs must be compiled in ANSI mode (use the **-Aa** option) in order to use these functions; otherwise, the compiler promotes the **float** arguments to **double**, and the functions return incorrect results.

DEPENDENCIES

Series 300/400

These functions are not supported on Series 300/400 systems.

Series 700/800

These functions are provided in the PA1.1 versions of the math library only. The **+DA1.1** option (the default on Series 700 systems) links in a PA1.1 version automatically. A PA1.1 library can be linked in explicitly. For more information, see the *HP-UX Floating-Point Guide*.

ERRORS

The IEEE-754 standard defines **drem(x,0)** and **drem(INFINITY,y)** to be invalid operations that produce a NaN.

The IEEE-754 standard defines **logb(±INFINITY)** = +INFINITY and **logb(0)** = -INFINITY, and requires the latter to signal a division-by-zero exception.

SEE ALSO

isnan(3M), isinf(3M), fpclassify(3M).

NAME

inet_addr(), inet_network(), inet_ntoa(), inet_makeaddr(), inet_lnaof(), inet_netof() - Internet address manipulation routines

SYNOPSIS

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_addr(const char *cp);

unsigned long inet_network(const char *cp);

char *inet_ntoa(struct in_addr in);

struct in_addr inet_makeaddr(int net, int lna);

int inet_lnaof(struct in_addr in);

int inet_netof(struct in_addr in);
```

DESCRIPTION

`inet_addr()`
`inet_network()` Interpret character strings representing numbers expressed in the Internet standard "dot" notation.

`inet_addr()` returns numbers suitable for use as Internet addresses.

`inet_network()` returns numbers suitable for use as Internet network numbers>

Return values can be assigned to a `struct in_addr` (defined in `/usr/include/netinet/in.h`) by using a technique similar to the following:

```
struct in_addr addr;
char *cp;

addr.s_addr = inet_addr(cp);
```

`inet_ntoa()` Take an Internet address and return an ASCII string representing the address in "." (dot) notation.

`inet_makeaddr()` Take an Internet network number and a local network address and construct an Internet address from it.

`inet_netof()` Break apart Internet host addresses, returning the network number part.

`inet_lnaof()` Break apart Internet host addresses, returning the local network address part.

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. Bytes in HP-UX systems are ordered from left to right.

Internet Addresses:

Values specified using 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.

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 in `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 in `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 dot notation can be decimal, octal, or hexadecimal, as specified in the C language (i.e., a leading 0x or 0X implies hexadecimal; a leading 0 implies octal; otherwise, the number is interpreted as decimal).

RETURN VALUE

`inet_addr()` and `inet_network()` return –1 for malformed requests.

WARNINGS

The string returned by `inet_ntoa()` resides in a static memory area and must be saved if needed for later use.

AUTHOR

These `inet` routines were developed by the University of California, Berkeley.

SEE ALSO

gethostent(3N), getnetent(3N), hosts(4), networks(4).

NAME

initgroups() - initialize group access list

SYNOPSIS

```
#include <unistd.h>

int initgroups(const char *name, gid_t basegid);
```

DESCRIPTION

`initgroups()` reads the login group file, `/etc/logingroup`, and sets up the group access list for the user specified by *name*, using the *setgroups*(2) system call. If the value of *basegid* is zero or positive, it is automatically included in the groups list. Typically this value is given as the group number from the password file. If the login group file does not exist or is empty, *basegid* is the only member of the list.

RETURN VALUE

`initgroups()` returns `-1` if it was not invoked by a user with appropriate privileges.

WARNINGS

`initgroups()` uses the routines based on *getgrent*(3C). If the invoking program uses any of these routines, the group structure is overwritten by the call to `initgroups()`. Subsequent calls to `initgroups()` with the same *name* parameter override the actions of previous calls.

On many systems, no one seems to keep `/etc/logingroup` up to date.

NETWORKING FEATURES

NFS

If `/etc/logingroup` is linked to `/etc/group`, `initgroups()` tries to use the Network Information Service (NIS) for entries beginning with a plus sign (+). If group membership for *name* is managed by NIS, and no NIS server is able to respond, a call to `initgroups()` does not return until a server does respond. This causes commands such as *login*(1) and *su*(1) to wait indefinitely.

See *group*(4) for proper syntax and operation.

AUTHOR

`initgroups()` was developed by the University of California, Berkeley.

FILES

`/etc/logingroup` login group file

SEE ALSO

login(1), su(1), getgroups(2), setgroups(2), group(4).

NAME

initopt() - initialize a NetIPC option buffer

SYNOPSIS

```
#include <sys/ns_ipc.h>

void initopt(short opt[], short maxoptions, short *result);
```

DESCRIPTION

`initopt()` must be used to intitialize a NetIPC option buffer. Options can be added to the buffer by calling `addopt()` and read by calling `readopt()` (see *addopt*(3N) and *readopt*(3N)).

The *maxoptions* parameter specifies the maximum number of options that can be placed in the buffer. For example, if *maxoptions* specifies one, then one option can be added to the buffer. If three is specified then three options can be added. Options are indexed starting from zero.

Each time a NetIPC options buffer is to be used, it should be initialized to the number of options to be added. If fewer options are added than the buffer is initialized for, a resulting unitialized option may cause an error. A given buffer can be reused, but should be reinitialized before each use.

A NetIPC option buffer consists of space for overhead and space for options. `optoverhead()` returns the number of bytes needed in a buffer for a given number of options (see *optoverhead*(3N)). The bytes needed for option data depends upon the number and type of the options to be added, and must be calculated by the programmer. An option buffer can be larger than necessary.

Parameters

opt	(input parameter) The address of the buffer to be initialized.
maxtoptions	(input parameter) The maximum number of options to be added to the buffer.
result	(output parameter) The result code returned. Refer to diagnostics section below for more information.

RETURN VALUE

None. Errors are returned in the *result* parameter.

DIAGNOSTICS

`initopt()` sets *result* to the value indicated when the following conditions are encountered:

[NSR_ADDR_OPT]	The options parameter is null.
[NSR_NO_ERROR]	The call was successful.
[NSR_OPT_TOTAL]	The *num_entries* parameter is negative.

AUTHOR

`initopt()` was developed by HP.

SEE ALSO

ipcconnect(2), ipccontrol(2), ipccreate(2), ipcdest(2), ipcgetnodename(2), ipclookup(2), ipcname(2), ipcnamerase(2), ipcrecv(2), ipcrecvcn(2), ipcselect(2), ipcsend(2), ipcsetnodename(2), ipcshutdown(2), addopt(3N), ipcerrmsg(3N), optoverhead(3N), readopt(2).

NAME

io_burst( ) - perform low-overhead I/O on an HP-IB/GPIO channel

SYNOPSIS

```
#include <dvio.h>

int io_burst(int eid, int flag);
```

DESCRIPTION

`io_burst()` is used to perform low-overhead burst transfers on the specified HP-IB, or GPIO interface. *eid* is the entity identifier for an open HP-IB or GPIO device file returned by a previous call to `open()`, `dup()`, `creat()`, or `fcntl()` with an `FDUPD` command option. *flag* is an integer which, if non-zero, enables burst mode or, if zero, disables it.

In burst mode, memory-mapped I/O address space assigned to the interface card select code is mapped directly into user space such that data can be transferred directly between user memory and the interface card, eliminating the need for kernel calls and the associated overhead. Burst mode affects only `read()`, `write()`, `gpio_get_status()`, `gpio_set_ctl()`, `hpib_io()`, and `hpib_send_cmd()` calls. All other operations are unaffected. When burst mode is enabled, the interface is locked so that no other process can access it until burst mode is disabled. When burst mode is disabled, the interface is reset (see *io_reset*(3I)).

RETURN VALUE

`io_burst()` returns zero if successful or −1 if an error is detected.

ERRORS

`io_burst()` fails under any of the following circumstances and sets `errno` (see *errno*(2)) to one of the following values:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to an HP-IB or GPIO device special file.
[EIO]	A timeout occurred during the call to `ioburst()` (Series 300/400 only).

Enabling burst mode locks the interface from all other processes, so it should never be used with any interface that supports a system disk or swap device.

Timeouts for `read()`, `write()`, `gpio_get_status()`, `gpio_set_ctl()`, `hpib_io()`, and `hpib_send_cmd()` do not work while in burst mode, but these commands can be interrupted by signals.

AUTHOR

`io_burst()` was developed by HP.

SEE ALSO

dup(2), creat(2), fcntl(2), open(2), read(2), write(2), gpio_get_status(3I), gpio_set_ctl(3I), hpib_io(3I), hpib_send_cmd(3I), io_reset(3I).

NAME

io_dma_ctl() - control DMA allocation for an interface

SYNOPSIS

```
#include <sys/dil.h>

int io_dma_ctl(int eid, int mode);
```

DESCRIPTION

`io_dma_ctl()` is used to control system DMA allocation for a specific interface. *eid* is the entity identifier for an open HP-IB, Centronics-compatible parallel, or GPIO device file returned by a previous call to `open()`, `dup()`, `creat()`, or `fcntl()` with an `FDUPD` command option.

The *mode* parameter describes what type of DMA allocation the system should use for the interface associated with *eid*. *mode* is determined by selecting one of flags from the following list in `<sys/dil.h>`:

One and only one of the following flags *must* be specified:

`DMA_ACTIVE`	Inform the DMA subsystem that this interface intends to use DMA and requires higher priority than slow devices. This is the level of DMA allocation used by CS/80, Amigo, and SCSI devices.
`DMA_UNACTIVE`	Remove the effect of a previous `DMA_ACTIVE`.
`DMA_RESERVE`	Guarantee that a DMA channel will remain unlocked for future requests for DMA by all devices on this interface.
`DMA_UNRESERVE`	Remove the effect of a previous `DMA_RESERVE`.
`DMA_LOCK`	Lock a DMA channel for exclusive use by all devices on this interface.
`DMA_UNLOCK`	Unlock a DMA channel locked by this interface.

RETURN VALUE

`io_dma_ctl()` returns 0 (zero) if successful, or –1 if an error was encountered.

ERRORS

`io_dma_ctl()` fails under the following circumstances, and sets `errno` (see *errno*(2)) to the value indicated:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to a Device I/O Library-compatible device file.
[EIO]	A timeout occurred (Series 300/400 only).
[EINTR]	Request was interrupted by a signal.
[EINVAL]	Interface was unable to reserve or lock a DMA channel.

WARNINGS

Series 300/400 systems have only two DMA channels. Use of `DMA_LOCK` could limit access to DMA resources by system disks, resulting in lower system performance.

AUTHOR

`io_dma_ctl()` was developed by HP.

SEE ALSO

dup(2), creat(2), fcntl(2), open(2).

NAME

io_eol_ctl() - set up read termination character on special file

SYNOPSIS

```
#include <dvio.h>

int io_eol_ctl(int eid, int flag, int match);
```

DESCRIPTION

`io_eol_ctl()` specifies a character to be used in terminating a read operation from the specified file (entity identifier).

eid is an entity identifier of an open HP-IB raw bus, Centronics-compatible parallel, or GPIO device file obtained from an `open()`, `dup()`, `fcntl()`, or `creat()` call. *flag* is an integer that enables or disables character-match termination. A non-zero value enables character-match termination, while a zero value disables it. *match* is an integer containing the numerical equivalent of the termination character. *match* is ignored if *flag* is zero. When in 8-bit mode, the lower 8 bits of *match* are used as the termination character. In 16-bit mode, the lower 16 bits are used.

Upon opening a file, the default condition is character-match termination disabled. When enabled, the character specified by *match* is checked for during read operations. The read is terminated upon receipt of this character, or upon any of the other termination conditions normally in effect for this file. Examples of other conditions are satisfying the specified byte count, and receiving a character when the EOI line is asserted (HP-IB). When the read is terminated by a *match* character, this character is the last character returned in the buffer.

Entity identifiers for the same device file obtained by separate `open()` calls have their own termination characters associated with them. Entity identifiers for the same device file inherited by a `fork()` call share the same termination character. In the latter case, if one process changes the termination character, the new termination character is in effect for all such entity identifiers.

RETURN VALUE

`io_eol_ctl()` returns 0 (zero) if successful, or –1 if an error was encountered.

ERRORS

`io_eol_ctl()` fails under the following circumstances, and sets `errno` (see *errno*(2)) to the value indicated:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to a channel device file.

AUTHOR

`io_eol_ctl()` was developed by HP.

SEE ALSO

dup(2), creat(2), fcntl(2), open(2), io_width_ctl(3I).

NAME

io_get_term_reason() - determine how last read terminated

SYNOPSIS

```
#include <dvio.h>

int io_get_term_reason(int eid);
```

DESCRIPTION

`io_get_term_reason()` returns the termination reason for the last read made on this entity id. *eid* is an entity identifier of an open HP-IB raw bus, Centronics-compatible parallel interface, or GPIO device file obtained from an `open()`, `dup()`, `fcntl()`, or `creat()` call.

All entity identifiers descending from an `open()` request (such as from `dup()` or `fork()`) set this status. For example, if the calling process had opened this entity identifier and later forked, the status returned would be from the last read done by either the calling process or its child.

RETURN VALUE

`io_get_term_reason()` returns a value indicating how the last read on the specified entity identifier was terminated. This value is interpreted as follows (note that combinations are possible):

Value	Description
−1	An error was encountered while making this function request.
0	Last read encountered some abnormal termination reason not covered by any of the other reasons.
1	Last read terminated by reading the number of bytes requested.
2	Last read terminated by detecting the specified termination character.
4	Last read terminated by detecting some device-imposed termination condition. Examples are: EOI for HP-IB, PSTS line on GPIO, or some other end-of-record condition, such as the physical end-of-record mark on a 9-track tape.

ERRORS

`io_get_term_reason()` fails under the following circumstances, and sets `errno` (see *errno*(2)) to the value indicated:

[EBADF] *eid* does not refer to an open file.

[ENOTTY] *eid* does not refer to a channel device file.

DEPENDENCIES

Series 300/400:

For the GPIO interface, PSTS is checked only at the beginning of a transfer. An interrupt caused by an EIR also terminates a transfer. The value of the termination reason in this case is also 4.

For the Centronics-compatible parallel interface, a termination reason value of 4 indicates that the transfer terminated because the peripheral asserted the ACK line.

AUTHOR

`io_get_term_reason()` was developed by HP.

SEE ALSO

dup(2), creat(2), fcntl(2), open(2), read(2), io_eol_ctl(3I).

NAME

io_interrupt_ctl() - enable/disable interrupts for the associated eid

SYNOPSIS

```
#include <dvio.h>

int io_interrupt_ctl(int eid, int enable_flag);
```

DESCRIPTION

eid is the entity identifier of an open HP-IB raw bus, Centronics-compatible parallel, or GPIO device file obtained from an `open()`, `dup()`, `fcntl()`, or `creat()` call. *flag* is an integer which enables or disables interrupts for the associated *eid*. A non-zero value enables interrupts.

Interrupts can be disabled or enabled as desired. When an interrupt occurs for a given *eid* the interrupts associated with this *eid* are automatically disabled from recurring. To re-enable interrupts for this *eid*, use `io_interrupt_ctl()`.

RETURN VALUE

`io_interrupt_ctl()` returns 0 (zero) if successful, or –1 if an error was encountered.

ERRORS

`io_interrupt_ctl()` fails under the following situations, and sets `errno` (see *errno*(2)) to the value indicated:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to a device that supports interrupts.
[EINVAL]	No interrupt conditions were specified for this *eid*.

AUTHOR

`io_interrupt_ctl()` was developed by HP.

SEE ALSO

dup(2), creat(2), fcntl(2), open(2), io_on_interrupt(3I).

NAME

io_lock, io_unlock - lock and unlock an interface

SYNOPSIS

```
#include <dvio.h>

int io_lock(int eid);

int io_unlock(int eid);
```

DESCRIPTION

`iolock()` attempts to lock the interface associated with an entity identifier for the requesting process. Locking an interface gives exclusive use of the interface associated with the *eid* to the requesting process, thus avoiding unintended interference from other processes during a series of separate I/O requests. All locks for a process are removed when the process closes the file or terminates.

eid is an entity identifier of an open HP-IB, Centronics-compatible parallel, or GPIO device file, obtained from an `open()`, `dup()`, `fcntl()`, or `creat()` call (see *open*(2), *dup*(2), *fcntl*(2), and *creat*(2)).

Other processes that attempt to access or lock a locked interface either return an error or sleep until the interface becomes unlocked. The action taken is determined by the current setting of the `O_NDELAY` flag (see *open*(2). If the `O_NDELAY` flag is set, accesses to a locked interface fail and set `errno` to indicate the error. If the `O_NDELAY` flag is not set, accesses to a locked interface block until the interface is unlocked, the current timeout expires, or the request is interrupted by a signal.

A lock is associated with a process, not with an *eid*. Locking an interface with a particular *eid* does not prevent the process that owns the lock from accessing the interface through another *eid*. A lock associated with an *eid* is not inherited by a child process during a `fork()` (see *fork*(2)).

Nested locking is fully supported. If a process owns a locked interface and calls a generic subroutine that does a lock and unlock, the calling process does not lose its lock on the interface. Locking requests produced by a given process for an interface already locked by the same process increment the current lock count for that interface.

`io_unlock()` allows a process to remove a lock from the interface associated with the *eid*. A locked interface can be unlocked only by the process that directly owns the lock. When an unlock operation is applied to an *eid* that is currently multiply locked, the unlock operation decrements the current lock counter for that interface, and the interface remains locked until the count is reduced to zero.

RETURN VALUE

`io_lock()` and `io_unlock()` return the integer value of the current lock count if successful. A lock count greater than zero indicates that the interface is still locked. A lock count of zero indicates that the interface is no longer locked. A -1 indicates that an error has occured.

ERRORS

`io_lock()` and `io_unlock()` fail in the following situations, and set `errno` (see *errno*(2)) to the value indicated:

[EACCES]	An attempt was made to lock an interface locked by another process with `O_NDELAY` set.
[EBADF]	*eid* does not refer to an open file.
[EINTR]	A signal was caught while attempting to perform the lock with `O_NDELAY` clear.
[EINVAL]	an attempt was made to unlock when the interface is not locked.
[ETIMEDOUT]	A timeout occured while attempting to perform the lock with O_NDELAY clear.
[ENOTTY]	*eid* does not refer to a channel device file.
[EPERM]	An attempt was made to unlock when lock is not owned by this user.

WARNINGS

`io_lock()` provides a mandatory lock enforced by the system, and should not be used with any interface supporting a system disk or swap device.

Series 800:

Processes that lock HP-IB or GPIO interfaces should clear all locks before exiting. The driver attempts to

clear them if the process terminates unexpectedly; however, a lock might be left outstanding if the locker dies after creating new file descriptors (via `fork()` or `dup()`) that refer to the same device file. Ensuring that all open file descriptors on a given interface are closed remedies the situation.

DEPENDENCIES

Series 300/400:

`io_lock()` and `io_unlock()` return [EIO] if a timeout occurs.

AUTHOR

`io_lock()` and `io_unlock()` were developed by HP.

SEE ALSO

io_timeout_ctl(3I), open(2).

NAME

io_on_interrupt() - device interrupt (fault) control

SYNOPSIS

```
#include <dvio.h>

int (*io_on_interrupt(
    int eid,
    struct interrupt_struct *causevec,
    int (*handler)(int, struct interrupt_struct *)
))(int, struct interrupt_struct *);
```

DESCRIPTION

eid is an entity identifier of an open HP-IB raw bus, Centronics-compatible parallel interface, or GPIO device file, obtained from an `open()`, `dup()`, `fcntl()`, or `creat()` call.

causevec is a pointer to a structure of the form:

```
struct interrupt_struct {
    integer     cause;
    integer     mask;
};
```

The `interrupt_struct` structure is defined in the file `dvio.h`.

cause is a bit vector specifying which of the interrupt or fault events can cause the handler routine to be invoked. The interrupt causes are often specific to the type of interface being considered. Also, certain exception (error) conditions can be handled using the `io_on_interrupt()` capability. Specifying a zero valued *cause* vector effectively turns off the interrupt for that *eid*.

The *mask* parameter is used when an HP-IB parallel poll interrupt is being defined. *mask* is an integer that specifies which parallel poll response lines are of interest. The value of *mask* is viewed as an 8-bit binary number where the least significant bit corresponds to line DIO1; the most significant bit to line DIO8. For example, to activate an interrupt handler when a response occurs on lines 2 or 6, the correct binary number is 00100010. Thus a hexadecimal value of 22 is the correct argument value for *mask*.

When an enabled interrupt condition on the specified *eid* occurs, the receiving process executes the interrupt-handler function pointed to by *handler*. The entity identifier *eid* and the interrupt condition *cause* are returned as the first and second parameters, respectively.

When an interrupt that is to be caught occurs during a `read()`, `write()`, `open()`, or `ioctl()` system call on a slow device such as a terminal (but not a file), during a `pause()` system call, a `sigpause()` system call, or a `wait()` system call that does not return immediately due to the existence of a previously stopped or zombie process, the interrupt handling function is executed and the interrupted system call returns –1 to the calling process with `errno` set to EINTR.

Interrupt *handlers* are not inherited across a `fork()`. *eids* for the same device file produced by `dup()` share the same *handler*.

An interrupt for a given *eid* is implicitly disabled after the occurrence of the event. The interrupt condition can be re-enabled by using `io_interrupt_ctl()` (see *io_interrupt_ctl*(3I)).

When an event specified by *cause* occurs, the receiving process executes the interrupt *handler* function pointed to by *handler*. When the *handler* returns, the user process resumes at the execution point where the event occurred.

Two parameters are passed to *handler*: the *eid* associated with the event, and a pointer to a `causevec` structure. The cause of the interrupt can be determined by the value returned in the *cause* field of the `causevec` structure (more than 1 bit can be set, indicating that more than 1 interrupting condition has occurred). If the interrupt *handler* was invoked due to a parallel poll interrupt, the *mask* field of the `causevec` structure contains the parallel poll response byte.

HP-IB Interrupts

This section describes interrupt causes specific to an HP-IB device. For an HP-IB device, the cause is a bit vector which is used as follows. To enable a given event, the appropriate bit (in *cause*), shown below, must be set to 1:

SRQ	SRQ and active controller
TLK	Talker addressed
LTN	Listener addressed
TCT	Controller in charge
IFC	IFC has been asserted
REN	Remote enable
DCL	Device clear
GET	Group execution trigger
PPOLL	Parallel poll

GPIO Interrupts

This section describes interrupt causes specific to a GPIO device. For a GPIO device, *cause* is a bit vector which is used as follows. To enable a given event, the appropriate bit (in *cause*), shown below, must be set to 1:

EIR	External interrupt
SIE0	Status line 0
SIE1	Status line 1

Parallel Interrupts

This section describes interrupt causes specific to a Centronics-compatible parallel device. For a Centronics-compatible parallel device, *cause* is a bit vector which is used as follows. To enable a given event, the appropriate bit (in *cause*), shown below, must be set to 1:

NERROR	Nerror interrupt
SELECT	Select interrupt
PE	Paper error interrupt

RETURN VALUE

`io_on_interrupt()` returns a pointer to the previous *handler* if the new *handler* is successfully installed; otherwise it returns a –1 and sets `errno` to indicate the error.

ERRORS

`io_on_interrupt()` fails for any of the following reasons and sets `errno` to the value indicated:

[EACCES]	The interface associated with this *eid* is locked by another process and `O_NDELAY` is set for this *eid* (see *iolock*(3I)).
[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to a GPIO, Centronics-compatible parallel, or a raw HP-IB device file.
[EFAULT]	*handler* points to an illegal address. The reliable detection of this error is implementation dependent.
[EFAULT]	*causevec* points to an illegal address. The reliable detection of this error is implementation dependent.

DEPENDENCIES

Series 300/400:

For the HP98622 GPIO interface, only the EIR interrupt is available. For the HP 98265A/B HP-IB interface, the IFC and GET interrupts are not provided.

Series 800:

For the HP 27114 AFI interface, only the EIR interrupt is available.

AUTHOR

`io_on_interrupt()` was developed by HP.

SEE ALSO

dup(2), creat(2), fcntl(2), open(2), pause(2), sigpause(2), io_interrupt_ctl(3I).

NAME

io_reset() - reset an I/O interface

SYNOPSIS

```
#include <dvio.h>

int io_reset(int eid);
```

DESCRIPTION

`io_reset()` resets the interface associated with the device file that was opened. It also pulses the peripheral reset line on the GPIO interface, or the IFC line on the HP-IB. *eid* is an entity identifier of an open HP-IB, Centronics-compatible parallel interface, or GPIO device file obtained from an `open()`, `dup()`, `fcntl()`, or `creat()` call.

`io_reset()` also causes an interface to go through its self-test, and returns a failure indication if the interface fails its test.

RETURN VALUE

`io_reset()` returns 0 (zero) if successful, or –1 if an error was encountered.

ERRORS

`io_reset()` fails under the following circumstances, and sets `errno` (see *errno*(2)) to the value indicated:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to a channel device file.
[EIO]	Interface could not be reset or failed self-test.
[EACCES]	The interface associated with this *eid* is locked by another process and `O_NDELAY` is set for this *eid* (see *io_lock*(3I)).

DEPENDENCIES

Series 300/400:

When an HP-IB interface is reset, the interrupt mask is set to 0, the parallel poll response is set to 0, the serial poll response is set to 0, the HP-IB address is assigned its powerup default value, the IFC line is pulsed (if system controller), the card is put on line, and REN is set (if system controller).

When a GPIO interface is reset, the peripheral reset line is pulled low, the PCTL line is placed in the clear state, and if the DOUT CLEAR jumper is installed, the data out lines are all cleared. The interrupt enable bit is also cleared.

Interface self-test is not supported.

AUTHOR

`io_reset()` was developed by HP.

NAME

io_speed_ctl() - inform system of required transfer speed

SYNOPSIS

```
#include <dvio.h>

int io_speed_ctl(int eid, int speed);
```

DESCRIPTION

`io_speed_ctl()` selects the data transfer speed for a data path used for a particular interface. The transfer method (i.e., DMA or fast-handshake) chosen by the system is determined by the speed requirements.

eid is an entity identifier of an open HP-IB raw bus, Centronics-compatible parallel, or GPIO device file obtained from an `open()`, `dup()`, `fcntl()`, or `creat()` call. *speed* is an integer specifying the data transfer speed in Kbytes per second (one Kbyte equals 1024 bytes).

RETURN VALUE

`io_speed_ctl()` returns 0 if successful, and –1 otherwise.

ERRORS

`io_speed_ctl()` fails under the following condition, and sets `errno` to the value indicated:

[ENOTTY] *eid* does not refer to channel device file.

[EBADF] *eid* does not refer to an open file.

DEPENDENCIES

Series 300/400:

For values of speed less than 7, the system uses an interrupt transfer. For larger values, DMA is used if available; otherwise, the system uses an interrupt transfer. The default transfer method is DMA.

Series 800:

DMA is the only supported transfer method.

AUTHOR

`io_speed_ctl()` was developed by HP.

NAME

io_timeout_ctl() - establish a time limit for I/O operations

SYNOPSIS

```
#include <dvio.h>

int io_timeout_ctl(int eid, long time);
```

DESCRIPTION

`io_timeout_ctl()` assigns a timeout value to the specified *eid* (entity identifier). *eid* is an entity identifier of an open HP-IB raw bus, auto-addressed, Centronics-compatible parallel, or GPIO device file obtained from an `open()`, `dup()`, `fcntl()`, or `creat()` call. *time* is a long integer value specifying the length of the timeout in microseconds. A value of 0 for *time* specifies no timeout (infinity).

This timeout applies to future read and write requests on this *eid*. If a read or write request does not complete within the specified time limit, the request is aborted and returns an error indication. If an operation is aborted due to a timeout, `errno` is set to ETIMEDOUT.

Although the timeout value is specified in microseconds, the resolution of the timeout is system-dependent. For example, a particular system might have a resolution of 10 milliseconds, in which case the specified timeout value is rounded up to the next 10 msec boundary. A timeout value of zero means that the system never causes a timeout. When a file is opened, a zero timeout value is assigned by default.

Entity identifiers for the same device file obtained by separate `open()` calls have their own timeout values associated with them. Entity identifiers for the same device file obtained by `dup()` or inherited by a `fork()` call share the same timeout value. In the latter case, if one process changes the timeout, the new timeout is in effect for all such *eid*s.

RETURN VALUE

`io_timeout_ctl()` returns 0 (zero) if successful, or –1 if an error was encountered.

ERRORS

`io_timeout_ctl()` fails under the following circumstances, and sets `errno` (see *errno*(2)) to the value indicated:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to a channel device file.

DEPENDENCIES

Series 300/400:

System timeout resolution is 20 msec.

EIO is returned if an operation is aborted due to a timeout.

AUTHOR

`io_timeout_ctl()` was developed by HP.

NAME

io_width_ctl - set width of data path

SYNOPSIS

```
#include <dvio.h>

int io_width_ctl(int eid, int width);
```

DESCRIPTION

`io_width_ctl()` enables you to select the width of the data path to be used for a particular interface. *eid* is an entity identifier of an open HP-IB, Centronics-compatible parallel interface, or GPIO device file obtained from an `open()`, `dup()`, `fcntl()`, or `creat()` call. *width* is an integer specifying the width of the data path in bits.

An error is given if an invalid width is specified. Specifying a width with this function sets the width for all users of the device file associated with the given entity id. When first opened, the default width is 8 bits.

For the GPIO interface only widths of 8 and 16 bits are currently supported. For the HP-IB and Centronics-compatible parallel interfaces, only a width of 8 bits is supported.

RETURN VALUE

`io_width_ctl()` returns 0 if successful, and –1 if an error was encountered.

ERRORS

`io_width_ctl()` fails under the following circumstances and sets `errno` (see *errno*(2)) to the value indicated:

[EBADF]	*eid* does not refer to an open file.
[ENOTTY]	*eid* does not refer to a channel device file.
[EINVAL]	the specified *width* is not supported on this device file.

AUTHOR

`io_width_ctl()` was developed by HP.

NAME

ipcerrmsg(), ipcerrstr() - provide text describing a NetIPC error number

SYNOPSIS

```
#include<sys/ns_ipc.h>

char *ipcerrstr(int error);

void ipcerrmsg(
    int error,
    char *buffer,
    int *len,
    int *result);
```

DESCRIPTION

`ipcerrstr()` Takes as input a NetIPC error number and returns a pointer to a NULL-terminated string describing the error. If the error is unknown, NULL is returned.

`ipcerrmsg()` Copies an error message for a NetIPC error into a supplied buffer. It copies *len−1* bytes into the buffer to ensure that the result is null-terminated.

`ipcerrmsg()` parameters are as follows:

error (input parameter) The NetIPC error number to be described.

buffer (input parameter) A data buffer into which the description is to be copied.

len (input/output parameter) A pointer to the length of the buffer. On output it contains the length of the description.

result (output parameter) The result code returned. Refer to ERRORS below for more information.

RETURN VALUE

`ipcerrstr()` returns NULL if the error number is unknown.

`ipcerrmsg()` returns results in the *result* parameter.

ERRORS

`ipcerrmsg()` sets *result* to the value indicated when any of these conditions are encountered:

[NSR_NO_ERROR] The call was successful.

[NSR_ERRNUM] An unknown error number was passed to `ipcerrmsg`.

AUTHOR

`ipcerrmsg()` was developed by HP.

SEE ALSO

ipcconnect(2), ipccontrol(2), ipccreate(2), ipcdest(2), ipcgetnodename(2), ipclookup(2), ipcname(2), ipcnamerase(2), ipcrecv(2), ipcrecvcn(2), ipcselect(2), ipcsend(2), ipcsetnodename(2), ipcshutdown(2), addopt(3N), initopt(3N), optoverhead(3N), readopt(3N).

NAME

is_68010_present(), is_68881_present(), is_98635A_present(), is_98248A_present() - check for presence of hardware capabilities

SYNOPSIS

```
#include <unistd.h> int is_68010_present(void);

int is_68881_present(void);

int is_98635A_present(void);

int is_98248A_present(void);
```

DESCRIPTION

Each function checks for the presence of a specified hardware capability, returning `1` if it exists or `0` if it does not.

RETURN VALUE

The value `1` is returned by:

`is_68010_present()`	if the system has an MC68010 as its CPU.
`is_68881_present()`	if an MC68881 floating-point coprocessor is present.
`is_98635A_present()`	if an HP98635A floating-point accelerator has been installed.
`is_98248A_present()`	if an HP98248A floating-point accelerator has been installed.

AUTHOR

`is_hw_present()` was developed by HP.

NAME

isinf(), isinff() - test for INFINITY functions

SYNOPSIS

```
#include <math.h>

int isinf(double x);

int isinff(float x);
```

DESCRIPTION

`isinf()` returns a positive integer if x is +INFINITY, or a negative integer if x is –INFINITY. Otherwise it returns zero.

`isinff()` is the `float` version of `isinf()`. It is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard. Programs must be compiled in ANSI mode (use the `-Aa` option) in order to use this function; otherwise, the compiler promotes the `float` argument to `double`, and the function returns incorrect results.

DEPENDENCIES

Series 300/400

`isinff()` is not supported on Series 300/400 systems.

Series 700/800

`isinff()` is provided in the PA1.1 versions of the math library only. The `+DA1.1` option (the default on Series 700 systems) links in a PA1.1 version automatically. A PA1.1 library can be linked in explicitly. For more information, see the *HP-UX Floating-Point Guide*.

SEE ALSO

isnan(3M), fpclassify(3M), ieee(3M).

NAME

isnan( ), isnanf( ) - test for NaN functions

SYNOPSIS

```
#include <math.h>

int isnan(double x);

int isnanf(float x);
```

DESCRIPTION

`isnan( )` returns a nonzero integer if *x* is NaN (not-a-number). Otherwise it returns zero.

`isnanf( )` is the `float` version of `isnan( )`. Programs must be compiled in ANSI mode (use the `-Aa` option) in order to use this function; otherwise, the compiler promotes the `float` argument to `double`, and the function returns incorrect results.

DEPENDENCIES

Series 300/400

`isnanf( )` is not supported on Series 300/400 systems.

Series 700/800

`isnanf( )` is not specified by any standard; however, it is named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard. It is provided in the PA1.1 versions of the math library only. The `+DA1.1` option (the default on Series 700 systems) links in a PA1.1 version automatically. A PA1.1 library can be linked in explicitly. For more information, see the *HP-UX Floating-Point Guide*.

SEE ALSO

isinf(3M), fpclassify(3M), ieee(3M).

STANDARDS CONFORMANCE

`isnan( )` in libm.a: AES, XPG3
`isnan( )` in libM.a: AES, XPG3, XPG4

NAME

l3tol(), ltol3() - convert between 3-byte integers and long integers

SYNOPSIS

```
#include <stdlib.h>

void l3tol(long int *lp, const char *cp, int n);

void ltol3(char *cp, const long int *lp, int n);
```

DESCRIPTION

`l3tol()` Convert a list of *n* three-byte integers packed into a character string pointed to by *cp* into a list of long integers pointed to by *lp*.

`ltol3()` Perform the reverse conversion from long integers (*lp*) to three-byte integers (*cp*).

These functions are useful for file-system maintenance where the block numbers are three bytes long.

SEE ALSO

fs(4).

WARNINGS

Because of possible differences in byte ordering, the numerical values of the long integers are machine-dependent.

STANDARDS CONFORMANCE

`l3tol()`: XPG2

`ltol3()`: XPG2

NAME

langinfo(), langtoid(), idtolang(), currlangid() - NLS information about native languages

SYNOPSIS

```
#include <langinfo.h>

char *langinfo(int langid, nl_item item);

int langtoid(const char *langname);

char *idtolang(int langid);

int currlangid(void);
```

DESCRIPTION

Note: All functions defined on this page are obsolete. Use of *nl_langinfo*(3C) is recommended as a replacement for `langinfo()`.

`langinfo()` returns a pointer to a null-terminated string containing information relevant to a particular language or cultural area defined in the program's locale (see *setlocale*(3C)). `langinfo()` effectively calls `langinit()` (see *nl_init*(3C)) to load the program's locale according to the language specified by *langid*. If *langid* or *item* (or both) is bad, `langinfo()` returns a pointer to a NULL string.

`currlangid()` looks for a **LANG** string in the user's environment. If it finds one, `currlangid()` returns the corresponding integer listed in *lang*(5). Otherwise, it returns 0 to indicate a default to native-computer, the method used before NLS was available.

`idtolang()` takes the integer *langid* and attempts to return the corresponding character string defined in *lang*(5). If *langid* is not found, an empty string is returned.

`langtoid()` is the inverse of `idtolang()`: it attempts to convert a string to a language ID, returning 0 to indicate native-computer if no match is found.

EXTERNAL INFLUENCES

Locale

The string returned by `langinfo()` for a particular *item* is determined by the locale category specified for that item in *langinfo*(5).

International Code Set Support

Single- and multi-byte character code sets are supported.

WARNINGS

`langinfo()` returns a pointer to a static area that is overwritten on each call.

AUTHOR

`langinfo()` was developed by HP.

SEE ALSO

nl_init(3C), nl_langinfo(3C), setlocale(3C), hpnls(5), lang(5), langinfo(5).

NAME

_ldecvt(), _ldfcvt(), _ldgcvt() - convert long-double floating-point number to string

SYNOPSIS

```
#include <stdlib.h>

char *_ldecvt(long_double value, size_t ndigit, int *decpt, int *sign);

char *_ldfcvt(long_double value, size_t ndigit, int *decpt, int *sign);

char *_ldgcvt(long_double value, size_t ndigit, char *buf);
```

DESCRIPTION

`_ldecvt()` converts *value* to a null-terminated string of *ndigit* digits and returns a pointer to the string. The high-order digit is non-zero, unless the value is zero. The low-order digit is rounded. The position of the radix character relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). The radix character is not included in the returned string. If the sign of the result is negative, the word pointed to by *sign* is non-zero; otherwise it is zero.

`_ldfcvt()` is identical to `_ldecvt()`, except that the correct digit has been rounded for printf `%Lf` (FORTRAN F-format) output of the number of digits specified by *ndigit*.

`_ldgcvt()` Convert the *value* to a null-terminated string in the array pointed to by *buf* and return *buf*. It produces *ndigit* significant digits in FORTRAN F-format if possible, or E-format otherwise. A minus sign, if required, and a radix character are included in the returned string. Trailing zeros are suppressed. The radix character is determined by the currently loaded NLS environment (see *setlocale*(3C)). If `setlocale()` has not been called successfully, the default NLS environment, "C" is used (see *lang*(5)). The default environment specifies a period (`.`) as the radix character.

RETURN VALUE

NaN is returned for Not-a-Number, and ±INFINITY is returned for Infinity.

WARNINGS

The values returned by `_ldecvt()` and `_ldfcvt()` point to a single static-data array whose content is overwritten by each call.

AUTHOR

`_ldecvt()`, `_ldfcvt()`, and `_ldgcvt()` were developed by HP.

SEE ALSO

setlocale(3C), printf(3S), hpnls(5), lang(5).

EXTERNAL INFLUENCES

Locale

The `LC_NUMERIC` category determines the radix character.

International Code Set Support

Single-byte character code sets are supported.

NAME

localeconv() - query the numeric formatting conventions of the current locale

SYNOPSIS

```
#include <locale.h>

struct lconv *localeconv(void);
```

DESCRIPTION

`localeconv()` sets the components of an object of type `struct lconv` (defined in <locale.h>) with values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the program's current locale (see *setlocale*(3C)).

The members of the structure with type `char *` are strings, any of which (except decimal_point) can point to `""` (the empty string) to indicate that the value is not available in the current locale or is of zero length. The members with type `char` are non-negative numbers, any of which can be `CHAR_MAX` (defined in <limits.h>) to indicate that the value is not available in the current locale. The members include the following:

`char *decimal_point`
: The decimal point character used to format non-monetary quantities. This is the same value as that returned by a call to `nl_langinfo()` with `RADIXCHAR` as its argument (see *nl_langinfo*(3C)).

`char *thousands_sep`
: The character used to separate groups of digits to the left of the decimal point character in formatted non-monetary quantities. This is the same value as that returned by a call to `nl_langinfo()` with `THOUSEP` as its argument (see *nl_langinfo*(3C)).

`char *grouping`
: A string where the numeric value of each byte indicates the size of each group of digits in formatted non-monetary quantities.

`char *int_curr_symbol`
: The international currency symbol applicable to the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in *ISO 4217 Codes for the Representation of Currency and Funds*. The fourth character (immediately preceding the null character) is the character used to separate the international currency symbol from the monetary quantity.

`char *currency_symbol`
: The local currency symbol applicable to the current locale. This value along with positioning information is returned by a call to `nl_langinfo()` with `CRNCYSTR` as its argument (see *nl_langinfo*(3C)).

`char *mon_decimal_point`
: The decimal point used to format monetary quantities.

`char *mon_thousands_sep`
: The separator for groups of digits to the left of the decimal point in formatted monetary quantities.

`char *mon_grouping`
: A string where the numeric value of each byte indicates the size of each group of digits in formatted monetary quantities.

`char *positive_sign`
: The string used to indicate a non-negative-valued formatted monetary quantity.

`char *negative_sign`
: The string used to indicate a negative-valued formatted monetary quantity.

`char int_frac_digits`
: The number of fractional digits (those to the right of the decimal point) to be displayed in an internationally formatted monetary quantity.

`char frac_digits`
The number of fractional digits (those to the right of the decimal point) to be displayed in a locally formatted monetary quantity.

`char p_cs_precedes`
Set to 1 or 0 if the `currency_symbol` respectively precedes or succeeds the value for a non-negative formatted monetary quantity.

`char p_sep_by_space`
Set to 1 or 0 if the `currency_symbol` respectively is or is not separated by a space from the value for a non-negative formatted monetary quantity.

`char n_cs_precedes`
Set to 1 or 0 if the `currency_symbol` respectively precedes or succeeds the value for a negative formatted monetary quantity.

`char n_sep_by_space`
Set to 1 or 0 if the `currency_symbol` respectively is or is not separated by a space from the value for a negative formatted monetary quantity.

`char p_sign_posn`
Set to a value indicating the positioning of the `positive_sign` for a non-negative formatted monetary quantity.

`char n_sign_posn`
Set to a value indicating the positioning of the `negative_sign` for a negative formatted monetary quantity.

The numeric value of each byte of `grouping` and `mon_grouping` is interpreted according to the following:

`CHAR_MAX`	No further grouping is to be performed.
`0`	The previous byte is to be repeatedly used for the remainder of the digits.
other	The value is the number of digits that comprise the current group. The next byte is examined to determine the size of the next group of digits to the left of the current group.

The value of `p_sign_posn` and `n_sign_posn` is interpreted according to the following:

`0`	Parentheses surround the quantity and `currency_symbol`.
`1`	The sign string precedes the quantity and `currency_symbol`.
`2`	The sign string succeeds the quantity and `currency_symbol`.
`3`	The sign string immediately precedes the `currency_symbol`.
`4`	The sign string immediately succeeds the `currency_symbol`.

`localeconv()` behaves as if no library function calls `localeconv()`.

EXTERNAL INFLUENCES

Locale

The `LC_NUMERIC` category influences the `decimal_point`, `thousands_sep`, and `grouping` members of the structure referenced by the pointer returned from a call to `localeconv()`.

The `LC_MONETARY` category influences all of the other members of this structure.

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUE

`localeconv()` returns a pointer to the filled-in `struct lconv`.

EXAMPLES

The following table illustrates the formatting used in five languages for monetary quantities.

Country	Positive format	Negative format	International format
american	$1,234.56	-$1,234.56	USD 1,234.56
italian	L.1.234	-L.1.234	ITL.1.234
dutch	F 1.234,56	F -1.234,56	NLG 1.234,56
norwegian	kr1.234,56	kr1.234,56-	NOK 1.234,56
portuguese	1,234$56	-1,234$56	PTE 1,234$56

For these five languages, the respective values for the monetary members of the structure returned by `localeconv()` are:

	american	italian	dutch	norwegian	portuguese
int_curr_symbol	USD	ITL.	NLG	NOK	PTE
currency_symbol	$	L.	F	kr	$
mon_decimal_point	.	""	,	,	$
mon_thousands_sep	,	.	.	.	,
mon_grouping	\3	\3	\3	\3	\3
positive_sign	""	""	""	""	""
negative_sign	-	-	-	-	-
int_frac_digits	2	0	2	2	2
frac_digits	2	0	2	2	2
p_cs_precedes	1	1	1	1	0
p_sep_by_space	0	0	1	0	0
n_cs_precedes	1	1	1	1	0
n_sep_by_space	0	0	1	0	0
p_sign_posn	1	1	1	1	1
n_sign_posn	4	1	4	2	1

WARNINGS

The structure returned by `localeconv()` should not be modified by the calling program. Calls to `setlocale()` with categories `LC_ALL`, `LC_MONETARY`, or `LC_NUMERIC` can overwrite the contents of the structure that `localeconv()` points to when it returns (see *setlocale*(3C)).

AUTHOR

`localeconv()` was developed by HP.

SEE ALSO

buildlang(1M), langinfo(3C), nl_langinfo(3C), setlocale(3C), hpnls(5).

STANDARDS CONFORMANCE

`localeconv()`: AES, XPG4, ANSI C

NAME

logname() - return login name of user

SYNOPSIS

```
#include <unistd.h>

char *logname(void);
```

DESCRIPTION

`logname()` returns a pointer to the null-terminated login name; it extracts the **$LOGNAME** variable from the user's environment.

WARNINGS

`logname()` returns a pointer to static data that is overwritten by each subsequent call.

This method of determining a login name is subject to forgery.

FILES

/etc/profile

SEE ALSO

env(1), login(1), profile(4), environ(5).

STANDARDS CONFORMANCE

`logname()`: SVID2, XPG2

NAME

lsearch(), lfind() - linear search and update

SYNOPSIS

```
#include <search.h>

void *lsearch(
    const void *key,
    void *base,
    size_t *nelp,
    size_t width,
    int (*compar)(const void *, const void *)
);

void *lfind(
    const void *key,
    const void *base,
    size_t *nelp,
    size_t width,
    int (*compar)(const void *, const void *)
);
```

DESCRIPTION

`lsearch()` is a linear search routine generalized from Knuth (6.1) Algorithm S. It returns a pointer into a table indicating where a datum may be found. If the datum does not occur, it is added at the end of the table.

key	Points to the datum to be sought in the table.
base	Points to the first element in the table.
nelp	Points to an integer containing the current number of elements in the table. The integer is incremented if the datum is added to the table.
compar	Name of the comparison function which the user must supply (`strcmp()`, for example). It is called with two arguments that point to the elements being compared. The function must return zero if the elements are equal and non-zero otherwise.

`lfind()`
Same as `lsearch()` except that if the datum is not found, it is not added to the table. Instead, a NULL pointer is returned.

Notes

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

EXAMPLES

This code fragment reads in ≤ `TABSIZE` strings of length ≤ `ELSIZE` and stores them in a table, eliminating duplicates.

```
#include <stdio.h>

#define TABSIZE 50
#define ELSIZE 120

    char line[ELSIZE], tab[TABSIZE][ELSIZE], *lsearch( );
    unsigned nel = 0;
    int strcmp( );
        ...
```

```
while (fgets(line, ELSIZE, stdin) != NULL &&
   nel < TABSIZE)
        (void) lsearch(line, (char *)tab, &nel,
                  ELSIZE, strcmp);
...
```

SEE ALSO

bsearch(3C), hsearch(3C), tsearch(3C).

RETURN VALUE

If the searched-for datum is found, both `lsearch()` and `lfind()` return a pointer to it. Otherwise, `lfind()` returns NULL and `lsearch()` returns a pointer to the newly added element.

WARNINGS

Undefined results can occur if there is not enough room in the table to add a new item.

STANDARDS CONFORMANCE

`lsearch()`: AES, SVID2, XPG2, XPG3, XPG4

`lfind()`: AES, SVID2, XPG2, XPG3, XPG4

NAME

ltostr(), ultostr(), ltoa(), ultoa() - convert long integers to strings

SYNOPSIS

```
#include <stdlib.h>

char *ltostr(long n, int base);

char *ultostr(unsigned long n, int base);

char *ltoa(long n);

char *ultoa(unsigned long n);
```

DESCRIPTION

`ltostr()` Convert a signed long integer to the corresponding string representation in the specified base. The argument *base* must be between 2 and 36, inclusive.

`ultostr()` Convert an unsigned long integer to the corresponding string representation in the specified base. The argument *base* must be between 2 and 36, inclusive.

`ltoa()` Convert a signed long integer to the corresponding base 10 string representation, returning a pointer to the result.

`ultoa()` Convert an unsigned long integer to the corresponding base 10 string representation, returning a pointer to the result.

These functions are smaller and faster than using `sprintf()` for simple conversions (see *sprintf*(3C)).

ERRORS

If the value of *base* is not between 2 and 36, `ltostr()` and `ultostr()` return NULL and set the external variable `errno` to ERANGE.

WARNINGS

The return values point to static data whose content is overwritten by each call.

AUTHOR

`ltostr()`, `ultostr()`, `ltoa()`, and `ultoa()` were developed by HP.

SEE ALSO

printf(3C), strtol(3C).

NAME

malloc(), free(), realloc(), calloc(), mallopt(), mallinfo(), memorymap() - main memory allocator

SYNOPSIS

```
#include <stdlib.h>

void *malloc(size_t size);

void *calloc(size_t nelem, size_t elsize);

void *realloc(void *ptr, size_t size);

void free(void *ptr);

void memorymap(int show_stats);
```

SYSTEM V SYNOPSIS

```
#include <malloc.h>

char *malloc(unsigned size);

void free(char *ptr);

char *realloc(char *ptr, unsigned size);

char *calloc(unsigned nelem, unsigned elsize);

int mallopt(int cmd, int value);

struct mallinfo mallinfo(void);
```

Remarks

The functionality in the old *malloc*(3X) package has been incorporated into *malloc*(3C). The library (`/usr/lib/libmalloc.a`) corresponding to the `-lmalloc` linker option is now an empty library. Makefiles that reference this library will continue to work. Applications that used the *malloc*(3X) package should still work properly with the new *malloc*(3C) package. If the old versions must be used, they are provided in files `/usr/old/libmalloc3x.a` and `/usr/old/libmalloc3c.o` for Release 8.07 only.

DESCRIPTION

The functions described in this manual entry provide a simple, general-purpose memory allocation package:

`malloc()` allocates space for a block of at least *size* bytes, but does not initialize the space.

`calloc()` allocates space for an array of *nelem* elements, each of size *elsize* bytes, and initializes the space to zeros.

`realloc()` changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. Existing contents are unchanged up to the lesser of the new and old sizes. If *ptr* is a NULL pointer, `realloc()` behaves like `malloc()` for the specified size. If *size* is zero and *ptr* is not a NULL pointer, the object it points to is freed and NULL is returned.

`free()` deallocates the space pointed to by *ptr* (a pointer to a block previously allocated by `malloc()`, `realloc()`, or `calloc()`) and makes the space available for further allocation. If *ptr* is a NULL pointer, no action occurs.

`mallopt()` provides for control over the allocation algorithm and other options in the *malloc*(3C) package. The available values for *cmd* are:

`M_MXFAST` Set *maxfast* to *value*. The algorithm allocates all blocks below the size of *maxfast* in large groups, then doles them out very quickly. The default value for *maxfast* is zero (`0`).

`M_NLBLKS` Set *numlblks* to *value*. The above mentioned "large groups" each contain *numlblks* blocks. *numlblks* must be greater than 1. The default value for *numlblks* is `100`.

`M_GRAIN` Set *grain* to *value*. The sizes of all blocks smaller than *maxfast* are considered to be rounded up to the nearest multiple of *grain*. *grain* must be greater than zero. The default value of *grain* is the smallest number of bytes that can accommodate

alignment of any data type. *value* is rounded up to a multiple of the default when *grain* is set.

M_KEEP Preserve data in a freed block until the next `malloc()`, `realloc()`, or `calloc()`. This option is provided only for compatibility with the old version of `malloc()` and is not recommended.

M_BLOCK Block all blockable signals in `malloc()`, `realloc()`, `calloc()`, and `free()`. This option is provided for those who need to write signal handlers that allocate memory. When set, the *malloc*(3C) routines can be called from within signal handlers (they become re-entrant). Default action is *not* to block all blockable signals.

M_UBLOCK Do not block all blockable signals in `malloc()`, `realloc()`, `calloc()`, and `free()`. This option cancels signal blocking initiated by the **M_BLOCK** option.

These values are defined in the <`malloc.h`> header file.

`mallopt()` can be called repeatedly, but must not be called after the first small block is allocated (unless *cmd* is set to **M_BLOCK** or **M_UBLOCK**).

`mallinfo()`
provides instrumentation describing space usage, but cannot be called until the first small block is allocated. It returns the structure:

```
struct mallinfo  {
    int arena;          /* total space in arena */
    int ordblks;        /* number of ordinary blocks */
    int smblks;         /* number of small blocks */
    int hblkhd;         /* space in holding block headers */
    int hblks;          /* number of holding blocks */
    int usmblks;        /* space in small blocks in use */
    int fsmblks;        /* space in free small blocks */
    int uordblks;       /* space in ordinary blocks in use */
    int fordblks;       /* space in free ordinary blocks */
    int keepcost;       /* space penalty if keep option is used */
}
```

This structure is defined in the <`malloc.h`> header file.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

`memorymap()`
can be used to display the contents of the memory allocator. A list of addresses and block descriptions is written (using `printf()`) to standard output. If the value of the *show_stats* parameter is 1, statistics concerning number of blocks and sizes used will also be written. If the value is zero, only the memory map will be written.

The addresses and sizes displayed by *memorymap* may not correspond to those requested by an application. The size of a block (as viewed by the allocator) includes header information and padding to properly align the block. The address is also offset by a certain amount to accomodate the header information.

RETURN VALUE

Upon successful completion, `malloc()`, `realloc()`, and `calloc()` return a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object. Otherwise, they return a NULL pointer. If `realloc()` returns a NULL pointer, the memory pointed to by the original pointer is left intact.

`mallopt()` returns zero for success and non-zero for failure.

ERRORS

[ENOMEM] `malloc()`, `realloc()`, and `calloc()` set `errno` to ENOMEM and return a NULL pointer when an out-of-memory condition arises.

[EINVAL] `malloc()`, `realloc()`, and `calloc()` set `errno` to EINVAL and return a NULL pointer when the memory being managed by `malloc()` has been detectably corrupted.

DIAGNOSTICS

`malloc()`, `realloc()`, and `calloc()` return a NULL pointer if there is no available memory, or if the memory managed by `malloc()` has been detectably corrupted. This memory may become corrupted if data is stored outside the bounds of a block, or if an invalid pointer (a pointer not generated by `malloc()`, `realloc()`, or *calloc()*) is passed as an argument to `free()` or `realloc()`.

If `mallopt()` is called after any allocation of a small block and *cmd* is not set to `M_BLOCK` or `M_UBLOCK` or if *cmd* or *value* is invalid, non-zero is returned. Otherwise, it returns zero.

WARNINGS

`malloc` functions use `brk()` and `sbrk()` (see *brk*(2)) to increase the address space of a process. Therefore, an application program that uses `brk()` or `sbrk()` must not use them to decrease the address space, because this confuses the `malloc` functions.

`free()` and `realloc()` do not check their pointer argument for validity.

If `free()` or `realloc()` is passed a pointer that was not the result of a call to `malloc()`, `realloc()`, or `calloc()`, or if space assigned by an allocation function is overrun, loss of data, a memory fault, bus error, or an infinite loop may occur at that time or during any subsequent call to `malloc()`, `realloc()`, `calloc()`, or `free()`.

The following actions are not supported and cause undesirable effects:

- Attempting to `free()` or `realloc()` a pointer not generated as the result of a call to `malloc()`, `realloc()`, or `calloc()`.

The following actions are strongly discouraged and may be unsupported in a future implementation of *malloc*(3C):

- Attempting to `free()` the same block twice.
- Depending on unmodified contents of a block after it has been freed.

Undocumented features of earlier memory allocators have not been duplicated.

COMPATIBILITY

The only external difference between the old *malloc*(3X) allocator and the *malloc*(3C) allocator is that the old allocator would return a NULL pointer for a request of zero bytes. The *malloc*(3C) allocator returns a valid memory address. This is not a concern for most applications.

Although the current implementation of *malloc*(3C) allows for freeing a block twice and does not corrupt the contents of a block after it is freed (until the next call to `realloc()`, `calloc()`, or `malloc()`), support for these features may be discontinued in a future implementation of *malloc*(3C) and should not be used.

SEE ALSO

brk(2), errno(2).

STANDARDS CONFORMANCE

`malloc()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
`calloc()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`free()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
`mallinfo()`: SVID2, XPG2

`mallopt()`: SVID2, XPG2
`realloc()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

matherr() - error-handling function

SYNOPSIS

```
#include <math.h>

int matherr(struct exception *x)
{
    /* your math error handling here */
}
```

DESCRIPTION

`matherr()` is invoked by functions in the Math Library when errors are detected. Programmers can define their own procedures for handling errors by including a function named `matherr()` in their programs. `matherr()` must be of the form described above. When an error occurs, a pointer to the exception structure *x* is passed to the user-supplied `matherr()` function. This structure, which is defined in the <`math.h`> header file, is as follows:

```
struct exception {
     int type;
     char *name;
     double arg1, arg2, retval;
};
```

The element `type` is an integer describing the type of error that has occurred, from the following list of constants (defined in the header file):

`DOMAIN`	argument domain error
`SING`	argument singularity
`OVERFLOW`	overflow range error
`UNDERFLOW`	underflow range error
`TLOSS`	total loss of significance
`PLOSS`	partial loss of significance

The element `name` points to a string containing the name of the function that incurred the error. The variables `arg1` and `arg2` are the arguments with which the function was invoked. `retval` is set to the default value that will be returned by the function unless the user's `matherr()` sets it to a different value. If there is only one argument, `arg1` is set to it, and `arg2` is undefined.

If the user's `matherr()` function returns non-zero, no error message is printed, and `errno` is not set.

If `matherr()` is not supplied by the user, the default error-handling procedures (described with the math functions involved) are invoked upon error. These procedures are also summarized in the table below. In every case, `errno` is set to EDOM or ERANGE and the program continues.

When `matherr()` is called from a `float` type math function (for example, `expf()` or `logf()`), the argument(s) and default return value (`arg1`, `arg2`, and `retval`) are converted to `double`. If an argument is a NaN, it is converted to a `double` NaN, without trapping, even if it is a signaling NaN. If a user-supplied `matherr()` function modifies `retval`, the value is converted to `float` when `matherr()` returns. If that conversion fails, then a signal is generated. Therefore, it is the responsibility of the user-supplied `matherr()` to select values for `retval` that can be successfully converted to `float`.

DEPENDENCIES

/lib/libM.a

In `/lib/libM.a`, `matherr()` has been renamed to `_matherr()` and no error messages are printed to the standard error output. `_matherr()` is provided in `/lib/libM.a` in order to assist in migrating programs from `libm.a` to `libM.a` and is *not* a part of XPG3, ANSI C, or POSIX.

EXAMPLES

```
#include <math.h>

int
matherr(x)
register struct exception *x;
{
   switch (x->type) {
```

```
        case DOMAIN:
            /* change sqrt to return sqrt(-arg1), not 0 */
            if (!strcmp(x->name, "sqrt")) {
                x->retval = sqrt(-x->arg1);
                return (0); /* print message and set errno */
            }
            else if (!strcmp(x->name, "sqrtf")) {
                x->retval = sqrtf(-x->arg1);
                return (0); /* print message and set errno */
            }
        case SING:
            /* all other domain or sing errors, print message and abort */
            fprintf(stderr, "domain error in %s\n", x->name);
            abort( );
        case PLOSS:
            /* print detailed error message */
            fprintf(stderr, "loss of significance in %s(%g) = %g\n",
                x->name, x->arg1, x->retval);
            return (1); /* take no other action */
        }
        return (0); /* all other errors, execute default procedure */
    }
```

DEFAULTS

DEFAULT ERROR HANDLING PROCEDURES

	Types of Errors					
type	DOMAIN	SING	OVERFLOW	UNDERFLOW	TLOSS	PLOSS
errno	EDOM	EDOM	ERANGE	ERANGE	ERANGE	ERANGE
BESSEL:	-	-	-	-	M, 0	*
y0, y1, yn (arg <= 0)	M, -H	-	-	-	-	-
EXP:	-	-	H	0	-	-
LOG, LOG10:						
(arg < 0)	M, -H	-	-	-	-	-
(arg = 0)	-	M, -H	-	-	-	-
POW:						
neg ** non-int	-	-	±H	0	-	-
0 ** non-pos	M, 0	-	-	-	-	-
SQRT:	M, 0	-	-	-	-	-
GAMMA:	-	M, H	H	-	-	-
HYPOT:	-	-	H	-	-	-
SINH:	-	-	±H	-	-	-
COSH:	-	-	H	-	-	-
SIN, COS, TAN:	-	-	-	-	M, 0	*
ASIN, ACOS, ATAN2:	M, 0	-	-	-	-	-

ABBREVIATIONS

*	As much as possible of the value is returned.
M	Message is printed (EDOM error) (except for Series 700/800 libM.a).
H	HUGE is returned.
-H	-HUGE is returned.
±H	HUGE or -HUGE is returned.
0	0 is returned.

STANDARDS CONFORMANCE

`matherr()` in libm.a: SVID2, XPG2, XPG3
`matherr()` in libM.a: XPG3

NAME

memccpy(), memchr(), memcmp(), memcpy(), memmove(), memset(), bcopy(), bcmp(), bzero(), ffs() - memory operations

SYNOPSIS

```
#include <string.h>

void *memccpy(void *s1, const void *s2, int c, size_t 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 *s1, const void *s2, size_t n);

void *memmove(void *s1, const void *s2, size_t n);

void *memset(void *s, int c, size_t n);

#include <strings.h>

int bcmp(const char *s1, const char *s2, int n);

void bcopy(const char *s1, char *s2, int n);

void bzero(char *s, int n);

int ffs(int i);
```

Remarks:

`bcmp()`, `bcopy()`, `bzero()`, `ffs()`, and `<strings.h>` are provided solely for portability of BSD applications, and are not recommended for new applications where portability is important. For portable applications, use `memcmp()`, `memmove()`, and `memset()`, respectively. `ffs()` has no portable equivalent.

DESCRIPTION

These functions operate as efficiently as possible on memory areas (arrays of characters bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

Definitions for all these functions, the type `size_t`, and the constant `NULL` are provided in the `<string.h>` header file.

`memccpy()` Copy characters from the object pointed to by *s2* into the object pointed to by *s1*, stopping after the first occurrence of character *c* has been copied, or after *n* characters have been copied, whichever comes first. If copying takes place between objects that overlap, the behavior is undefined. `memccpy()` returns a pointer to the character after the copy of *c* in *s1*, or a NULL pointer if *c* was not found in the first *n* characters of *s2*.

`memchr()` Locate the first occurrence of *c* (converted to an `unsigned char`) in the initial *n* characters (each interpreted as `unsigned char`) of the object pointed to by *s*. `memchr()` returns a pointer to the located character, or a NULL pointer if the character does not occur in the object.

`memcmp()` Compare the first *n* characters of the object pointed to by *s1* to the first *n* characters of the object pointed to by *s2*. `memcmp()` returns an integer greater than, equal to, or less than zero, according to whether the object pointed to by *s1* is greater than, equal to, or less than the object pointed to by *s2*. The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of characters (both interpreted as `unsigned char`) that differ in the objects being compared.

`memcpy()` Copy *n* characters from the object pointed to by *s2* into the object pointed to by *s1*. If copying takes place between objects that overlap, the behavior is undefined. `memcpy()` returns the value of *s1*.

`memmove()` Copy *n* characters from the object pointed to by *s2* into the object pointed to by *s1*. Copying takes place as if the *n* characters from the object pointed to by *s2* are first copied into a temporary array of *n* characters that does not overlap the objects pointed to by *s1* and *s2*, and then the *n* characters from the temporary array are copied into the object pointed to by *s1*.

`memmove()` returns the value of *s1*.

`memset()` Copy the value of *c* (converted to an `unsigned char`) into each of the first *n* bytes of the object pointed to by *s*. `memset()` returns the value of *s*.

`bcopy()` copies *n* bytes from the area pointed to by *s1* to the area pointed to by *s2*.

`bcmp()` Compare the first *n* bytes of the area pointed to by *s1* with the area pointed to by *s2*. `bcopy()` returns zero if they are identical; non-zero otherwise. Both areas are assumed to be *n* bytes in length.

`bzero()` Clear *n* bytes in the area pointed to by *s* by setting them to zero.

`ffs()` Find the first bit set (beginning with the least significant bit) and return the index of that bit. Bits are numbered starting at one. A return value of 0 indicates that *i* is zero.

International Code Set Support

These functions support only single-byte character code sets.

WARNING

The functions defined in <`string.h`> were previously defined in <`memory.h`>.

SEE ALSO

string(3C)

STANDARDS CONFORMANCE

`memccpy()`: AES, SVID2, XPG2, XPG3, XPG4
`memchr()`: AES, SVID2, XPG2, XPG3, XPG4, ANSI C
`memcmp()`: AES, SVID2, XPG2, XPG3, XPG4, ANSI C
`memcpy()`: AES, SVID2, XPG2, XPG3, XPG4, ANSI C
`memmove()`: AES, XPG4, ANSI C
`memset()`: AES, SVID2, XPG2, XPG3, XPG4, ANSI C

NAME

mkfifo() - make a FIFO file

SYNOPSIS

```
#include <sys/stat.h>

int mkfifo(char *path, mode_t mode);
```

DESCRIPTION

mkfifo() creates a new FIFO (first-in-first-out) file, at the path name to which *path* points. The file permission bits of the new file are initialized from the *mode* argument, as modified by the process's file creation mask: for each bit set in the process's file mode creation mask, the corresponding bit in the new file's mode is cleared (see *umask*(2)). Bits in *mode* other than the file permission bits are ignored.

The FIFO owner ID is set to the process's effective-user-ID. The FIFO group ID is set to the group ID of the parent directory if the set-group-ID bit is set on that directory. Otherwise the FIFO group ID is set to the process's effective group ID.

For details of the I/O behavior of pipes see *read*(2) and *write*(2).

The following symbolic constants are defined in the **<sys/stat.h>** header, and should be used to construct the value of the *mode* argument. The value passed should be the bitwise inclusive OR of the desired permissions:

S_IRUSR	Read by owner.
S_IWUSR	Write by owner.
S_IRGRP	Read by group.
S_IWGRP	Write by group.
S_IROTH	Read by other users.
S_IWOTH	Write by other users.

RETURN VALUE

mkfifo() returns 0 upon successful completion. Otherwise, it returns -1, no FIFO is created, and **errno** is set to indicate the error.

ERRORS

mkfifo() fails and the new file is not created if any of the following conditions are encountered:

[EACCES] A component of the path prefix denies search permission.

[EEXIST] The named file already exists.

[EFAULT] The *path* argument points outside the process's allocated address space. The reliable detection of this error is implementation dependent.

[ELOOP] Too many symbolic links encountered in translating the path name.

[ENAMETOOLONG]
The length of the specified path name exceeds **PATH_MAX** bytes, or the length of a component of the path name exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect.

[ENOENT] A component of the path prefix does not exist.

[ENOENT] The *path* argument is null.

[ENOSPC] Not enough space on the file system.

[ENOTDIR] A component of the path prefix is not a directory.

[EROFS] The directory in which the file is being created is located in a read-only file system.

SEE ALSO

chmod(2), mknod(2), pipe(2), stat(2), umask(2), cdf(4), fs(4), mknod(5), stat(5), types(5).

AUTHOR

mkfifo() was developed by HP.

STANDARDS CONFORMANCE

mkfifo(): AES, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

mktemp(), mkstemp() - make a unique file name

SYNOPSIS

```
#include <unistd.h>

char *mktemp(char *template);

int mkstemp(char *template);
```

Remarks:

These functions are provided solely for backward compatibility and importability of applications, and are not recommended for new applications where portability is important. For portable applications, use `tmpfile()` instead (see *tmpfile*(3S)).

DESCRIPTION

`mktemp()` replaces the contents of the string pointed to by *template* by a unique file name, and returns the address of *template*. The string in *template* should look like a file name with six trailing **X**s; `mktemp()` replaces the **X**s with a letter and the current process ID. The letter is chosen such that the resulting name does not duplicate the name of an existing file. If there are fewer than six **X**s, the letter is dropped first, followed by dropping the high-order digits of the process ID.

`mkstemp()` makes the same replacement to the template, but also returns a file descriptor for the template file after opening the file for reading and writing. `mkstemp()` thus prevents any possible race condition between testing whether the file exists and opening it for use.

RETURN VALUE

`mktemp()` returns its argument except when it runs out of letters, in which case the result is a pointer to the empty string `""`.

`mkstemp()` returns an open file descriptor upon successful completion, or -1 if no suitable file could be created.

SEE ALSO

getpid(2), open(2), tmpfile(3S), tmpnam(3S).

WARNINGS

It is possible to run out of letters.

`mktemp()` and `mkstemp()` do not check to determine whether the file name part of *template* exceeds the maximum allowable file name length.

STANDARDS CONFORMANCE

`mktemp()`: SVID2, XPG2

NAME

mktimer - allocate a per-process timer

SYNOPSIS

```
#include <sys/timers.h>

timer_t mktimer(int clock_type, int notify_type, void *itimercbp);
```

DESCRIPTION

mktimer() is used to allocate a per-process timer using the specified system-wide clock as the timing base. **mktimer()** returns an unique timer ID of type *timer_t* used to idnetify the timer in timer requests (see *gettimer*(3C)). *clock_type* specifies the system-wide clock to be used as the timing base for the new timer. *nofity_type* specifies the mechanism by which the process is to be notified when the timer expires.

mktimer() supports one per-process timer with a *clock_type* of **TIMEOFDAY** and *notify_type* of **DELIVERY_SIGNALS**.

If *notify_type* is **DELIVERY_SIGNALS**, the system causes a **SIGALRM** signal to be sent to the process whenever the timer expires.

For *clock_type* **TIMEOFDAY**, the machine-dependent clock resolution and maximum value are **1/HZ** and **MAX_ALARM** seconds respectively. These constants are defined in **<sys/param.h>**.

RETURN VALUE

Upon successful completion, **mktimer()** returns a **timer_t** which can be passed to the per_process timer calls. If unsuccessful, **mktimer()** returns a value of (**timer_t**)−1 and sets **errno** to indicate the error.

ERRORS

mktimer() fails if any of the following conditions are encountered:

[EAGAIN] The calling process has already allocated all of the timers it is allowed.

[EINVAL] *clock_type* is not defined, or does not allow the specified notification mechanism.

SEE ALSO

getclock(3C), gettimer(3C), reltimer(3C), rmtimer(3C), setclock(3C), <sys/timers.h>, <sys/param.h>.

STANDARDS CONFORMANCE

mktimer(): AES

NAME

monitor() - prepare execution profile

SYNOPSIS

```
#include <mon.h>

void monitor(
    void (*lowpc)(),
    void (*highpc)(),
    WORD *buffer,
    int bufsize,
    int nfunc
);
```

DESCRIPTION

An executable program created by `cc -p` automatically includes calls for `monitor()` with default parameters; `monitor()` need not be called explicitly except to gain fine control over profiling.

`monitor()` is an interface to *profil*(2). *lowpc* and *highpc* are the addresses of two functions; *buffer* is the address of a (user-supplied) array of *bufsize* WORDs (defined in the <mon.h> header file). `monitor()` arranges to record in the buffer a histogram of periodically sampled values of the program counter, and of counts of calls of certain functions. The lowest address sampled is that of *lowpc* and the highest is just below *highpc*. *lowpc* must not equal 0 for this use of *monitor*. Not more than *nfunc* call counts can be kept; only calls of functions compiled with the profiling option `-p` of *cc*(1) are recorded. (The C Library and Math Library supplied when `cc -p` is used also have call counts recorded.)

For results to be significant, especially where there are small, heavily used routines, it is suggested that the buffer be no more than a few times smaller than the range of locations sampled.

To profile the entire program, it is sufficient to use

```
extern etext;
        ...
monitor ((int (*)())2, ((int(*)())& etext, buf, bufsize, nfunc);
```

etext lies just above all the program text (see *end*(3C)).

To stop execution monitoring and write the results on file `mon.out`, use

```
monitor ((int (*)())0, (int(*)())0, 0, 0, 0);
```

prof(1) can then be used to examine the results.

FILES

```
/lib/libp/libc.a
/lib/libp/libm.a
mon.out
```

SEE ALSO

cc(1), prof(1), profil(2), end(3C).

STANDARDS CONFORMANCE

`monitor()`: SVID2, XPG2

NAME

mount() - keep track of remotely mounted filesystems

SYNOPSIS

```
#include <rpcsvc/mount.h>
```

DESCRIPTION

program number:

MOUNTPROG

Following are the xdr routines provided:

```
xdr_exportbody(xdrs, ex)
        XDR *xdrs;
        struct exports *ex;
xdr_exports(xdrs, ex);
        XDR *xdrs;
        struct exports **ex;
xdr_fhandle(xdrs, fh);
        XDR *xdrs;
        fhandle_t *fp;
xdr_fhstatus(xdrs, fhs);
        XDR *xdrs;
        struct fhstatus *fhs;
xdr_groups(xdrs, gr);
        XDR *xdrs;
        struct groups *gr;
xdr_mountbody(xdrs, ml)
        XDR *xdrs;
        struct mountlist *ml;
xdr_mountlist(xdrs, ml);
        XDR *xdrs;
        struct mountlist **ml;
xdr_path(xdrs, path);
        XDR *xdrs;
        char **path;
```

procs:

```
MOUNTPROC_MNT
        argument of xdr_path, returns fhstatus.
        Requires unix authentication.
MOUNTPROC_DUMP
        no args, returns struct mountlist
MOUNTPROC_UMNT
        argument of xdr_path, no results.
        requires unix authentication.
MOUNTPROC_UMNTALL
        no arguments, no results.
        requires unix authentication.
        umounts all remote mounts of sender.
MOUNTPROC_EXPORT
MOUNTPROC_EXPORTALL
        no args, returns struct exports
```

versions:

MOUNTVERS_ORIG

structures:

```
struct mountlist {              /* what is mounted */
        char *ml_name;
        char *ml_path;
        struct mountlist *ml_nxt;
};
```

```
struct fhstatus {
        int fhs_status;
        fhandle_t fhs_fh;
};
/*
 * List of exported directories
 * An export entry with ex_groups
 * NULL indicates an entry which is exported to the
 * world.
 */
struct exports {
        dev_t         ex_dev;   /* dev of directory */
        char         *ex_name;  /* name of directory */
        struct groups  *ex_groups; /* groups allowed to */
                                   /* mount this entry */
        struct exports *ex_next;
};
struct groups {
        char         *g_name;
        struct groups  *g_next;
};
```

AUTHOR

mount() was developed by Sun Microsystems, Inc.

SEE ALSO

mount(1M), mountd(1M), showmount(1M).

NAME

mblen(), mbtowc(), mbstowcs(), wctomb(), wcstombs() - multibyte characters and strings conversions

SYNOPSIS

```
#include <stdlib.h>

int mblen(const char *s, size_t n);

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

int wctomb(char *s, wchar_t wchar);

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

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

DESCRIPTION

A multibyte character is composed of one or more bytes that represent a "whole" character in a character encoding. A wide character (type of `wchar_t`) is composed of a fixed number of bytes whose code value can represent any character in a character encoding.

`mblen()` Determine the number of bytes in the multibyte character pointed to by *s*. Equivalent to:

```
mbtowc((wchar_t *)0, s, n);
```

If *s* is a null pointer, *mblen* returns a nonzero or zero value, depending on whether the multibyte character encodings do or do not have state-dependent encodings, respectively. Since no character encodings currently supported by HP-UX are state-dependent, zero is always returned in this case. However, for maximum portability to other systems, application programs should not depend on this.

If *s* is not a null pointer, *mblen* returns the number of bytes in the multibyte character if the next *n* or fewer bytes form a valid multibyte character, or return -1 if they do not form a valid multibyte character. If *s* points to the null character, *mblen* returns 0.

`mbtowc()` Determine the number of bytes in the multibyte character pointed to by *s*, determine the code for the value of type `wchar_t` corresponding to that multibyte character, then store the code in the object pointed to by *pwc*. The value of the code corresponding to the null character is zero. At most *n* characters are examined, starting at the character pointed to by *s*.

If *s* is a null pointer, `mbtowc()` returns a non-zero or zero value, depending on whether the multibyte character encodings do or do not have state-dependent encodings, respectively. Since no character encodings currently supported by HP-UX are state-dependent, zero is always returned in this case. However, for maximum portability to other systems, application programs should not depend on this.

If *s* is not a null pointer, `mbtowc()` returns the number of bytes in the converted multibyte character if the next *n* or fewer bytes form a valid multibyte character, or -1 if they do not form a valid multibyte character. If *s* points to the null character, `mbtowc()` returns 0. The value returned is never greater than *n* or the value of the `MB_CUR_MAX` macro.

`wctomb()` Determine the number of bytes needed to represent the multibyte character corresponding to the code whose value is *wchar* and store the multibyte character representation in the array object pointed to by *s*. At most `MB_CUR_MAX` characters are stored.

If *s* is a null pointer, `wctomb()` returns a nonzero or zero value, depending on whether the multibyte character encodings do or do not have state-dependent encodings, respectively. Since no character encodings currently supported by HP-UX are state-dependent, zero is always returned in this case. However, for maximum portability to other systems, application programs should not depend on this.

If *s* is not a null pointer, `wctomb()` returns the number of bytes in the multibyte character corresponding to the value of *wchar*, or -1 if the value of *wchar* does not correspond to a valid multibyte character. The value returned is never greater than the value of the `MB_CUR_MAX` macro.

`mbstowcs()`
Convert a sequence of multibyte characters from the array pointed to by *s* into a sequence of

corresponding codes and store these codes into the array pointed to by *pwcs*, stopping after either *n* codes or a code with value zero (a converted null character) is stored. Each multibyte character is converted as if by a call to `mbtowc()`. No more than *n* elements are modified in the array pointed to by *pwcs*.

If an invalid multibyte character is encountered, `mbstowcs()` returns (size_t)−1. Otherwise, `mbstowcs()` returns the number of array elements modified, not including a terminating zero code, if any. The array is not null- or zero-terminated if the value returned is *n*. If *pwcs* is a null pointer, `mbstowcs()` returns the number of elements required for the wide-character-code array.

`wcstombs()`
Convert a sequence of codes corresponding to multibyte characters from the array pointed to by *pwcs* into a sequence of multibyte characters and store them into the array pointed to by *s*, stopping if a multibyte character exceeds the limit of *n* total bytes or if a null character is stored. Each code is converted as if by a call to `wctomb()`. No more than *n* bytes are modified in the array pointed to by *s*.

If a code is encountered that does not correspond to a valid multibyte character, `wcstombs()` returns (size_t)−1. Otherwise, `wcstombs()` returns the number of bytes modified, not including a terminating null character, if any. The array is not null- or zero-terminated if the value returned is *n*. If *s* is a null pointer, `wcstombs()` returns the number of bytes required for the character array.

EXTERNAL INFLUENCES

Locale

The `LC_CTYPE` category determines the behavior of the multibyte character and string functions.

ERRORS

`mblen()`, `mbstowcs()`, `mbtowc()`, `wcstombs()` and `wctomb()` may fail and `errno` is set if the following condition is encountered:

[EILSEQ] An invalid multibyte sequence or wide character code was found.

WARNINGS

With the exception of ASCII characters, the code values of wide characters (type of `wchar_t`) are specific to the effective locale specified by the `LC_CTYPE` environment variable. These values may not be compatible with values obtained by specifying other locales that are supported now, or which may be supported in the future. It is recommended that wide character constants and wide string literals (see the *C Reference Manual*) not be used, and that wide character code values not be stored in files or devices because future standards may dictate changes in the code value assignments of the wide characters. However, wide character constants and wide string literals corresponding to the characters of the ASCII code set can be safely used since their values are guaranteed to be the same as their ASCII code set values.

AUTHOR

The multibyte functions in this entry were developed by HP.

SEE ALSO

setlocale(3C), nl_tools_16(3C), wctype(3X).

STANDARDS CONFORMANCE

`mblen()`: AES, XPG4, ANSI C
`mbstowcs()`: AES, XPG4, ANSI C
`mbtowc()`: AES, XPG4, ANSI C
`wcstombs()`: AES, XPG4, ANSI C
`wctomb()`: AES, XPG4, ANSI C

NAME

dbm_open, dbm_close, dbm_fetch, dbm_store, dbm_delete, dbm_firstkey, dbm_nextkey, dbm_error, dbm_clearerr - database subroutines

SYNOPSIS

```
#include <ndbm.h>

DBM *dbm_open(const char *file, int flags, int mode);

void dbm_close(DBM *db);

datum dbm_fetch(DBM *db, datum key);

int dbm_store(DBM *db, datum key, datum content, int flags);

int dbm_delete(DBM *db, datum key);

datum dbm_firstkey(DBM *db);

datum dbm_nextkey(DBM *db);

int dbm_error(DBM *db);

int dbm_clearerr(DBM *db);
```

DESCRIPTION

These functions maintain key/content pairs in a database. They handle very large (a billion blocks (block = 1024 bytes)) databases and can access a keyed item in one or two file system accesses. This package replaces the earlier *dbm*(3X) library, which managed only a single database. The functions can be accessed by giving the **-lndbm** option to *ld*(1) or *cc*(1).

key and *content* parameters are described by the **datum** type. A **datum** specifies a string of *dsize* bytes pointed to by *dptr*. Arbitrary binary data, as well as normal ASCII strings, are allowed. The database is stored in two files. One file is a directory containing a bit map of keys and has **.dir** as its suffix. The second file contains all data and has **.pag** as its suffix.

Before a database can be accessed, it must be opened by **dbm_open**. This will open and/or create the files *file* **.dir** and *file* **.pag** depending on the *flags* parameter (see *open*(2)).

Once open, the data stored under a key is accessed by **dbm_fetch** and data is placed under a key by **dbm_store**. The *flags* field can be either **DBM_INSERT** or **DBM_REPLACE**. **DBM_INSERT** can only insert new entries into the database, and cannot change an existing entry having the same key. **DBM_REPLACE** replaces an existing entry if it has the same key. A key (and its associated contents) is deleted by **dbm_delete**. A linear pass through all keys in a database can be made in (apparently) random order by use of **dbm_firstkey** and **dbm_nextkey**. **dbm_firstkey** returns the first key in the database. **dbm_nextkey** returns the next key in the database, The following code can be used to traverse the database:

```
for (key = dbm_firstkey(db); key.dptr != NULL; key = dbm_nextkey(db))
```

dbm_error returns non-zero when an error has occurred reading or writing the database. **dbm_clearerr** resets the error condition on the named database.

DIAGNOSTICS

All functions that return an **int** indicate errors with negative values and success with zero. Functions that return a **datum** indicate errors with a null *dptr*. If **dbm_store** is called with a *flags* value of **DBM_INSERT** and finds an existing entry with the same key, a value of **1** is returned.

WARNINGS

The *ndbm* functions provided in this library should not be confused in any way with those of a general-purpose database management system such as ALLBASE/HP-UX SQL. These functions *do not* provide for multiple search keys per entry, they *do not* protect against multi-user access (in other words they do not lock records or files), and they *do not* provide the many other useful database functions that are found in more robust database management systems. Creating and updating databases by use of these functions is relatively slow because of data copies that occur upon hash collisions. These functions *are useful* for applications requiring fast lookup of relatively static information that is to be indexed by a single key.

The **.pag** file will contain holes so that its apparent size is about four times its actual content. Some older UNIX systems create real file blocks for these holes when touched. These files cannot be copied by normal

means (such as *cp*(1), *cat*(1), *tar*(1), or *ar*(1)) without expansion.

dptr pointers returned by these subroutines point into static storage that is changed by subsequent calls.

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 1024 bytes). Moreover, all key/content pairs that hash together must fit on a single block. **dbm_store** returns an error in the event that a disk block fills with inseparable data.

dbm_delete does not physically reclaim file space, although it does make it available for reuse.

The order of keys presented by **dbm_firstkey** and **dbm_nextkey** depends on a hashing function, not on anything interesting.

A **dbm_store** or **dbm_delete** during a pass through the keys by **dbm_firstkey** and **dbm_nextkey** may yield unexpected results.

AUTHOR

ndbm(3X) was developed by the University of California, Berkeley.

SEE ALSO

dbm(3X).

NAME

net_aton(), net_ntoa() - network station address string conversion routines

SYNOPSIS

```
#include <sys/netio.h>

char *net_aton(char *dstr, const char *sstr, int size);

char *net_ntoa(char *dstr, const char *sstr, int size);
```

DESCRIPTION

`net_aton()` and `net_ntoa()` translate station addresses between hexadecimal, octal or decimal, and binary formats:

`net_aton()`	converts a hexadecimal, octal or decimal address to a binary address;
`net_ntoa()`	converts a binary address to an ASCII hexadecimal address.

Both routines are provided in the standard C library and are loaded automatically during compilation.

net_aton Parameters

The following parameters are used by `net_aton()`:

dstr	Pointer to the binary address returned by the function.
sstr	Pointer to a null-terminated ASCII form of a station address (Ethernet or IEEE 802.3). This address can be an octal, decimal, or hexadecimal number as used in the C language (in other words, a leading 0x or 0X implies hexadecimal; a leading 0 implies octal; otherwise, the number is interpreted as decimal).
size	Length of the binary address to be returned in *dstr*. The length is 6 for Ethernet/IEEE 802.3 addresses.

net_ntoa Parameters

`net_ntoa()` converts a 48-bit binary station address to its ASCII hexadecimal equivalent. The following parameters are used by `net_ntoa()`:

dstr	Pointer to the ASCII hexadecimal address returned by the function. *dstr* is null-terminated and padded with leading zeroes if necessary. *dstr* must be at least ($2 \times size + 3$) bytes long to accommodate the size of the converted address.
sstr	Pointer to a station address in its binary form.
size	Length of *sstr*.

RETURN VALUE

`net_aton()` and `net_ntoa()` return NULL if any error occurs.

EXAMPLES

```
#include <netio.h>
#define  destination_addr "0x00DD0002AD00"
      ...
struct fis arg;
char str[16];
      ...
(void) net_aton(arg.value.s, destination_addr, 6);
                /* arg.value.s = "<48-bit binary value>"  */
(void) net_ntoa(str, arg.value.s, 6);
                /* str = "0x00DD0002AD00" */
```

AUTHOR

`net_aton()` was developed by HP.

SEE ALSO

lan(7).

NAME

nl_toupper(), nl_tolower() - translate characters for use with NLS

SYNOPSIS

```
#include <nl_ctype.h>

int nl_toupper(int c, int langid);

int nl_tolower(int c, int langid);
```

DESCRIPTION

`nl_toupper()` and `nl_tolower()` are extensions of their counterparts in the *conv*(3C) manual entry. They function in the same way, but have a *langid* parameter (see *lang*(5)) whose value represents a supported language. If *langid* is not valid, or if the NLS environment corresponding to *langid* is not available, `n-computer`, the default NLS environment associated with `langinit()`, is used (see *nl_init*(3C)).

WARNINGS

These routines are provided for historical reasons only. Use of the alternate functions listed by *conv*(3C) which provide for international support via *setlocale*(3C) is recommended.

`nl_toupper()` and `nl_tolower()` effectively call `langinit()` to load the NLS environment according to the language specified by *langid*.

AUTHOR

`nl_conv()` was developed by the HP.

SEE ALSO

conv(3C), nl_init(3C), hpnls(5), lang(5).

EXTERNAL INFLUENCES

Locale

The `LC_CTYPE` category determines the translations to be done.

International Code Set Support

Single-byte character code sets are supported.

NAME

nl_isalpha(), nl_isupper(), nl_islower(), nl_isdigit(), nl_isxdigit(), nl_isalnum(), nl_isspace(), nl_ispunct(), nl_isprint(), nl_isgraph(), nl_iscntrl() - classify characters for use with NLS

SYNOPSIS

```
#include <nl_ctype.h>

int nl_isalpha(int c, int langid);

...
```

DESCRIPTION

These routines classify character-coded integer values by table lookup. *langid* corresponds to a particular NLS environment (see *lang*(5)). Each is a predicate returning nonzero for true, zero for false. All are defined for the range −1 to 255. If *langid* is not defined, or if the NLS environment corresponding to *langid* is not available, `n-computer`, the default NLS environment associated with `langinit()`, is used (see *nl_init*(3C)).

`nl_isalpha()`	*c* is a letter.
`nl_isupper()`	*c* is an uppercase letter.
`nl_islower()`	*c* is a lowercase letter.
`nl_isdigit()`	*c* is a decimal digit (in ASCII: characters [0-9]).
`nl_isxdigit()`	*c* is a hexadecimal digit (in ASCII: characters [0-9], [A-F] or [a-f]).
`nl_isalnum()`	*c* is an alphanumeric (letters or digits).
`nl_isspace()`	*c* is a character that creates "white space" in displayed text (in ASCII: space, tab, carriage return, new-line, vertical tab, and form-feed).
`nl_ispunct()`	*c* is a punctuation character (in ASCII: any printing character except the space character (040), digits, letters.)
`nl_isprint()`	*c* is a printing character.
`nl_isgraph()`	*c* is a visible character (in ASCII: printing characters, excluding the space character (040)).
`nl_iscntrl()`	*c* is a control character (in ASCII: character codes less than 040 and the delete character (0177)).

DIAGNOSTICS

If the argument to any of these is not in the domain of the function, the result is undefined.

WARNINGS

These macros are provided for historical reasons only. Use of the macros in *ctype*(3C), which now provide for international support via *setlocale*(3C), is recommended.

Macros described in this manual entry call `langinit()` to load the NLS environment according to the language specified by *langid*.

AUTHOR

`nl_ctype()` was developed by the HP.

SEE ALSO

ctype(3C), nl_init(3C), hpnls(5), lang(5).

EXTERNAL INFLUENCES

Locale

The `LC_CTYPE` category determines the classification of character type.

International Code Set Support

Single-byte character code sets are supported.

NAME

nl_init(), langinit() - initialize the NLS environment of a program

SYNOPSIS

```
#include <langinfo.h>

int nl_init(const char *langname);

int langinit(const char *langname);
```

DESCRIPTION

`nl_init()` Initializes the NLS (Native Language Support) environment of a program to the language specified by *langname*. If *langname* is null or points to an empty string, the default-mode language, `n-computer` (see *lang*(5)), is initialized.

`nl_init()` affects the behavior of the macros and routines defined in *conv*(3C), *ctime*(3C), *ctype*(3C), *ecvt*(3C), *langinfo*(3C), *multibyte*(3C), *nl_langinfo*(3C), *nl_string*(3C), *nl_tools_16*(3C), *printf*(3S), *printmsg*(3C), *scanf*(3S), *strftime*(3C), *string*(3C), *strtod*(3C), and *vprintf*(3S).

Typically, `nl_init()` is used to bind program operation to the end-user's specified language requirements. For example,

```
nl_init(getenv("LANG"));
```

Prior to successfully calling `nl_init()`, functions supporting NLS operate as though the default-mode language `n-computer` had been initialized.

`langinit()`
Performs the same initialization of the environment control areas as does `nl_init()`. However, `nl_init()` and `langinit()` differ in the action taken when the requested language environment cannot be initialized (see ERRORS below).

RETURN VALUE

`nl_init()` and `langinit()` return 0 if the environment is successfully initialized to the requested language. Otherwise, they return –1.

ERRORS

`nl_init()` fails if the string specified by *langname* does not identify a valid language name (see *lang*(3C)), or the language is not available on the system.

If `nl_init()` fails but had previously succeeded, operation continues with the environment initialized by the last successful call. If `nl_init()` fails and has never been called successfully, the environment reverts to the default-mode language `n-computer`.

If `langinit()` fails, the environment reverts to the default-mode language `n-computer`.

WARNINGS

`nl_init()` and `langinit()` are provided for historical reasons only. Use `setlocale()` instead (see *setlocale*(3C)). The default processing language for `setlocale()` is "C"; the default processing language for `nl_init()` is `n-computer`. This is maintained for backward portability.

`langinit()` is implicitly called by the macros and routines which use a *langid* parameter (see *ctime*(3C), *langinfo*(3C), *nl_conv*(3C), *nl_ctype*(3C), *nl_string*(3C), and *strtod*(3C)). Using any *langid* parameter routine or macro initializes the environment of the associated language name, thus affecting the behavior of other routines that interact with the NLS environment. For maximum portability and performance, use of macros and routines without the *langid* parameter is recommended.

AUTHOR

`nl_init()` was developed by HP.

SEE ALSO

conv(3C), ctime(3C), ctype(3C), ecvt(3C), langinfo(3C), multibyte(3C), nl_conv(3C), nl_ctype(3C), nl_langinfo(3C), nl_string(3C), nl_tools_16(3C), printf(3S), printmsg(3C), scanf(3S), string(3C), strtod(3C), vprintf(3S), environ(5), hpnls(5), lang(5), nl_langinfo(5).

STANDARDS CONFORMANCE

`nl_init()`: XPG2

NAME

nl_langinfo() - language information

SYNOPSIS

```
#include <langinfo.h>

char *nl_langinfo(nl_item item);
```

DESCRIPTION

`nl_langinfo()` returns a pointer to a null-terminated string containing information relevant to a particular language or cultural area defined in the program's locale (see *setlocale*(3C)). The manifest constant names and values of *item* are defined in `<langinfo.h>`. For example:

```
nl_langinfo( ABDAY_1 )
```

returns a pointer to the string "Dom" if the language identified by the current locale is Portuguese, and "Sun" if the identified language is Finnish.

If an invalid *item* is specified, a pointer to an empty string is returned. An empty string can also be returned for a valid *item* if that *item* is not applicable to the language or customs of the current locale. For example, a thousands separator is not used when writing numbers according to the customs associated with the Arabic language.

EXTERNAL INFLUENCES

Locale

The string returned for a particular *item* is determined by the locale category specified for that item in *langinfo*(5).

International Code Set Support

Single- and multi-byte character code sets are supported.

WARNINGS

`nl_langinfo()` returns a pointer to a static area that is overwritten on each call.

AUTHOR

`nl_langinfo()` was developed by HP.

SEE ALSO

localeconv(3C), setlocale(3C), hpnls(5), langinfo(5).

STANDARDS CONFORMANCE

nl_langinfo: XPG2, XPG3

NAME

strcmp8(), strncmp8(), strcmp16(), strncmp16() - non-ASCII string collation

SYNOPSIS

```
#include <nl_types.h>

int strcmp8(
    const unsigned char *s1,
    const unsigned char *s2,
    int langid,
    int *status
);

int strncmp8(
    const unsigned char *s1,
    const unsigned char *s2,
    size_t n,
    int langid,
    int *status
);

int strcmp16(
    const unsigned char *s1,
    const unsigned char *s2,
    const unsigned char *file_name,
    int *status
);

int strncmp16(
    const unsigned char *s1,
    const unsigned char *s2,
    size_t n,
    const unsigned char *file_name,
    int *status
);
```

DESCRIPTION

strcmp8() Compares string *s1* and *s2* according to the collating sequence of the NLS environment specified by *langid* (see *lang*(5)). If *langid* is invalid, or if the NLS environment corresponding to *langid* is unavailable, **n-computer**, the default NLS environment associated with **langinit()** is used (see *nl_init*(3C)). An integer greater than, equal to, or less than 0 is returned, depending on whether *s1* is, respectively, greater than, equal to, or less than *s2*. Trailing blanks in strings *s1* and *s2* are ignored.

strncmp8() Same as **strcmp8()**, but looks at a maximum of *n* characters.

strcmp16() Compares strings *s1* and *s2* and returns an integer greater than, equal to, or less than 0 depending on whether *s1* is, respectively, greater than, equal to, or less than *s2*. Strings *s1* and *s2* can contain 16-bit characters mixed with 7-bit and 8-bit characters (see *hpnls*(5)). Strings *s1* and *s2* are compared, with 8-bit characters collating before 16-bit characters.

strncmp16() Same as **strcmp16()**, but looks at a maximum of *n* characters.

nl_init() must be called before the first call to **strcmp16()** or **strncmp16()** (see *nl_init*(3C)).

ERRORS

If an error condition is encountered, the integer pointed to by *status* is set to one of the non-zero values (listed below) defined in **<langinfo.h>**. For **ENOCFFILE** and **ENOLFILE**, **errno** indicates that a file system call failed.

[ENOCFFILE] Attempt to access file **/usr/lib/nls/config** has failed.

[ENOCONV] The entry for the language sought is not in the file **/usr/lib/nls/config**.

[ENOLFILE] Access to the NLS environment corresponding to *langid* or *file_name* has failed.

WARNINGS

These routines are provided for historical reasons only. Use `strcoll()` instead (see *string*(3C)). However, note that all characters are significant to `strcoll()`, whereas `strcmp8()` and `strncmp8()` ignore trailing blanks.

`strcmp16()` and `strncmp16()` do not support a collation sequence table. (A null string must be passed as *file_name* to maintain the correct argument count.)

`strcmp8()` and `strncmp8()` call `langinit()` (see *nl_init*(3C)) to load the NLS environment according to the language specified by *langid*.

AUTHOR

`nl_string()` was developed by HP.

SEE ALSO

nl_init(3C), string(3C), hpnls(5), lang(5).

EXTERNAL INFLUENCES

Locale

The `LC_CTYPE` category determines the interpretation of the bytes within the string arguments to `strcmp8()`, `strncmp8()`, `strcmp16()` , and `strncmp16()` as single- and/or multi-byte characters.

The `LC_COLLATE` category determines the collation ordering used by the `strcmp8()` and `strncmp8()`. See *hpnls*(5) for a description of supported collation features. See *nlsinfo*(1) to view the collation used for a particular locale.

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

firstof2(), secof2(), byte_status(), c_colwidth(), FIRSTof2(), SECof2(), BYTE_STATUS(), C_COLWIDTH(), CHARAT(), ADVANCE(), CHARADV(), WCHAR(), WCHARADV() - tools to process 16-bit characters

SYNOPSIS

```
#include <nl_ctype.h>
int firstof2(int c);
int secof2(int c);
int byte_status(int c, int laststatus);
int c_colwidth(int c);
int FIRSTof2(int c);
int SECof2(int c);
int BYTE_STATUS(int c, int laststatus);
int C_COLWIDTH(int c);
int CHARAT(const char *p);
int ADVANCE(const char *p);
int CHARADV(const char *p);
int WCHAR(wchar_t wc, char *p);
int WCHARADV(wchar_t wc, char *p);
void PCHAR(int c, char *p);
void PCHARADV(int c, char *p);
```

Remarks

All interfaces listed above whose names begin with a capital letter are implemented as macros; the others are functions.

DESCRIPTION

The following macros and routines perform their operations based upon the loaded NLS environment (see *setlocale*(3C)).

`FIRSTof2()` Takes a byte and returns a non-zero value if it can be the first byte of a two-byte character according to the NLS environment loaded, and zero if it cannot.

`SECof2()` Takes a byte and returns a non-zero value if it can be the second byte of a two-byte character according to the loaded NLS environment, and zero if it cannot.

`BYTE_STATUS()` Returns one of the following values based on the value of the current byte in *c* and the status of the previous byte interpreted in *laststatus* as returned by the last call to `BYTE_STATUS()`. These are the status values as defined in `<nl_ctype.h>`:

`ONEBYTE`	Single-byte character
`SECOF2`	Second byte of two-byte character
`FIRSTOF2`	First byte of two-byte character

To validate a two-byte character, both the first and second bytes must be valid. If the value of *laststatus* is `FIRSTOF2` but `SECof2(c)` returns false, `BYTE_STATUS(`*c*`,` *laststatus*`)` returns `ONEBYTE`.

`C_COLWIDTH()`
Takes a byte which is assumed to be either a one-byte character or the first byte of a two-byte character, and returns the number of columns the character would occupy on a terminal display.

For the macros `FIRSTof2()`, `SEC0of2()`, `BYTE_STATUS()` , and `C_COLWIDTH()` results are undefined for values of *c* less than –1 (EOF) or greater than 255.

`CHARAT()`
Takes as an argument a pointer *p*, which is assumed to be pointing at either a one-byte character or the first byte of a two-byte character. In either case, `CHARAT()` returns the `wchar_t` value that

corresponds to the character pointed to by *p*.

ADVANCE()
Advances its pointer argument by the byte width of the character it is pointing at (either one or two bytes).

CHARADV()
Combines the functions of **CHARAT()** and **ADVANCE()** in a single macro. It takes as an argument a pointer *p*, which is assumed to be pointing at either a one-byte character or the first byte of a two-byte character. In either case **CHARADV()** returns the **wchar_t** value that corresponds to the character pointed to by *p*, and advances *p* beyond the last byte of the character.

WCHAR()
Converts the **wchar_t** value *wc* into the corresponding one or two byte character, and writes it at the location specified by *p*. **WCHAR()** returns the **wchar_t** value *wc*.

WCHARADV()
Combines the functions of **WCHAR()** and **ADVANCE()** in a single macro. It converts the **wchar_t** value *wc* into the corresponding one or two byte character, and writes it at the location specified by *p*, then advances *p* past the last byte. **WCHARADV()** returns the **wchar_t** value *wc*.

firstof2()
secof2()
byte_status()
chwidth()
Other versions of the corresponding macros. These functions can be called from languages other than C.

EXTERNAL INFLUENCES

Locale
The **LC_CTYPE** category determines the interpretation of single and/or multi-byte characters.

WARNINGS

For maximum portability, use the routines documented in the *multibyte*(3C) manual entry for multi-byte character processing.

Other macros listed in this manual entry cannot be used as the first argument to **WCHAR()** or **WCHARADV()**. For example,

```
*t++ = *f++
```

cannot be replaced by

```
WCHARADV(CHARADV (f),t).
```

Instead, use a method such as

```
int c; ... c = CHARADV (f), WCHARADV (c,t).
```

WCHAR() and **WCHARADV()** may produce a "null effect" warning from *lint*(1) if not used as part of another expression or as part of a statement. This does not affect the functionality of either macro.

Note that **WCHAR()** and **WCHARADV()** are not "replace_char" macros. They do not prevent the second byte of a two-byte character from being left dangling if **WCHAR()** or **WCHARADV()** overwrite the first byte of the two-byte character with a single-byte character.

CHARAT(), **ADVANCE()**, and **CHARADV()** do not examine the byte following the location pointed to by the argument to verify its validity as a **SECof2** byte.

AUTHOR

nl_tools_16() was developed by HP.

SEE ALSO

setlocale(3C), multibyte(3C), wconv(3X), wctype(3X), hpnls(5).

NAME

nlappend() - append the appropriate language identification to a valid MPE file name

SYNOPSIS

```
#include <portnls.h>

void nlappend(
    char *filename,
    short int langid,
    unsigned short int err[2]
);
```

DESCRIPTION

`nlappend()` replaces the first three blanks found in *filename* with the language number. Its purpose is to identify the language of a file in an operating system-independent manner.

Arguments to `nlappend()` are used as follows:

filename A string of up to eight ASCII characters terminated by three blanks.

langid A short integer specifying the language ID.

err The first element contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
4	*Filename* is not terminated by 3 blanks.

WARNINGS

This routine is provided for compatibility with MPE, a proprietary HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described by *hpnls*(5) for HP-UX NLS support.

AUTHOR

`nlappend()` was developed by HP.

SEE ALSO

portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlcollate() - compare two character strings according to the MPE language-dependent collating sequence

SYNOPSIS

```
#include <portnls.h>

void nlcollate(
    const char *string1,
    const char *string2,
    short int length,
    short *result,
    short int langid,
    unsigned short int err[2],
    const char *collseq
);
```

DESCRIPTION

nlcollate collates two character strings according to the collating sequence of the specified language. This routine's purpose is to determine a lexical ordering. It is not intended to be used for searching or matching.

If the *collseq* parameter points to the null address, and *langid* is specified as (or defaults to) a language in which binary collation is appropriate, the binary collation is used to compare the two indicated strings. Otherwise, the *collseq* array is used to determine the string-compare operation (note that this may be a binary collation).

Arguments to `nlcollate()` are as follows:

string1 One of the character strings to be collated.

string2 The second character string to be collated.

length The length of the string segments to be collated.

result The result of the character collation is stored in the short integer variable to which *result* points.

0	If *string1* collates equal to *string2*.
-1	If *string1* collates before *string2*.
1	If *string1* collates after *string2*.

langid

The language ID indicating the collating sequence to be used for the collation.

err

The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Invalid collating table entry.
4	Invalid length parameter.

collseq

An array containing the collating sequence to be used, as returned from a call to *nlinfo*(3X)'s *item-number* 11.

WARNINGS

This routine is provided for compatibility with MPE, a proprietary HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described by *hpnls*(5) for HP-UX NLS support.

AUTHOR

nlcollate was developed by HP.

SEE ALSO

nlinfo(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlconvclock() - check and convert time string to MPE internal format

SYNOPSIS

```
#include <portnls.h>

unsigned int nlconvclock(
    const char *instr,
    short int leninstr,
    short int langid,
    unsigned short int err[2]
);
```

DESCRIPTION

`nlconvclock()` converts *instr* to a general time format as returned by *nlinfo*(3X) *itemnumber* 3. This routine is the inverse of *nlfmtclock*(3X). Note that the seconds and tenths of seconds are always set to zero.

The arguments to `nlconvclock()` are used as follows:

instr A character buffer containing the time to be converted.

leninstr An unsigned short specifying the length of the buffer.

langid A short containing the language ID.

err The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Invalid time format.
4	Invalid length.

RETURN VALUE

`nlconvclock()` returns the time in the format:

Bits 0 7 8 15

Hour of Day	Minute of Hour

Bits 16 23 24 31

Seconds	Tenths of Seconds

WARNINGS

This routine is provided for compatibility with MPE, a proprietary HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described by *hpnls*(5) for HP-UX NLS support.

AUTHOR

`nlconvclock()` was developed by HP.

SEE ALSO

clock(3X), nlfmtclock(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlconvcustdate() - convert date string to MPE packed date format

SYNOPSIS

```
#include <portnls.h>

unsigned short nlconvcustdate(
    const char *instr,
    short int leninstr,
    short int langid,
    unsigned short int err[2]
);
```

DESCRIPTION

`nlconvcustda()` converts *instr* to a packed date format. This routine is the inverse of *nlfmtcustdate*(3X).

Arguments to `nlconvcustda()` are as follows:

instr A character buffer containing the date to be converted.

leninstr A positive integer specifying the length of the string (in bytes).

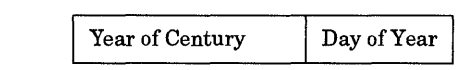
langid A short containing the language ID number.

err The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Invalid date format.
4	Invalid string length.

RETURN VALUE

The routine returns the date as an unsigned integer in the format:



WARNINGS

This routine is provided for compatibility with MPE, another HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described by *hpnls*(5) for HP-UX NLS support.

AUTHOR

`nlconvcustda()` was developed by HP.

SEE ALSO

calendar(3X), nlfmtcustdate(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlconvnum() - convert MPE native-language formatted number to ASCII number

SYNOPSIS

```
#include <portnls.h>

void nlconvnum(
    short int langid,
    const char *instr,
    short int leninstr,
    char *outstr,
    short int *plenoutstr,
    unsigned short int err[2],
    const char *numspec,
    short int fmtmask,
    short int *pdecimals
);
```

DESCRIPTION

`nlconvnum()` converts a native-language formatted number to an ASCII number, with an n-computer decimal separator (`.`) and thousands separator (`,`), to use for further conversion to INTEGER, REAL, etc.

This routine converts the decimal separator and the thousands separators to the n-computer equivalent, or strips them, according to the value of *fmtmask*. If *fmtmask* and `M_NUMBERSONLY` is not zero, *instr* is validated as a number. If it is null, no validation takes place.

For languages using an alternate set of digits (currently only `arabic`, which uses HINDI digits), `nlconvnum()` also converts these digits to ASCII digits so they can be recognized and used as numeric characters.

Arguments to `nlconvnum()` are as follows:

langid A language ID number.

instr A character buffer containing the native language formatted number to convert. Leading and trailing spaces are ignored.

leninstr Length, in bytes, of *instr*.

outstr Output buffer; an array containing the converted output. The output is left-justified in the buffer, and *plenoutstr* contains the actual length of the converted number. *outstring* may refer to the same address as *instr*.

plenoutstr A pointer to the length, in bytes, of *outstr*. After a successful call to *nlconvnum*, the short integer to which *plenoutstr* points contains the actual length of the converted number.

err The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Invalid length specified (*leninstr* or *plenoutstr*).
4	Invalid number specified (*instr*).
7	Truncation has occurred (*outstr* is left partially formatted).
8	Invalid *numspec* parameter.
9	Invalid *fmtmask* parameter.

numspec
A character buffer, as returned from *nlnumspec*, containing information about correct formatting. If this parameter is not null, *langid* is ignored and performance is improved (see the description of *nlnumspec*).

fmtmask
An unsigned short specifying how to format the number. The default value is zero, which means substitution only, convert thousands separators, convert decimal separators, and that *instr* can contain any

character.

Value	Description
`M_STRIPTHOU`	Strip thousands separators.
`M_STRIPDEC`	Strip decimal separators.
`M_NUMBERSONLY`	*instr* contains a number.

pdecimals
Pointer to a variable in which the number of decimal places in the input number is returned.

WARNINGS
This routine is provided for compatibility with MPE, a proprietary HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described by *hpnls*(5) for HP-UX NLS support.

AUTHOR
`nlconvnum()` was developed by HP.

SEE ALSO
nlfmtnum(3X), portnls(5).

EXTERNAL INFLUENCES
International Code Set Support
Single- and multi-byte character code sets are supported.

NAME

nlfindstr() - search for a string in another string using the MPE character set definition

SYNOPSIS

```
#include <portnls.h>

short int nlfindstr(
    short int langid,
    const char *string1,
    short int length1,
    const char *string2,
    short int length2,
    unsigned short int err[2],
    const char *charset
);
```

DESCRIPTION

`nlfindstr()` searches for the first occurrence of a given string of characters in another character string.

Arguments to `nlfindstr()` are:

langid The ID number of the desired language.

string1 A pointer to the character buffer to be searched. It can contain single-byte and two-byte characters.

length1 Length (in bytes) of *string1*.

string2 The character buffer for which to search.

length2 Length (in bytes) of *string2*. *length2* must be less than or equal to *length1*.

err The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Invalid *length1* parameter.
4	Invalid *length2* parameter.

charset

A byte buffer containing the character set definition for the language to be used, as returned by *nlinfo*(3X)'s *itemnumber* 12.

RETURN VALUE

offset is a short integer that holds the number of bytes into *string1* where *string2* was found. `nlfindstr()` returns –1 if the string is not found.

WARNINGS

This routine is provided for compatibility with MPE, a proprietary HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described by *hpnls*(5) for HP-UX NLS support.

AUTHOR

`nlfindstr()` was developed by HP.

SEE ALSO

nlinfo(3X), mpnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlfmtcalendar() - format an MPE packed date using a localized format

SYNOPSIS

```
#include <portnls.h>

void nlfmtcalendar(
    unsigned short int date,
    char *outstr,
    short int langid,
    unsigned short int err[2]
);
```

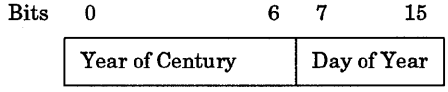
DESCRIPTION

`nlfmtcal()` formats the specified date in the localized custom version of the date format, but with no time information (see *nlfmtclock*(3X)). For example:

FRI, OCT 2, 1987

Arguments to `nlfmtcal()` are used as follows:

date An unsigned short indicating the date in the packed date format:



outstr
A character buffer in which the formatted date is returned. This buffer is 18 bytes long, and padded with blanks if necessary.

langid
A short integer specifying the language whose custom is to be used.

err
The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Invalid date format.

WARNINGS

This routine is provided for compatibility with MPE, a proprietary HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described by *hpnls*(5) for HP-UX NLS support.

AUTHOR

`nlfmtcal()` was developed by HP.

SEE ALSO

calendar(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlfmtclock() - format MPE time of day using localized format

SYNOPSIS

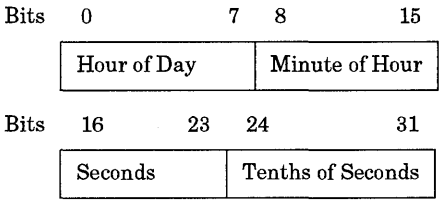
```
#include <portnls.h>

void nlfmtclock(
    unsigned int time,
    char *outstr,
    short int langid,
    unsigned short int err[2]
);
```

DESCRIPTION

`nlfmtclock()` formats the time of day obtained with the clock routine, according to the clock format defined for the specified language.

Arguments to `nlfmtclock()` are used as follows:

time An unsigned int obtained from the clock routine:



outstr
An 8-byte buffer in which the formatted time of day is returned.

langid
A short integer specifying the language whose clock format is to be used.

err
The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Invalid time format.

WARNINGS

This routine is provided for compatibility with MPE, a proprietary HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described by *hpnls*(5) for HP-UX NLS support.

AUTHOR

`nlfmtclock()` was developed by HP.

SEE ALSO

clock(3X), nlconvclock(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlfmtcustdate() - format an MPE packed date using a custom date

SYNOPSIS

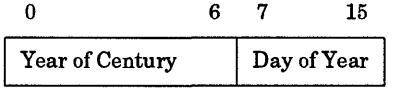
```
#include <portnls.h>

void nlfmtcustdate(
    unsigned short int date,
    char *outstr,
    short int langid,
    unsigned short int err[2]
);
```

DESCRIPTION

`nlfmtcustdate()` converts the packed date format to the language-dependent custom date as specified in the language definition file. A custom date has an abbreviated format such as `10/21/87` or `87.10.21`.

Arguments to `nlfmtcustdate()` are used as follows:

date An unsigned short containing the date in the packed date format:



outstr
A 13-byte buffer in which the formatted date is returned.

langid
A short integer of the language whose custom date specification is to be used for the format.

err
The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Invalid date format.

WARNINGS

This routine is provided for compatibility with MPE, a proprietary HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described by *hpnls*(5) for HP-UX NLS support.

AUTHOR

`nlfmtcustdate()` was developed by HP.

SEE ALSO

calendar(3X), nlconvcustdate(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlfmtdate() - format MPE date and time in a localized format

SYNOPSIS

```
#include <portnls.h>

void nlfmtdate(
    unsigned short int date,
    unsigned long int time,
    char *outstr,
    short int langid,
    unsigned short int err[2]
);
```

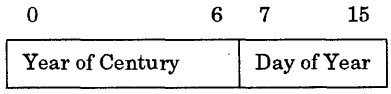
DESCRIPTION

`nlfmtdate()` formats the specified date and time in a localized custom version. For example:

```
SUN, FEB 7, 1988  9:00 AM
```

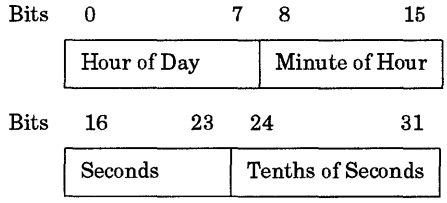
Arguments to `nlfmtdate()` are used as follows:

date An unsigned short indicating the date to be formatted in the packed date format:



time

An unsigned int indicating the time to be formatted. The double word is in the clock format:



outstr

A 28-byte buffer in which the formatted date is returned.

langid

A short containing the language ID indicating the custom to be used.

err

The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Invalid date format.
4	Invalid time format.

WARNINGS

This routine is provided for compatibility with MPE, a proprietary HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described by *hpnls*(5) for HP-UX NLS support.

AUTHOR

`nlfmtdate()` was developed by HP.

SEE ALSO

calendar(3X), clock(3X), nlfmtcal(3X), nlfmtclock(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlfmtlongcal() - format an MPE packed date using a long calendar format

SYNOPSIS

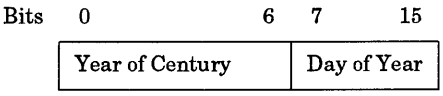
```
#include <portnls.h>

void nlfmtlongcal(
    unsigned short int date,
    char *outstr,
    short int langid,
    unsigned short int err[2]
);
```

DESCRIPTION

`nlfmtlongcal()` formats the supplied date according to the long calendar format. The formatting is done according to the template returned by *nlinfo*(3X), *itemnumber* 30.

Arguments to `nlfmtlongcal()` are used as follows:

date A short integer value containing a date in the packed date format:



outstr
A 36-byte buffer to which the formatted long calendar date is returned, padded with blanks if necessary.

langid
An ID number specifying which language-specific format is to be used.

err
The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Invalid date format.

WARNINGS

This routine is provided for compatibility with MPE, a proprietary HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described by *hpnls*(5) for HP-UX NLS support.

AUTHOR

`nlfmtlongcal()` was developed by HP.

SEE ALSO

calendar(3X), nlfmtcalendar(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlfmtnum() - convert an ASCII number to an MPE language-specific formatted number

SYNOPSIS

```
#include <portnls.h>

void nlfmtnum(
    short int langid,
    const char *instr,
    short int leninstr,
    char *outstr,
    short int *plenoutstr,
    unsigned short int err[2],
    const char *numspec,
    short int fmtmask,
    short int decimals
);
```

DESCRIPTION

`nlfmtnum()` converts a string containing an ASCII number to a language-specific formatted number using the currency name/symbol, decimal separator, and thousands separators defined for the language. The string may contain the n-computer decimal separator (`.`), thousands separator (`,`), and a dollar sign (`$`).

This routine operates in two modes: substitution mode and formatting mode. The substitution mode (if *fmtmask* is zero) substitutes the native equivalent for `.` and `,` and, for `arabic`, the alternate set of digits for ASCII digits. The input is not validated as a number, and can contain several individual numbers. No justification takes place, and the output is left-truncated if *outstr* is shorter than *instr* (for example, `1,234.56` becomes `234,56`).

If *fmtmask* is not zero, the formatting mode formats the input according to *fmtmask* in addition to performing the substitution. In this mode the input is validated as a number and only ASCII digits and `-`, `+`, `$`, `.`, and `,` are allowed. Only one sign and one `$` is allowed and they must be the first character(s) in *instr*. Even if insertion (of thousands separators, etc.) is specified in *fmtmask*, thousands separators and a decimal separator are still valid characters in the input. In this case they are substituted. If no justification is specified, the output is right-justified with the same number of trailing spaces as the input. Note that for languages written right-to-left, trailing spaces in the input are preserved as leading spaces in the output. If the output is truncated, it is left-truncated (for example, `1,234.56` becomes `.234,56`).

Arguments to `nlfmtnum()` are used as follows:

langid — A language ID number specifying which language's formatting specifications to use for the formatting.

instr — A byte array containing the n-computer formatted ASCII number to be converted, for example, 123,456.78. Leading and trailing spaces are allowed.

leninstr — Length, in bytes, of *instr*.

outstr — A byte buffer where the language-specific formatted number is returned. The decimal separator, thousands separator, and currency symbol/name are replaced according to the language definition, if present or inserted, or if specified by *fmtmask*. *outstr* can reference the same address as *instr*.

plenoutstr — Length, in bytes, of *outstr*. After a successful call, if specified by *fmtmask* (the two bits starting with bit 12 (from highest to lowest) are equal to 3), *plenoutstr* returns the actual length, in bytes, of the formatted number.

err — The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured,
3	Invalid length specified (*leninstr* or **plenoutstr*).
4	Invalid number specified (*instr*).

5	Invalid decimal point in number specified (*instr*).
6	Invalid thousand separators in number specified (*instr*).
7	Truncation has occurred (*outstr* is left partially formatted).
8	Invalid *numspec* parameter.
9	Invalid *fmtmask* parameter.
10	Invalid *decimals* parameter.

numspec
A byte array, as returned from **nlnumspec()**, containing formatting specifications for the specified language (currency symbol/name, decimal separator, etc.). If this parameter is not null, *langid* is ignored, and performance is improved. (See *nlnumspec*(3X)).

fmtmask
A short integer value specifying any formatting to be done on the input. The default value is zero, which means a simple substitution.

Value	Description
NULL	Do not insert thousands separators. Do not insert decimal separator. No justification of the output.
M_INSTHOU	Insert thousands separators.
M_INSDEC	Insert decimal separator.
M_CURRENCY	Insert currency name/symbol.
M_LEFTJUST	The output is left-justified.
M_RIGHTJUST	Right-justify the output.
M_RETLENGTH	Left-justify the output and return the actual length of the formatted number in *plenoutstr*

decimals
An integer specifying where to insert the decimal separator. The value is ignored if *fmtmask* and **M_INSDEC** are zero, or a decimal separator is present in the number.

WARNINGS
This routine is provided for compatibility with MPE, a proprietary HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described by *hpnls*(5) for HP-UX NLS support.

AUTHOR
nlfmtnum() was developed by HP.

SEE ALSO
nlconvnum(3X), portnls(5).

EXTERNAL INFLUENCES
International Code Set Support
Single- and multi-byte character code sets are supported.

NAME

nlgetlang() - return the current user, data, or system default language

SYNOPSIS

```
#include <portnls.h>

short int nlgetlang(short int function, unsigned short int err[2]);
```

DESCRIPTION

`nlgetlang()` looks for a LANG string in the user's environment. If it finds it, it returns the corresponding integer as described in *lang*(5). Otherwise, or if the value of *function* is not valid, it returns *0* and sets the *err* parameter.

Arguments to `nlgetlang()` are used as follows:

function A short integer that specifies which language is returned.

Value	Description
1	User language
2	Data language
3	System default language

err

The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
1	Native Language Support file(s) not found
2	Specified language not configured
3	Invalid *function* value
4	No language specified for `nlgetlang()` to access

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUE

`nlgetlang()` returns the language ID as a short integer. In case of error, zero is returned.

WARNINGS

This routine is provided for compatibility with MPE, a proprietary HP operating system. See *portnls*(5) for more information on the use of this function. Use the Native Language Support routines for C programmers described by *hpnls*(5) for HP-UX NLS support.

AUTHOR

`nlgetlang()` was developed by HP.

SEE ALSO

getenv(3C), currlangid(3C), portnls(5).

NAME

nlinfo() - return MPE language-dependent information

SYNOPSIS

```
#include <portnls.h>

void nlinfo(
    short int itemnumber,
    int *itemvalue,
    short int *langid,
    unsigned short int err[2]
);
```

DESCRIPTION

`nlinfo()` returns such information as the format of the date, the radix character, the direction of the language, etc.

The *itemnumber* indicates the type of information the user has requested. The data is passed back in *itemvalue*.

The arguments to `nlinfo()` are used as follows:

itemnumber A short integer of the item desired. This number specifies which item value is to be returned. See below for a list of item numbers.

itemvalue A pointer to an integer that contains the value of the item specified by the corresponding item number. The data type of the item value depends on the item itself.

langid A pointer to a short integer containing the language ID or, for *itemnumber* 22, the location in which the language ID is returned.

err The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
1	Native Language Support file(s) not found
2	Specified language is not configured.
3	Specified character set is not configured.
10	*itemnumber* is out of range.

Item numbers

The following is a list of the currently defined item numbers and the information returned.

itemnumber	Description
1	An 18-byte buffer in which the calendar format is returned.
2	A 13-byte buffer in which the custom date format is returned.
3	An 8-byte buffer in which the clock specification is returned.
4	A 48-byte buffer in which the month denotation abbreviation table is returned. The abbreviation of each month is 4 bytes long (with blank padding if necessary). The first 4 bytes are the abbreviation for January.
5	A 144-byte array in which the month denotation table is returned. Each month denotation is 12 bytes long. The table starts with January.
6	A 21-byte array in which the day of the week denotation abbreviation table is returned. Each weekday abbreviation is three bytes long. The first three bytes are the abbreviation for Sunday.
7	An 84-byte array in which the day of the week denotation table is returned. Each weekday denotation is 12 bytes long. The table starts with Sunday.
8	A 12-byte array in which the YES/NO responses are returned. The first 6 bytes contain the (upshifted) "YES" response; the second 6 bytes contain the (upshifted) "NO" response.

9 A 2-byte array in which the symbols for decimal point and thousands indicator are returned. The first byte contains the decimal point, the second contains the thousands indicator.

10 A 6-byte array in which the currency signs are returned. The first byte contains the currency sign used in the business formats, the second byte is either a numeric zero, which indicates that the currency symbol precedes the value, or a one, which indicates that a symbol follows the value. The next 4 bytes contain the fully qualified currency sign.

11 An array in which the collating sequence table is returned. To determine the size of this array, the length must be determined by a call to `nlinfo()` with *itemnumber* 27.

12 A 256-byte array in which the character set definition is returned. Each byte has numeric identification of the character type:

0	Numeric character
1	Alphabetic lowercase character
2	Alphabetic uppercase character
3	Undefined graphic character
4	Special character
5	Control code
6	First byte of a two-byte character

15
A 256-byte array in which the upshift table is returned.

16
A 256-byte array in which the downshift table is returned.

17
An array of unsigned shorts in which the language numbers of all configured languages are returned. The first element of this array contains the number of configured languages. The second word contains the language number of the first configured language, etc. The system default language is returned (the *langid* parameter, if specified, is insignificant).

18
A short int in which true (`-1`) is returned if the specified language is supported (configured) on the system. Otherwise, false (**0**) is returned.

21
A 16-byte array in which the (uppercase) name of the specified language is returned. If the name contains less than 16 bytes, it is padded with blanks.

22
The *itemvalue* contains a byte buffer containing a language name or language number (ASCII digits) terminated by a blank. The array must contain less than or equal to 16 bytes. The *langid* (third) parameter is assigned the associated language ID number.

26
A short integer in which the class number of the specified language is returned.

27
An integer in which the length (in two-byte units) of the collating sequence table corresponding to the specified language is returned.

28
A short integer in which the length (in two-byte units) of the national dependent information table is returned. If no national table exists for the specified language, an error is returned.

29
A byte buffer in which the national-dependent information table is returned. To determine the size of this array, the length must be obtained via a prior call to `nlinfo()` with *itemnumber* 28.

30
A 36-byte array in which the long calendar format is returned. It may contain arbitrary text as well as the following descriptors:

D	1 through 3 of these are to be replaced by that many bytes from the day abbreviation.
W	1 through 12 of these are to be replaced by that many bytes from the day of the week.
M	1 through 4 of these are to be replaced by that many bytes from the month abbreviation.
O	1 through 12 of these are to be replaced by that many bytes from the month of the year.
mm	Numeric month of the year.
yy	Numeric year of the century.
yyyy	Numeric year of the century.
Nyy	National year.

In addition, a special literal character ~ (tilde) can be used to indicate that the following character should be taken literally in the format, even if it is one of the special characters above.

For example, a format could be:

```
"WWWWWWWWW, OOOOOOOOO  dd, A.~D. yyyy  "
```

This format in n-computer would result in the following:

```
"WEDNESDAY, NOVEMBER  21, A.D. 1984  "
```

31
A 16-byte array in which the currency name is returned.

32
An 8-byte array, containing information about an Alternate set of digits (currently only used by `arabic`).

Byte	Description
0-1	Alternate digit indicator 0 - No Alternate digits defined 1 - Alternate digits defined
2	The Alternate digit 0
3	The Alternate digit 9
4	The + used with Alternate digits
5	The - used with Alternate digits
6	The decimal separator used with Alternate digits
7	The thousands separator used with Alternate digits

33
A 4-byte array, containing information about the direction of the language.

Byte	Description
0 1	Language direction 0 - Direction is "left-to-right" 1 - Direction is "right-to-left"
2	The "right-to-left" space
3	Undefined

34
An unsigned short that returns the data ordering of the language.

0	Keyboard order
1	Left-to-Right screen order
2	Right-to-Left screen order

35
An unsigned short that returns the size of the character used by the language.

0	One-byte characters (8 bits)
1	Two-byte characters (16 bits)

WARNINGS

This routine is provided for compatibility with MPE, a proprietary HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described by *hpnls*(5) for HP-UX NLS support.

AUTHOR

`nlinfo()` was developed by HP.

SEE ALSO

hpnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlist() - get entries from name list

SYNOPSIS

```
#include <nlist.h>

int nlist(const char *file_name, struct nlist *nl);
```

Remarks

The use of symbol table type and value information is inherently non-portable. Use of `nlist()` should reduce the effort required to port a program that uses such information, but complete portability across all HP-UX implementations cannot be expected.

DESCRIPTION

`nlist()` examines the name list in the executable file whose name is pointed to by *file_name*, and selectively extracts a list of values and puts them in the array of `nlist()` structures pointed to by *nl*. The array of `nlist()` structures initially contains only the names of variables. Once `nlist()` has been called, the variable names are augmented with types and values. The list is terminated by a null name, which consists of a null string in the variable-name position of the structure. The name list of the file is searched for each variable name. If the name is found, type and value information from the file is inserted into the name list structure. If the name is not found, type and value fields are set to zero. The structure `nlist` is defined in the include file <`nlist.h`>. See *a.out*(4) and *nlist*(4) for further description of the symbol table structure.

The file must have the organization and symbol table described for an `a.out` file in *a.out*(4). The information is extracted from the symbol table used by the loader, *ld*(1).

On machines that have such a file, this subroutine is useful for examining the system name list kept in file `/hp-ux`. In this way programs can obtain system addresses that are up to date.

RETURN VALUE

All nlist structure fields are set to 0 if the file cannot be found or if it is not a valid object file containing a linker symbol table.

`nlist()` returns -1 upon error; otherwise it returns 0.

WARNINGS

The <`nlist.h`> header file is automatically included by <`a.out.h`> for compatibility. However, including <`a.out.h`> is discouraged if the only information needed from <`a.out.h`> is for use by `nlist()`. If <`a.out.h`> is included, the line `#undef n_name` may need to follow it.

SEE ALSO

a.out(4), nlist(4).

STANDARDS CONFORMANCE

nlist: SVID2

NAME

nljudge() - judge whether a character is a one-byte or multi-byte Asian character using MPE character definition table

SYNOPSIS

```
#include <portnls.h>

short int nljudge(
    short int langid,
    const char *instr,
    short int length,
    char *judgeflag,
    unsigned short int err[2],
    const char *charset
);
```

DESCRIPTION

nljudge() judges whether or not a character is a one-byte or multi-byte Asian character. If it is a multi-byte character, *judgeflag* is set to **1** or **2**. If it is a one-byte character, *judgeflag* is set to **0**.

Any language number can be specified as the *langid* parameter. However, if the language specified uses only one-byte characters (see *nlinfo*(3X)'s *itemnumber* 35), the *judgeflag* returns all zeroes.

Arguments to **nljudge()** are used as follows:

langid The ID number for the desired language.

instr The character buffer to be judged.

length A short integer value specifying the number of bytes in *instr*.

judgeflag A pointer to a char whose value is set to:

0	One-byte character
1	First byte of a two-byte character
2	Second byte of a two-byte character
3	Invalid two-byte character

err

The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Invalid string length.
7	Invalid characters found in *instr*.

charset

A character buffer containing the character set definition for the language to be used, as returned by *nlinfo*(3X)'s *itemnumber* 12. If it doesn't point to a null address, the *langid* parameter is ignored, and this routine is more efficient.

RETURN VALUE

nljudge() returns the number of multi-byte Asian characters that could be used to check if a string of character contains any Asian characters.

WARNINGS

This routine is provided for compatibility with MPE, a proprietary HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described by *hpnls*(5) for HP-UX NLS support.

AUTHOR

nljudge() was developed by HP.

SEE ALSO

nlinfo(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlkeycompare() - determine if a character array (key1) is almost equal to another (key2) using the MPE language-dependent collation table

SYNOPSIS

```
#include <portnls.h>

void nlkeycompare(
    const char *key1,
    short int length1,
    const char *key2,
    short int length2,
    short int *presult,
    short int langid,
    unsigned short err[2],
    const unsigned short *collseq
);
```

DESCRIPTION

`nlkeycompare()` determines if a character array (key1) is almost equal to another character array (key2). Two character arrays are considered almost equal when they differ only in case or accent priorities. For example, the arrays `ABC` and `aBc` are almost equal in English.

`nlkeycompare()` determines if a given character array can be collated before or after another character array of a different length. For example, `nlkeycompare()` examines the records in a file sorted in a given language and determines if the character array `key1` can be found later on in the file as the leading substring of the sort key, if the value of the last record read is `key2.`

Arguments to `nlkeycompare()` are used as follows:

key1 A byte array being compared to *key2*.

length1 The length in bytes of *key1*. *length1* must be less than *length2*.

key2 A byte array containing a character array to which to compare *key1*.

length2 The length in bytes of *key2*. *length2* must be greater than *length1*.

presult A pointer to a short integer variable in which to return the result of the comparison.

0	The retrieved key2 matches the key1.
1	The retrieved key2 does not match the key1. It is different only in case or accent priority.
2	The retrieved key2 is less than the key1 (its collating order is before the desired one).
3	The retrieved key2 is greater than the key1 (it collates after the desired key).

langid

The language ID number indicating the collating sequence to be used for the compare.

err

The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Invalid collating table entry.
4	Invalid length parameter.
7	*length1* is greater than *length2*.

collseq

An array containing the collating sequence table as returned by *nlinfo*(3X)'s *itemnumber* 11.

WARNINGS

This routine is provided for compatibility with MPE, a proprietary HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers

described by *hpnls*(5) for HP-UX NLS support.

AUTHOR

`nlkeycompare()` was developed by HP.

SEE ALSO

nlcollate(3X), nlinfo(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlnumspec() - return information needed by MPE routines for formatting and converting numbers

SYNOPSIS

```
#include <portnls.h>

void nlnumspec(short int langid, char *numspec, unsigned short err[2]);
```

DESCRIPTION

`nlnumspec()` returns the information needed for formatting and converting numbers. It combines several calls to `nlinfo()` in order to simplify the use of native language formatting. By calling `nlnumspec()` once, and passing the obtained information to `nlfmtnum()` and `nlconvnum()`, implicit calls to `nlnumspec()` from `nlfmtnum()` and `nlconvnum()` are avoided and performance is improved.

`nlnumspec()` combines the functions of *nlinfo*(3X)'s *itemnumber*s 9, 10, 31, 32, and 33. The information is formatted where needed. For example, any spaces in the currency symbol/name are included. The currency symbol/name is the shortest non-blank descriptor, as returned from *nlinfo*(3X) *itemnumber*s 10 and 31.

`nlnumspec()` does not, apart from the mentioned formatting, provide any information not obtainable with `nlinfo()`, but is included for the convenience of the user. For efficiency, the user of this routine calls it once, saves the result, and then calls `nlfmtnum()` and/or `nlconvnum()` multiple times.

Arguments to `nlnumspec()` are used as follows:

langid The ID number of the desired language.

numspec A character buffer of at least 60 bytes in which the following information is returned:

Byte	Description
00-01	Language ID number.
02-03	Alternate Digit Indicator: 0 - No Alternate digits exist. 1 - Alternate digits exist.
04-05	Language Direction Indicator. 0 - The Language is "left-to-right". 1 - The Language is "right-to-left".
06-07	The Alternate digit range ("0", "9").
08	Decimal separator (ASCII-digits).
09	Decimal separator (Alternate-digits).
10	Thousands separator (ASCII-digits).
11	Thousands separator (Alternate-digits).
12	"+" Alternate-digits.
13	"–" Alternate-digits.
14	"Right-to-left" space.
15	Reserved.
16-17	Currency place. 0 - Currency symbol precedes the number. 1 - Currency symbol follows the number. 2 - Currency symbol replaces the decimal separator.
18-19	Length of Currency symbol (including any spaces).
20-37	Currency symbol (including any spaces).
38-39	Data ordering of the language.
40-41	Size of character used by the language.
42-59	Reserved.

err

The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.

WARNINGS

This routine is provided for compatibility with MPE, a proprietary HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described by *hpnls*(5) for HP-UX NLS support.

AUTHOR

`nlnumspec()` was developed by HP.

SEE ALSO

nlinfo(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlrepchar() - replace non-displayable characters of a string using the MPE character set table

SYNOPSIS

```
#include <portnls.h>

void nlrepchar(
    const char *instr,
    char *outstr,
    short int length,
    char repchar,
    short int langid,
    unsigned short int err[2],
    const char *charset
);
```

DESCRIPTION

`nlrepchar()` replaces all non-displayable characters in the input character buffer with the replacement character. Non-displayable characters are those of types 3 and 5, as returned by *nlinfo*(3X), *itemnumber* 12. Native language characters of the supported character set are not replaced.

Arguments to `nlrepchar()` are used as follows:

instr A character buffer in which the non-displayable characters must be replaced.

outstr A character buffer to which the replaced character string is returned.

length A short integer specifying the length (in bytes) of *instr*.

repchar A byte specifying the replacement character to be used.

langid A short integer value specifying the language ID number of the language that determines the character set to be used.

err The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Invalid replacement character.
4	Invalid length parameter.
8	The value of *outstr* would overwrite *instr*.
10	Invalid Asian character.

charset

Contains the character set definition for the language to be used, as returned in *nlinfo*(3X)'s *itemnumber* 12. If this parameter is supplied (i.e., not NULL), *langid* is ignored and this routine is much more efficient.

WARNINGS

This routine is provided for compatibility with MPE, a proprietary HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described by *hpnls*(5) for HP-UX NLS support.

AUTHOR

`nlrepchar()` was developed by HP.

SEE ALSO

nlinfo(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlscanmove() - move, scan and case-shift character strings using the MPE character set definition table

SYNOPSIS

```
#include <portnls.h>

short int nlscanmove(
    const char *instr,
    char *outstr,
    short int flags,
    int length,
    short int langid,
    unsigned short int err[2],
    const char *pcharset,
    const char *pshift
);
```

DESCRIPTION

`nlscanmove()` moves, scans and case-shifts character strings.

Arguments to `nlscanmove()` are used as follows:

instr A character buffer that acts as the source string of the scan or move functions.

outstr A character buffer that acts as the target. Note that if *outstr* is equal to *instr*, this routine will act as scan. Otherwise, a move will be performed; see *err* below.

flags A flag defining the options for the routine invocation. This parameter defines the end condition for the scan or move.

Value	Description
`M_L`	Select lowercase alphabetic characters.
`M_U`	Select uppercase alphabetic characters.
`M_N`	Select numeric characters.
`M_S`	Select special characters.
`M_WU`	By default `nlscanmove()` scans or moves characters while the character currently being scanned is one of those selected (i.e. upper, lower, numeric, special). If `M_WU` is used, `nlscanmove()` scans or moves characters until the character currently being scanned is one of those selected.
`M_US`	Shift scanned or moved characters to the uppercase.
`M_DS`	Shift scanned or moved characters to the lowercase.
`M_OB`	Select one-byte characters.
`M_TB`	Select two-byte (Asian) characters.
`M_OB or M_TB`	Select both one- and two-byte characters.

length
A short integer indicating the maximum number of valid bytes to be acted upon during the indicated option.

langid
A short integer containing the language ID number which implies the both the character set definitions of character attributes and the language specific shift.

err
The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Overlapping strings, *instr* overwrites *outstr*.
4	Invalid length parameter.
7	The reserved part of *flags* is not zero.
8	Both upshift and downshift request.
9	Invalid table element.
10	Invalid Asian character.

pcharset
A pointer to a character buffer containing the character set definition for the language to be used, as returned *nlinfo*(3X)'s *itemnumber* 12. If not zero, the *langid* parameter is ignored, and this routine is much more efficient. This parameter is required for calls in which bits (12:4) of *flags* is neither `0` nor `15`.

pshift
A pointer to a character buffer containing shift information for a desired upshift or downshift (e.g., as returned in *nlinfo*(3X)'s *itemnumber* 15 or 16). This parameter is used when bits (9:2) of *flags* is not `0`.

RETURN VALUE
A short containing the number of bytes acted upon in the scan or move operation.

WARNINGS
This routine is provided for compatibility with MPE, a proprietary HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described by *hpnls*(5) for HP-UX NLS support.

AUTHOR
`nlscanmove()` was developed by HP.

SEE ALSO
nlinfo(3X), portnls(5).

EXTERNAL INFLUENCES
International Code Set Support
Single- and multi-byte character code sets are supported.

NAME

nlsubstr() - extract substring of a string using the MPE character set definition table

SYNOPSIS

```
#include <portnls.h>

void nlsubstr(
    const char *instring,
    short int inlength,
    char *outstring,
    short *poutlength,
    short int start,
    short int movelength,
    short int langid,
    short int flags,
    unsigned short int err[2],
    const unsigned short int *charset
);
```

DESCRIPTION

nlsubstr() extracts a substring from *instring* and places the result in *outstring*.

Arguments to **nlsubstr()** are used as follows:

instring The byte buffer from which the substring is extracted. The string can contain both one-byte and two-byte (Asian) characters.

inlength Length, in bytes, of *instring*

outstring Where the sub-string is placed.

poutlength Length, in bytes, of *outstring*. After a successful call, the variable to which *poutlength* points contains the actual length of the sub-string moved to *outstring*.

start The offset into *instring* where the sub-string starts. A value of zero is the beginning point.

movelength Length, in bytes, of the sub-string.

langid The ID number of the desired language.

flags This flag word is used primarily with Asian languages. It is meaningless with one-byte oriented languages. *flags* is used to indicate the treatment of the case when the first byte of the sub-string is the second byte of a two-byte Asian character and in the case where the last byte in the sub-string is the first byte of a two-byte Asian character.

Selection of **nlsubstr()**'s behavior if the first character is the second byte of an Asian character:

Value	Description
F_RETURNERR	Return an error condition.
F_SPP1	Start from *start*+1.
F_SPM1	Start from *start*-1.
F_SPBL	Start from *start*, but replace the character with a blank in *outstring*.
F_SP	Start from *start*, regardless of the value of the first character.

Selection of **nlsubstr()**'s behavior if the last character is the first byte of an Asian character:

Value	Description
F_LMP1	Move until *movelength*+1 is reached.
F_LMM1	Move until *movelength*-1 is reached.

`F_LMBL` Move until *movelength* is reached, but replace the character with a blank in *outstring*.

`F_LM` Move until *movelength* is reached, regardless of the value of the last byte.

err

The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
7	Invalid *inlength*.
8	Invalid *start*.
9	Invalid *movelength*.
11	Invalid value in *flags* bits (8:4).
12	Invalid value for *flags* bits (12:4).
13	The start position is the second byte of an Asian character, or an underflow condition occurred because of *flags*.
14	The end position is the first byte of an Asian character, or an overflow condition occurred because of *flags*.

charset

An array containing the character set definition for the language to be used, as returned by *nlinfo*(3X)'s *itemnumber* 12.

WARNINGS

This routine is provided for compatibility with MPE, a proprietary HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described by *hpnls*(5) for HP-UX NLS support.

AUTHOR

`nlsubstr()` was developed by HP.

SEE ALSO

nlinfo(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nlswitchbuf() - convert a string of characters between phonetic order and screen order using the MPE character set definition table

SYNOPSIS

```
#include <portnls.h>

void nlswitchbuf(
    short int langid,
    const char *instr,
    char *outstr,
    short int length,
    unsigned short int lefttoright,
    unsigned short int err[2]
);
```

DESCRIPTION

`nlswitchbuf()` is useful for handling data from languages written from right-to-left (e.g., Middle Eastern languages). It is used by a program to convert a buffer that is in phonetic order (i.e., the order in which the characters would be typed at a terminal or spoken by a person) to screen order (i.e., the order in which the characters are displayed on a terminal screen or piece of paper), or vice-versa. Screen order is defined as right-to-left if the primary mode of the terminal or printer is from right-to-left (as when it is used principally for entering or displaying data from a right-to-left language). Otherwise, screen order is defined as left-to-right.

Phonetic order and screen order are, in general, not the same if USASCII text is mixed with that from a right-to-left language. The relationship between phonetic order and screen order is further complicated by the Hindi digits in Arabic, which play a third role intermediate between ASCII characters and characters of the right-to-left language.

Note that this is a somewhat special-purpose native language support routine. `nlswitchbuf()` is useful only for languages that are written from right-to-left, and which may occasionally mix left-to-right text (e.g., English) with right-to-left. Nonetheless, it can be used by a general-purpose (not specifically for handling right-to-left data) program. Such a program calls `nlswitchbuf()` to convert data from phonetic order to screen order and back again (for example, an editor that wants to track cursor movement on a terminal against a buffer of text in memory needs to do this). If the data is not that of a right-to-left language, this routine simply returns the same text unchanged, since for all other languages phonetic order and screen order are the same.

Arguments to `nlswitchbuf()` are:

langid The ID number for the desired language.

instr The character buffer in phonetic order to be converted to screen order.

outstr The buffer in which the result of the conversion to screen order is returned. *outstr* and *instr* can reference the same address.

length The length, in characters, of the buffer to be converted.

lefttoright An unsigned short integer that specifies whether the implied primary mode of the data (i.e., the way it would be displayed on a terminal) is left-to-right (TRUE) or right-to-left (FALSE). This determines what the opposite language is and, therefore, strings of which characters get switched.

err The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Invalid string length.

WARNINGS

This routine is provided for compatibility with MPE, a proprietary HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described by *hpnls*(5) for HP-UX NLS support.

AUTHOR

`nlswitchbuf()` was developed by HP.

SEE ALSO

nlinfo(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

nltranslate() - translate ASCII strings to EBCDIC using MPE conversion table

SYNOPSIS

```
#include <portnls.h>

void nltranslate(
    short int code,
    const char *instr,
    char *outstr,
    short int length,
    short int langid,
    unsigned short int err[2],
    const char *table
);
```

DESCRIPTION

`nltranslate()` translates a string of bytes from EBCDIC to ASCII or ASCII to EBCDIC, using the appropriate native language table.

Arguments to `nltranslate()` are used as follows:

code Specifies type of conversion:

Value	Meaning
1	Convert EBCDIC to ASCII.
2	Convert ASCII to EBCDIC.

instr
Byte buffer containing the input string to be translated.

outstr
Byte buffer where the translated string is to be returned. *instr* and *outstr* can specify the same array.

length
A short integer specifying the number of bytes of *instr* to be translated.

langid
A short integer containing the ID number of the language whose translation tables are to be used.

err
The first element of this array contains the error number. The second element is always zero. If the call is successful, both elements contain zero.

Error #	Meaning
2	Specified language is not configured.
3	Invalid code specified.
4	Invalid length parameter.

table
A 256-byte array that holds a translation table. Each byte contains the translation of the byte whose value is its index. This table is provided by the user.

WARNINGS

This routine is provided for compatibility with MPE, a proprietary HP operating system. See *portnls*(5) for more information on the use of this routine. Use the Native Language Support routines for C programmers described by *hpnls*(5) for HP-UX NLS support.

AUTHOR

`nltranslate()` was developed by HP.

SEE ALSO

nlinfo(3X), portnls(5).

EXTERNAL INFLUENCES

International Code Set Support

Single- and multi-byte character code sets are supported.

NAME

optoverhead( ) - return number of bytes needed by a NetIPC option

SYNOPSIS

```
#include <sys/ns_ipc.h>

int optoverhead(short eventualentries, short *result);
```

DESCRIPTION

`optoverhead` returns the number of bytes needed by the *opt* parameter, excluding the data area.

PARAMETERS

eventualentries (input parameter) The number of option entries that will be placed in the *opt* parameter.

result (output parameter) The result code returned. See "Diagnostics" below for more information.

RETURN VALUE

Upon successful completion, `optoverhead()` returns a 16-bit integer value indicating the number of bytes requires for the *opt* parameter, not including the data portion of the parameter.

ERRORS

[NSR_NO_ERROR] The call was successful.

[NSR_OPT_TOTAL] The *num_entries* parameter is negative.

AUTHOR

optoverhead was developed by HP.

SEE ALSO

ipcconnect(2), ipccontrol(2), ipccreate(2), ipcdest(2), ipcgetnodename(2), ipclookup(2), ipcname(2), ipcnamerase(2), ipcrecv(2), ipcrecvcn(2), ipcselect(2), ipcsend(2), ipcsetnodename(2), ipcshutdown(2), addopt(3N), initopt(3N), ipcerrmsg(3N), readopt(3N).

NAME

perror(), strerror(), errno, sys_errlist, sys_nerr - system error messages

SYNOPSIS

```
#include <errno.h>

void perror(const char *s);

char *strerror(int errnum);

extern int errno;

extern char *sys_errlist[];

extern int sys_nerr;
```

DESCRIPTION

`perror()` writes a language-dependent message to the standard error output, describing the last error encountered during a call to a system or library function. The argument string *s* is printed first, followed by a colon, a blank, the message, and a new-line. To be most useful, the argument string should include the name of the program that incurred the error. The error number is taken from the external variable `errno`, which is set when errors occur but not cleared when non-erroneous calls are made. The contents of the message is identical to those returned by the `strerror()` function with `errno` as the argument. If given a NULL string, the `perror()` function prints only the message and a new-line.

To simplify variant formatting of messages, the `strerror()` function and the `sys_errlist` array of message strings are provided. The `strerror()` function maps the error number in *errnum* to a language-dependent error message string and returns a pointer to the string. The message string is returned without a new-line. `errno` can be used as an index into `sys_errlist` to get an untranslated message string without the new-line. `sys_nerr` is the largest message number provided for in the table; it should be checked because new error codes might be added to the system before they are added to the table. `strerror()` must be used to retrieve messages when translations are desired.

EXTERNAL INFLUENCES

Environment Variables

The language of the message returned by `strerror()` and printed by `perror()` is specified by the **LANG** environment variable. If the language-dependent message is not available, or if **LANG** is not set or is set to the empty string, the default version of the message associated with the "C" language (see *lang*(5)) is used.

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUE

`perror()` returns no value.

If the *errnum* message number is valid, `strerror()` returns a pointer to a language-dependent message string. The array pointed to should not be modified by the program, and might be overwritten by a subsequent call to the function. If a valid *errnum* message number does not have a corresponding language-dependent message, `strerror()` uses *errnum* as an index into `sys_errlist` to get the message string. If the *errnum* message number is invalid, `strerror()` returns a pointer to a NULL string.

SEE ALSO

errno(2), lang(5), environ(5).

STANDARDS CONFORMANCE

`perror()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`strerror()`: AES, XPG3, XPG4, ANSI C

`sys_errlist()`: SVID2, XPG2

`sys_nerr()`: SVID2, XPG2

NAME

pfm_$cleanup - establish a cleanup handler

SYNOPSIS (C)

C Syntax

```
#include <idl/c/base.h>
#include <ppfm.h>

status_$t pfm_$cleanup(
      pfm_$cleanup_rec *cleanup_record)
```

Pascal Syntax

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/ppfm.ins.pas';

function pfm_$cleanup(
      out cleanup_record: pfm_$cleanup_rec): status_$t;
```

Remarks

To view this manual entry via the *man*(1) command, use the function name shown above without the "$" character.

DESCRIPTION

`pfm_$cleanup()` establishes a cleanup handler that is executed when a fault occurs. A cleanup handler is a piece of code executed before a program exits when a signal is received by the process. The cleanup handler begins where `pfm_$cleanup()` is called; the `pfm_$cleanup()` call registers an entry point with the system where program execution resumes when a fault occurs. When a fault occurs, execution resumes after the most recent call to `pfm_$cleanup()`.

There can be more than one cleanup handler in a program. Multiple cleanup handlers are executed consecutively on a last-in/first-out basis, starting with the most recently established cleanup handler and ending with the first cleanup handler.

On Apollo systems, a default cleanup handler is established at program invocation. The default cleanup handler is always called last, just before a program exits, and releases any system resources still held before returning control to the process that invoked the program.

On other systems, there is no default cleanup handler.

When called to establish a cleanup handler, `pfm_$cleanup()` returns the status `pfm_$cleanup_set` to indicate that the cleanup handler was successfully established. When the cleanup handler is entered in response to a fault signal, `pfm_$cleanup()` effectively returns the value of the fault that triggered the cleanup handler.

See the reference description of `pfm_$init()` for a list of the C signals that the PFM package intercepts.

cleanup_record Is a record of the context when `pfm_$cleanup()` is called. A program should treat this as an opaque data structure and not try to alter or copy its contents. It is needed by `pfm_$rls_cleanup()` and `pfm_$reset_cleanup()` to restore the context of the calling process at the cleanup handler entry point.

NOTE

The `pfm_$cleanup()` call implicitly performs a `pfm_$inhibit()`. Cleanup handler code hence runs with asynchronous faults inhibited. When `pfm_$cleanup()` returns something other than `pfm_$cleanup_set()`, indicating that a fault has occurred, there are four possible ways to leave the cleanup code:

- The program can call `pfm_$signal()` to start the next cleanup handler with a fault signal you specify.
- The program can call `pgm_$exit()` to start the next cleanup handler with a status of *status_$ok*.
- The program can continue with the code following the cleanup handler. It should generally call `pfm_$enable()` to re-enable asynchronous faults. Execution continues from the end of the cleanup handler code; it does not resume where the fault signal was received.

- The program can re-establish the cleanup handler by calling `pfm_$reset_cleanup()` (which implicitly performs a `pfm_$enable()`) before proceeding.

SEE ALSO

pfm_$init(3), pfm_$signal(3).

NAME

pfm_$enable - enable asynchronous faults

SYNOPSIS

C Syntax

```
#include <idl/c/base.h>
#include <ppfm.h>

void pfm_$enable(void)
```

Pascal Syntax

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pfm.ins.pas';

procedure pfm_$enable;
```

Remarks

To view this manual entry via the *man*(1) command, use the function name shown above without the "$" character.

DESCRIPTION

`pfm_$enable()` enables asynchronous faults after they have been inhibited by a call to `pfm_$inhibit()`; `pfm_$enable()` causes the operating system to pass asynchronous faults on to the calling process.

While faults are inhibited, the operating system holds at most one asynchronous fault. Consequently, when `pfm_$enable()` returns, there can be at most one fault waiting on the process. If more than one fault was received between calls to `pfm_$inhibit()` and `pfm_$enable()`, the process receives the first asynchronous fault received while faults were inhibited.

SEE ALSO

pfm_$enable_faults(3), pfm_$inhibit(3).

NAME

pfm_$enable_faults - enable asynchronous faults

SYNOPSIS

C Syntax

```
#include <idl/c/base.h>
#include <ppfm.h>

void pfm_$enable_faults(void)
```

Pascal Syntax

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pfm.ins.pas';

procedure pfm_$enable_faults;
```

Remarks

To view this manual entry via the *man*(1) command, use the function name shown above without the "$" character.

DESCRIPTION

The `pfm_$enable_faults()` call enables asynchronous faults after they have been inhibited by a call to `pfm_$inhibit_faults()`; `pfm_$enable_faults()` causes the operating system to pass asynchronous faults on to the calling process.

While faults are inhibited, the operating system holds at most one asynchronous fault. Consequently, when `pfm_$enable_faults()` returns, there can be at most one fault waiting on the process. If more than one fault was received between calls to `pfm_$inhibit_faults()` and `pfm_$enable_faults()`, the process receives the first asynchronous fault received while faults were inhibited.

SEE ALSO

pfm_$enable(3), pfm_$inhibit_faults(3).

NAME

pfm_$inhibit - inhibit asynchronous faults

SYNOPSIS (C)

```
#include <idl/c/base.h>
#include <ppfm.h>

void pfm_$inhibit(void);
```

SYNOPSIS (PASCAL)

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pfm.ins.pas';

procedure pfm_$inhibit;
```

Remarks

To view this manual entry via the *man*(1) command, use the function name shown above without the "$" character.

DESCRIPTION

`pfm_$inhibit()` prevents asynchronous faults from being passed to the calling process. While faults are inhibited, the operating system holds at most one asynchronous fault. Consequently, a call to `pfm_$inhibit()` can result in the loss of some signals. For that and other reasons, it is good practice to inhibit faults only when absolutely necessary.

On systems with Concurrent Programming Support (CPS), `pfm_$inhibit()` also disables time-sliced task switching. It does not prevent task switching due to voluntary task yielding, either explicitly via `task_$yield()` or implicitly via other functions that yield. Do not use `pfm_$inhibit()` for critical region concurrency control; use the `mutex_` facility instead.

See the reference description of `pfm_$init()` for a list of the C signals that the PFM package intercepts.

NOTE

This call has no effect on the processing of synchronous faults such as floating-point and overflow exceptions, access violations, and so on.

SEE ALSO

pfm_$enable(3), pfm_$inhibit_faults(3), pfm_$init(3).

Concurrent Programming Support Reference.

NAME

pfm_$inhibit_faults - inhibit asynchronous faults but allow time-sliced task switching

SYNOPSIS

C Syntax

```
#include <idl/c/base.h>
#include <ppfm.h>

void pfm_$inhibit_faults(void);
```

Pascal Syntax

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pfm.ins.pas';

procedure pfm_$inhibit_faults;
```

Remarks

To view this manual entry via the *man*(1) command, use the function name shown above without the "$" character.

DESCRIPTION

`pfm_$inhibit_faults()` prevents asynchronous faults (except for time-sliced task switching) from being passed to the calling process. While faults are inhibited, the operating system holds at most one asynchronous fault. Consequently, a call to `pfm_$inhibit_faults()` can result in the loss of some signals. For that and other reasons, it is good practice to inhibit faults only when absolutely necessary.

See the reference description of `pfm_$init()` for a list of the C signals that the PFM package intercepts.

NOTE

This call has no effect on the processing of synchronous faults such as floating-point and overflow exceptions, access violations, and so on.

SEE ALSO

pfm_$enable_faults(3), pfm_$inhibit(3), pfm_$init(3).

NAME

pfm_inhibit - pointer entry to conflicting online manual entries

DESCRIPTION

This manual entry is provided for accessing manual entries whose online versions have conflicting filenames due to maximum name length imposed by short-filename (14-character maximum) systems.

The following message is provided for online manual users:

NOTE

You have selected a name that conflicts with one or more other names. To display the manual entry you want, enter the man command again as follows:

To view this entry:	Use this command:
pfm_inhibit	**man pfm_inhib**
pfm_inhibit_faults	**man pfm_inhib_f**

NAME

pfm_$init - initialize the process fault manager (PFM) package

SYNOPSIS

C Syntax

```
#include <idl/c/base.h>
#include <ppfm.h>

void pfm_$init(
     unsigned long flags)
```

Remarks

To view this manual entry via the *man*(1) command, use the function name shown above without the "$" character.

DESCRIPTION

`pfm_$init()` initializes the PFM package. The *flags* parameter indicates which initialization activities to perform.

Currently, only one *flags* value is valid:

`pfm_$init_signal_handlers()`
: Intercept C signals and convert them to PFM signals. The following HP-UX signals are intercepted: `SIGINT`, `SIGILL`, `SIGFPE`, `SIGTERM`, `SIGHUP`, `SIGQUIT`, `SIGTRAP`, `SIGBUS`, `SIGSEGV`, and `SIGSYS`. On MS-DOS systems, the first four of these, plus `SIGABRT`, are intercepted.

On Apollo systems, the PFM package does not require initialization, and `pfm_$init()` is a no-op. On all other systems, applications that use the PFM package should invoke `pfm_$init()` before invoking any other NCS calls.

NAME

pfm_$intro - fault management

SYNOPSIS (C)

C Syntax

```
#include <idl/c/base.h>
#include <ppfm.h>
```

Pascal Syntax

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/ppfm.ins.pas';
```

Remarks

To view this manual entry via the *man*(1) command, use the function name shown above without the "$" character.

DESCRIPTION

`pfm_$()` calls allow programs to manage signals, faults, and exceptions by establishing cleanup handlers.

NCS software products include a portable subset of the Apollo Domain/OS `pfm_$()` calls:

`pfm_$cleanup()` Establish a cleanup handler.

`pfm_$enable()` Enable asynchronous faults.

`pfm_$enable_faults()`
Enable asynchronous faults after faults have been inhibited via `pfm_$inhibit_faults()`.

`pfm_$inhibit()` Inhibit asynchronous faults.

`pfm_$inhibit_faults()`
Inhibit asynchronous faults but allow time-sliced task switching.

`pfm_$init()` Initialize the PFM package.

`pfm_$reset_cleanup()`
Reset a cleanup handler.

`pfm_$rls_cleanup()`
Release cleanup handlers.

`pfm_$signal()` Signal the calling process.

Cleanup Handlers

A cleanup handler is a piece of code that allows a program to terminate gracefully when it receives an error. A cleanup handler begins with a `pfm_$cleanup()` call and usually ends with a call to `pfm_$signal()` or `pgm_$exit()`, though it can also simply continue back into the program after the cleanup code.

Include Files in NCS Software

This section describes the include files for the `pfm_` interface provided with NCS software.

Version 1.1 of NCK and NIDL, contained a <pfm.h> include file that supports the `std_$call()` calling convention of Apollo SR9 system software, whereby all parameters of a call are passed by reference rather than by value. For example, a call in C source code to `pfm_$reset_cleanup()` resembles:

```
pfm_$reset_cleanup (crec, st)
```

even though both `crec` and `st` are passed by reference to the implementation of `pfm_$reset_cleanup()`. On Apollo SR9 systems, the C compiler treats these parameters as though each was preceded by the address operator `&`. On SunOS, ULTRIX, and VMS systems with Version 1.1 of NCK or NIDL, the <pfm.h> file defines macros that convert these parameters to `&crec` and `&st`.

In Version 1.5.1 of NCK and NIDL, a new include file for the `pfm_$()` calls, <ppfm.h>, is provided. This is the include file for the "portable PFM" interface, an interface in the style of ANSI C. When an application invokes a call through this interface, all output parameters must be preceded by an explicit `&`. For example, a call to `pfm_$reset_cleanup()` resembles:

```
pfm_$reset_cleanup (&crec, &st)
```

since *crec* and *st* are output parameters passed by reference. This calling convention is more natural to most C programmers.

The previous include file, <pfm.h>, is still available, providing backward compatibility for programs coded according to the std_$call() convention. However, new programs should include <ppfm.h>.

Include Files in Apollo SR10 Domain/OS Software

In Apollo SR10 system software, the include file <apollo/pfm.h>, defines the pfm_ interface in the style of ANSI C.

Beginning at SR10.2, the file <apollo/ppfm.h>, which includes <apollo/pfm.h> is also provided; /usr/include/ppfm.h is a symbolic link pointing to /usr/include/apollo/ppfm.h.

Thus, the directive

```
#include <ppfm.h>
```

can be used both on Apollo SR10.2 systems and on other systems with Version 1.5.1 of NCK or NIDL (including HP-UX Releases 8.0 and 8.05).

The signatures for pfm_$reset_cleanup() and pfm_$rls_cleanup() in the SR10.0 and SR10.1 versions of <apollo/pfm.h> are incorrect. They have been corrected at SR10.2. These corrections may require you to modify an application developed on SR10.0 and SR10.1 Apollo systems in order to compile it on an SR10.2 Apollo system. See the reference descriptions of these calls for details.

Constants

pfm_$init_signal_handlers
A constant used as the *flags* parameter to pfm_$init(), causing C signals to be intercepted and converted to PFM signals.

Data Types

pfm_$cleanup_rec
An opaque data type for passing process context among cleanup handler calls.

status_$t
A status code. Most NCS calls supply their completion status in this format. The status_$t type is defined as a structure containing a long integer:

```
struct status_$t {
      long all;
      }
```

However, the calls can also use status_$t as a set of bit fields. To access the fields in a returned status code, assign the value of the status code to a union defined as follows:

```
typedef union {
    struct {
        unsigned fail : 1,
                 subsys : 7,
                 modc : 8;
        short    code;
    } s;
    long all;
} status_u;
```

where:

all	All 32 bits in the status code. If all is equal to status_$ok, the call that supplied the status was successful.
fail	If this bit is set, the error was not within the scope of the module invoked, but occurred within a lower-level module.
subsys	This indicates the subsystem that encountered the error.
modc	This indicates the module that encountered the error.

`code` This is a signed number that identifies the type of error that occurred.

Status Codes

`pfm_$bad_rls_order`
Attempted to release a cleanup handler out of order.

`pfm_$cleanup_not_found`
There is no pending cleanup handler.

`pfm_$cleanup_set`
A cleanup handler was established successfully.

`pfm_$cleanup_set_signalled`
Attempted to use `pfm_$cleanup_set` as a signal.

`pfm_$invalid_cleanup_rec`
Passed an invalid cleanup record to a call.

`pfm_$no_space`
Cannot allocate storage for a cleanup handler.

`status_$ok`
The call was successful.

SEE ALSO

pfm_$cleanup(3), pfm_$enable(3), pfm_$enable_faults(3), pfm_$inhibit(3), pfm_$inhibit_faults(3), pfm_$init(3), pfm_$reset_cleanup(3), pfm_$rls_cleanup(3), pfm_$signal(3).

NAME

pfm_$reset_cleanup - reset a cleanup handler

SYNOPSIS

C Syntax

```
#include <idl/c/base.h>
#include <ppfm.h>

void pfm_$reset_cleanup(
      pfm_$cleanup_rec *cleanup_record,
      status_$t *status)
```

Pascal Syntax

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pfm.ins.pas';

procedure pfm_$reset_cleanup(
      in cleanup_record: pfm_$cleanup_rec;
      out status: status_$t);
```

Remarks

To view this manual entry via the *man*(1) command, use the function name shown above without the "$" character.

DESCRIPTION

`pfm_$reset_cleanup()` re-establishes the cleanup handler last entered so that any subsequent errors enter it first. This procedure should only be used within cleanup handler code.

A implicitly performs a thereby undoing the implicit that performs.

cleanup_record A record of the context at the cleanup handler entry point. It is supplied by *pfm_$cleanup* when the cleanup handler is first established.

status The completion status.

NOTE

This note concerns use of `pfm_$reset_cleanup()` on Apollo systems.

In the SR10.0 and SR10.1 versions of `<apollo/pfm.h>`, the first argument of `pfm_$reset_cleanup()` is incorrectly preceded by an ampersand (`&`). In the SR10.2 version, the first argument is correctly preceded by an asterisk (`*`).

Programs compiled under SR10.0 or SR10.1 will continue to run correctly, since the implementation of `pfm_$reset_cleanup()` has not changed, but you may need to modify these programs in order to compile them under SR10.2. Invocations of `pfm_$reset_cleanup()` that resembled:

```
pfm_$reset_cleanup(crec, &st)
```

when compiled under SR10.0 and SR10.1 must be modified to

```
pfm_$reset_cleanup(&crec, &st)
```

when compiled under SR10.2.

NAME

pfm_$rls_cleanup - release a cleanup handler

SYNOPSIS

C Syntax

```
#include <idl/c/base.h>
#include <ppfm.h>

void pfm_$rls_cleanup(
      pfm_$cleanup_rec *cleanup_record,
      status_$t *status)
```

Pascal Syntax

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pfm.ins.pas';

procedure pfm_$rls_cleanup(
      in cleanup_record: pfm_$cleanup_rec;
      out status: status_$t);
```

Remarks

To view this manual entry via the *man*(1) command, use the function name shown above without the "$" character.

DESCRIPTION

`pfm_$rls_cleanup()` releases the cleanup handler associated with `cleanup_record()`.

On Apollo systems, this call releases the specified cleanup handler and all cleanup handlers established after it.

On other systems, this call releases only the specified cleanup handler, and only the most recently established cleanup handler can be released.

If you are concerned about portability, use `pfm_$rls_cleanup()` only to release the most recent cleanup handler.

cleanup_record specifies the cleanup record to be released by the cleanup handler.

status is the completion status.

ERRORS

`pfm_$bad_rls_order`

The caller attempted to release a cleanup handler other than the one most recently established. On Apollo systems, this status is only a warning; the specified cleanup handler is released, along with any established after it. On other systems, this status probably indicates a user programming error; no cleanup handlers are released, and continued execution may result in more serious errors.

NOTE

This note concerns use of `pfm_$rls_cleanup()` on Apollo systems.

In the SR10.0 and SR10.1 versions of `<apollo/pfm.h>`, the first argument of `pfm_$rls_cleanup()` is incorrectly preceded by an ampersand (`&`). In the SR10.2 version, the first argument is correctly preceded by an asterisk (`*`).

Programs compiled under SR10.0 or SR10.1 will continue to run correctly, since the implementation of `pfm_$rls_cleanup()` has not changed, but you may need to modify these programs in order to compile them under SR10.2. Invocations of `pfm_$rls_cleanup()` that resembled:

```
pfm_$rls_cleanup(crec, &st)
```

when compiled under SR10.0 and SR10.1 must be modified to:

```
pfm_$rls_cleanup(&crec, &st)
```

when compiled under SR10.2.

NAME

pfm_$signal - signal the calling process

SYNOPSIS

C Syntax

```
#include <idl/c/base.h>
#include <ppfm.h>

void pfm_$signal(status_$t fault_signal)
```

C Syntax

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pfm.ins.pas';

procedure pfm_$signal(in fault_signal: status_$t);
```

Remarks

To view this manual entry via the *man*(1) command, use the function name shown above without the "$" character.

DESCRIPTION

`pfm_$signal()` signals the fault specified by *fault_signal* to the calling process. It is usually called to leave cleanup handlers.

fault_signal A fault code.

NOTE

This call does not return when successful.

NAME

pgm_$exit() - exit a program

SYNOPSIS

C Syntax

```
#include <idl/c/base.h>
#include <ppfm.h>

void pgm_$exit(void)
```

Pascal Syntax

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pgm.ins.pas';

procedure pgm_$exit;
```

Remarks

To view this manual entry via the *man*(1) command, use the function name shown above without the "$" character.

DESCRIPTION

`pgm_$exit()` exits from the calling program.

Any cleanup handlers that have been established are executed in sequence from the most recently established to the first.

On Apollo systems, this call invokes `pfm_$signal()` with a fault code equal to the last severity level set by `pgm_$set_severity()`, or `pgm_$ok()` if `pgm_$set_severity()` was not called.

On other systems, this call invokes `pfm_$signal()` with a fault code of `status_$ok`.

SEE ALSO

pfm_$cleanup(3), pfm_$signal(3).

NAME

pgm_$intro() - program management

SYNOPSIS

C Syntax

```
#include <idl/c/base.h>
#include <ppfm.h>
```

Pascal Syntax

```
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/pgm.ins.pas';
```

Remarks

To view this manual entry via the *man*(1) command, use the function name shown above without the "$" character.

DESCRIPTION

A portable version of the Apollo Domain/OS `pgm_$exit()` call is supplied with NCS software products. The include file for the "portable PFM" interface contains a declaration for this call.

NAME

popen(), pclose() - initiate pipe I/O to/from a process

SYNOPSIS

```
#include <stdio.h>

FILE *popen(const char *command, const char *type);

int pclose(FILE *stream);
```

DESCRIPTION

`popen()` creates a pipe between the calling program and a command to be executed by the POSIX shell, `/bin/posix/sh` (see *sh-posix*(1)).

The arguments to `popen()` are pointers to null-terminated strings containing, respectively, a shell command line and an I/O mode, either `r` for reading or `w` for writing.

`popen()` returns a stream pointer such that one can write to the standard input of the command if the I/O mode is `w` by writing to the file *stream*; and one can read from the standard output of the command if the I/O mode is `r` by reading from the file *stream*.

A stream opened by `popen()` should be closed by `pclose()`, which waits for the associated process to terminate and returns the exit status of the command.

Because open files are shared, a type `r` command can be used as an input filter and a type `w` command as an output filter.

RETURN VALUE

`popen()` returns a NULL pointer if files or processes cannot be created. The success of the command execution can be checked by examining the return value of `pclose()`.

`pclose()` returns `-1` if *stream* is not associated with a `popen()`ed command, or 127 if `/bin/posix/sh` could not be executed for some reason.

WARNINGS

If the original and `popen()`ed processes concurrently read or write a common file, neither should use buffered I/O because the buffering will not work properly. Problems with an output filter can be forestalled by careful buffer flushing, e.g., with `fflush()`; see *fclose*(3S).

SEE ALSO

pipe(2), wait(2), fclose(3S), fopen(3S), system(3S).

STANDARDS CONFORMANCE

`popen()`: AES, SVID2, XPG2, XPG3, XPG4, POSIX.2

`pclose()`: AES, SVID2, XPG2, XPG3, XPG4, POSIX.2

NAME

printf(), nl_printf(), fprintf(), nl_fprintf(), sprintf(), nl_sprintf() - print formatted output

SYNOPSIS

```
#include <stdio.h>
int printf(const char *format, /* [arg,] */ ...);
int nl_printf(const char *format, /* [arg,] */ ...);
int fprintf(FILE *stream, const char *format, /* [arg,] */ ...);
int nl_fprintf(FILE *stream, const char *format, /* [arg,] */ ...);
int sprintf(char *s, const char *format, /* [arg,] */ ...);
int nl_sprintf(char *s, const char *format, /* [arg,] */ ...);
```

DESCRIPTION

`printf()` and `nl_printf()` place output on the standard output stream *stdout*.

`fprintf()` and `nl_fprintf()` place output on the named output *stream*.

`sprintf()` and `nl_sprintf()` place "output", followed by the null character (`\0`), in consecutive bytes starting at **s*. It is the user's responsibility to ensure that enough storage is available.

Each function converts, formats, and prints its *arg*s under control of the *format*. *format* is a character string containing two types of objects: plain characters that are copied to the output stream, and conversion specifications, each of which results in fetching zero or more *arg*s. The results are undefined if there are insufficient *arg*s for the format. If the format is exhausted while *arg*s remain, excess *arg*s are ignored.

Each conversion specification is introduced by the character `%` or *%n*`$`, where *n* is a decimal integer in the range 1 through `{NL_ARGMAX}` (`NL_ARGMAX` is defined in `<limits.h>`). The *%n*`$` construction indicates that this conversion should be applied to the *n*th argument, rather than to the next unused one.

An argument can be referenced by a *%n*`$` specification more than once. The two forms of introducing a conversion specification, `%` and *%n*`$`, cannot be mixed within a single *format* string. Improper use of *%n*`$` in a format string results in a negative return value.

After the `%` or *%n*`$`, the following appear in sequence:

1. Zero or more *flags*, which modify the meaning of the conversion specification.
2. An optional string of decimal digits to specify a minimum *field width* in bytes. If the converted value has fewer characters than the field width, it is be padded on the left (or right, if the left-adjustment flag (`-`), described below, has been given) to the field width. If the field width is preceded by a zero, the string is right adjusted with zero-padding on the left (see the leading-zero flag (`0`) described below).
3. A *precision* that gives the minimum number of digits to appear for the `d`, `i`, `o`, `u`, `x`, or `X` conversions, the number of digits to appear after the radix character for the `e` and `f` conversions, the maximum number of significant digits for the `g` conversion, or the maximum number of bytes to be printed from a string in the `s` conversion. The *precision* takes the form of a period (`.`) followed by a decimal digit string; a null digit string is treated as zero.
4. An optional `l` (the letter "ell"), specifying that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion character applies to a long integer *arg*; an optional `l` specifying that a following `n` conversion character applies to a pointer to a long integer *arg*; an optional `h`, specifying that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion character applies to a short integer *arg*; an optional `h` specifying that a following `n` conversion character applies to a pointer to a short integer *arg*; an optional `L` specifying that a following `e`, `E`, `f`, `g`, or `G` conversion character applies to a long double *arg*. An `l`, `h` or `L` before any other conversion character is ignored.
5. A conversion character that indicates the type of conversion to be applied.

A field width or precision can be indicated by an asterisk (`*`) instead of a digit string. In this case, an integer *arg* supplies the field width or precision. The *arg* that is actually converted is not fetched until the conversion letter is seen, so the *arg*s specifying field width, or precision, or both must appear in that order before the *arg*, if any, to be converted. A negative field width is taken as a `-` flag followed by a

positive field width. A negative precision is taken as if the precision were omitted. Format strings containing *%n$* conversion specifications can also indicate a field width or precision by the sequence **n$*. The *n* indicates the position of an integer *arg*. With the **n$* sequence, the *arg*s specifying field width or precision can appear before or after the *arg* to be converted.

The flag characters and their meanings are:

- The resulting conversion is left-justified within the field.

+ The resulting signed conversion always begins with a sign (+ or -).

blank If the first character of a signed conversion is not a sign, a blank is prefixed to the result. This implies that if the blank and + flags both appear, the blank flag is ignored.

This flag specifies that the value is converted to an "alternate*form*". For `c`, `d`, `i`, `s`, `n`, and `u` conversions, the flag has no effect. For `o` conversion, it increases the precision to force the first digit of the result to be a zero. For `x` or `X` conversion, a non-zero result has `0x` or `0X` prefixed to it. For a `p` conversion, a non-zero result has `0x` prefixed to it. For `e`, `E`, `f`, `g`, and `G` conversions, the result always contains a radix character, even if no digits follow the radix (normally, a radix character appears in the resulting conversions only if followed by a digit). For `g` and `G` conversions, trailing zeroes are *not* removed from the result (which they normally are).

0 Leading zeros (following any indication of sign or base) are used to pad to the field width for all conversion characters. No space padding is performed. If both the `0` and - appear, the `0` flag is ignored. For `d`, `i`, `o`, `u`, `p`, `x`, and `X`, conversions, if a precision is specified, the `0` flag is ignored.

The conversion characters and their meanings are:

`d,i,o,u,x,X` The integer *arg* is converted to signed decimal (`d` and `i` are identical), unsigned octal (`o`), decimal (`u`), or hexadecimal notation (`x` and `X`), respectively; the letters `abcdef` are used for `x` conversion and the letters `ABCDEF` for `X` conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeroes. (For compatibility with older versions, padding with leading zeroes can alternatively be specified by inserting a zero in front of the field width. This does not imply an octal value for the field width.) The default precision is 1. The result of converting a zero value with a precision of zero is a null string.

`f` The double *arg* is converted to decimal notation in the style [-]`ddd`*r*`ddd`, where *r* is the radix character. The number of digits after the radix character is equal to the precision specification. If the precision is missing, six digits are output. If the precision is explicitly zero, no radix character appears.

`e,E` The double *arg* is converted in the style [-]`d`*r*`ddde±ddd`, where `r` is the radix character. There is one digit before the radix character and the number of digits after it is equal to the precision; when the precision is missing, six digits are produced; if the precision is zero, no radix character appears. The `E` format code produces a number with `E` instead of `e` introducing the exponent. The exponent always contains at least two digits.

`g,G` The double *arg* is printed in style `f` or `e` (or in style `E` in the case of a `G` format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style `e` is used only if the exponent resulting from the conversion is less than `-4` or greater than or equal to the precision. Trailing zeroes are removed from the fractional part of the result; a radix character appears only if it is followed by a digit.

`c` The int *arg* is converted to an unsigned char, and the resulting character is printed.

`C` The `wchar_t` *arg* is converted to an array of bytes representing the single wide character according to the setting of `LC_CTYPE`. Resulting bytes are printed. If the precision is given, it is ignored. If the field width would otherwise cause the wide character to be split, the wide character is printed and the field width is adjusted upward.

`s` The *arg* is taken to be a string (character pointer) and characters from the string are printed until a null character (`\0`) is encountered or the number of bytes indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed. A NULL value for *arg* yields undefined results.

`S` The *arg* is taken to be a pointer to a wide character string (`wchar_t *`). Wide characters from the string are converted to an array of bytes representing the string of wide characters according to the setting of `LC_CTYPE`. Resulting bytes are printed until a null wide character (`(wchar_t)0`) is encountered or the number of bytes indicated by the precision is reached. If the precision is missing, it is taken to be infinite, so all wide characters up to the first null wide character are printed. If the field width would otherwise cause the last multibyte character to be split, the last wide character is not printed. A NULL value for *arg* yields undefined results.

`p` The value of a pointer to void *arg* is printed as a sequence of unsigned hexadecimal numbers. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeroes. The default precision is 1. The result of converting a zero value with a precision of zero is a null string.

`n` A pointer to an integer *arg* is expected. This pointer is used to store the number of bytes printed on the output stream so far by this call to the function. No argument is converted.

`%` Print a `%`; no argument is converted.

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

Characters generated by `printf()`, `fprintf()`, `nl_printf()`, and `nl_fprintf()` are printed as if `putc()` had been called (see *putc*(3S)).

EXTERNAL INFLUENCES

Locale

The `LC_CTYPE` category affects the following features:

- Plain characters within format strings are interpreted as single and/or multi-byte characters.
- Field width is given in terms of bytes. As characters are placed on the output stream, they are interpreted as single or multi-byte characters and the field width is decremented by the length of the character.
- Precision is given in terms of bytes. As characters are placed on the output stream, they are interpreted as single or multi-byte characters and the precision is decremented by the length of the character.
- The return value is given in terms of bytes. As characters are placed on the output stream, they are interpreted as single or multi-byte characters and the byte count that makes up the return value is incremented by the length of the character.

The `LC_NUMERIC` category determines the radix character used to print floating-point numbers.

International Code Set Support

Single-byte character code sets are supported. Multi-byte character code sets are also supported as described in the `LC_CTYPE` category above.

RETURN VALUE

Each function returns the number of bytes transmitted (excluding the `\0` in the case of `sprintf()` and `nl_sprintf()`), or a negative value if an output error was encountered. Improper use of `%`*n*`$` in a format string results in a negative return value.

ERRORS

`printf()`, `fprintf()`, `nl_printf()`, and `nl_fprintf()` fail if either the *stream* is unbuffered or *stream*'s buffer needed to be flushed causing an underlying `write()` call to be invoked (see *write*(2)), and:

[EAGAIN]	The `O_NONBLOCK` flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation.
[EBADF]	The file descriptor underlying *stream* is not a valid file descriptor open for writing.
[EFBIG]	An attempt was made to write to a file that exceeds the process's file size limit or the maximum file size (see *ulimit*(2)).
[EINTR]	A signal was caught during the `write()` system call.
[EIO]	The process is in a background process group and is attempting to write to its controlling terminal, `TOSTOP` is set, the process is neither ignoring nor blocking the `SIGTTOU` signal, and the process group of the process is orphaned.
[ENOSPC]	There was no free space remaining on the device containing the file.
[EPIPE]	An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A `SIGPIPE` signal is also sent to the process.

Additional `errno` values can be set by the underlying `write()` function (see *write(2)).*

EXAMPLES

To print a date and time in the form "Sunday, July 3, 10:02", where *weekday* and *month* are pointers to null-terminated strings:

```
printf("%s, %s %d, %d:%.2d", weekday, month, day, hour, min);
```

To print π to 5 decimal places:

```
printf("pi = %.5f", 4 * atan(1.0));
```

To create a language-independent date-and-time printing routine write:

```
printf(format,weekday,month,day,hour,min,2,2);
```

For American usage, *format* would point to the string:

```
"%1$s, %2$s %3$d, %4$*6$.*7$d:%5$*6$.*7$d"
```

and result in the output:

```
"Sunday, July 3, 10:02"
```

For German usage, the string:

```
"%1$s, %3$s %2$d, %4$*6$.*7$d:%5$*6$.*7$d"
```

results in the output:

```
Sonntag, 3 Juli 10:02
```

WARNINGS

`nl_printf()`, `nl_fprintf()`, and `nl_sprintf()` are provided for historical reasons only. Their use is not recommended. Use `printf()`, `fprintf()`, and `sprintf()` instead.

Notice that with the `c` conversion character, an int *arg* is converted to an unsigned char. Hence, whole multi-byte characters cannot be printed using a single `c` conversion character.

A precision with the `s` conversion character might result in the truncation of a multi-byte character.

AUTHOR

`printf()`, `fprintf()`, and `sprintf()` were developed by AT&T and HP. `nl_printf()`, `nl_fprintf()`, and `nl_sprintf()` were developed by HP.

SEE ALSO

ecvt(3C), setlocale(3C), putc(3S), scanf(3S), stdio(3S).

STANDARDS CONFORMANCE

`printf()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
`fprintf()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`nl_fprintf()`: XPG2
`nl_printf()`: XPG2

`nl_sprintf()`: XPG2
`sprintf()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

printmsg(), fprintmsg(), sprintmsg() - print formatted output with numbered arguments

SYNOPSIS

```
#include <stdio.h>

int printmsg(const char *format, /* [arg,] */ ...);

int fprintmsg(FILE *stream, const char *format, /* [arg,] */ ...);

int sprintmsg(char *s, const char *format, /* [arg,] */ ...);
```

DESCRIPTION

`printmsg()`, `fprintmsg()`, and `sprintmsg()` are derived from their counterparts in the *printf*(3S) manual entry. The conversion character `%` can be replaced by *%digits*`$`. *digits* are decimal digits representing a number *n* in the range (1 – {`NL_ARGMAX`}) (`NL_ARGMAX` is defined in <`limits.h`>), and indicates that this conversion should be applied to the *n*th argument, rather than to the next unused one. All other aspects of formatting are unchanged. All conversion specifications must contain the *%digits*`$` sequence and the user must ensure correct numbering. All parameters must be used exactly once.

EXTERNAL INFLUENCES

Locale

The `LC_CTYPE` category affects the following features:

- Plain characters within format strings are interpreted as single and/or multi-byte characters.
- Field width is given in terms of bytes. As characters are placed on the output stream, they are interpreted as single or multi-byte characters and the field width is decremented by the length of the character.
- Precision is given in terms of bytes. As characters are placed on the output stream, they are interpreted as single or multi-byte characters and the precision is decremented by the length of the character.
- The return value is given in terms of bytes. As characters are placed on the output stream, they are interpreted as single- or multi-byte characters and the byte count that makes up the return value is incremented by the length of the character.

The `LC_NUMERIC` category determines the radix character used to print floating-point numbers.

International Code Set Support

Single-byte character code sets are supported. Multi-byte character code sets are also supported as described in the `LC_CTYPE` category above.

EXAMPLES

To create a language-independent date and time printing routine, use

```
printmsg(format, weekday, month, day, hour, min);
```

For American usage *format* would point to the string:

```
%1$s, %2$s %3$d, %4$d:%5$.2d
```

resulting in the output:

```
Sunday, July 3, 10:02
```

For German usage, the string:

```
%1$s, %3$d %2$s %4$d:%5$.2d
```

results in the following output:

```
Sonntag, 3 Juli 10:02
```

provided the proper strings have been read.

WARNINGS

These routines are provided for historical reasons only. Use of the *printf*(3S) equivalent routines `printf`, `fprintf()`, and `sprintf()` is recommended.

AUTHOR

`printmsg()` was developed by HP.

SEE ALSO

catgetmsg(3C), setlocale(3C), printf(3S), hpnls(5).

NAME

ptsname - get the name of a slave pty

SYNOPSIS

```
char *ptsname(int fildes);
```

Remarks:

`ptsname()` is useful only on systems that follow the *insf*(1M) naming conventions for ptys.

DESCRIPTION

The passed parameter, *fildes*, is a file descriptor of an opened master pty. `ptsname()` generates the name of the slave pty corresponding to this master pty. This means that their minor numbers will be the same.

RETURN VALUE

Upon successful completion, `ptsname()` returns a string containing the the full path name of a slave pty. Otherwise, a NULL pointer is returned. The return value may point to static data which is overwritten with each call to `ptsname()`, so it should be copied if it is to be saved.

ERRORS

`ptsname()` fails and returns a NULL pointer under the following conditions:

- File descriptor does not refer to an open master pty.
- Request falls outside pty name-space.
- Pty device naming conventions have not been followed.
- `ptsname()` failed to find a match.

EXAMPLES

The following example gets the path of a slave pty corresponding to a master pty obtained through a pty clone open.

```
int fd_master;
char *path;
    ...
    fd_master = open("/dev/ptym/clone", O_RDONLY);
    path = ptsname(fd_master);
```

AUTHOR

`ptsname()` was developed by HP.

SEE ALSO

insf(1M), devnm(3), pty(7).

NAME

putc(), putchar(), fputc(), putw() - put character or word on a stream

SYNOPSIS

```
#include <stdio.h>

int putc(int c, FILE *stream);

int putchar(int c);

int fputc(int c, FILE *stream);

int putw(int w, FILE *stream);
```

DESCRIPTION

putc() Writes the character *c* onto the output *stream* at the position where the file pointer, if defined, is pointing. **putchar**(*c*) is defined as **putc**(*c*, **stdout**). **putc()** and **putchar()** are defined both as macros and as functions.

fputc() Same as **putc()**, but is a function rather than a macro, and can therefore be used as an argument. **fputc()** runs more slowly than **putc()**, but takes less space per invocation, and its name can be passed as an argument to a function.

putw() Writes the word (i.e., **int** in C) *w* to the output *stream* (at the position at which the file pointer, if defined, is pointing). The size of a word is the size of an integer and varies from machine to machine. **putw()** neither assumes nor causes special alignment in the file.

Output streams, with the exception of the standard error stream **stderr**, are by default buffered if the output refers to a file and line-buffered if the output refers to a terminal. The standard error output stream, **stderr**, is by default unbuffered, but use of **freopen()** (see *fopen*(3S)) causes it to become buffered or line-buffered. **setbuf()** or **setvbuf()** (see *setbuf*(3S)) can be used to change the stream's buffering strategy.

RETURN VALUE

On success, **putc()**, **fputc()**, and **putchar()** each return the value they have written. On failure, they return the constant EOF, set the error indicator for the stream, and set **errno** to indicate the error.

On success, **putw()** returns 0. Otherwise, a non-zero value is returned, the error indicator for the stream is set, and **errno** is set to indicate the error.

ERRORS

putc(), **putchar()**, **fputc()**, and **putw()** fail if, either the *stream* is unbuffered or *stream*'s buffer needed to be flushed causing an underlying **write()** call to be invoked, and:

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation.

[EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for writing.

[EFBIG] An attempt was made to write to a file that exceeds the process's file size limit or the maximum file size (see *ulimit*(2)).

[EINTR] A signal was caught during the **write()** system call.

[EIO] The process is in a background process group and is attempting to write to its controlling terminal, **TOSTOP** is set, the process is neither ignoring nor blocking the **SIGTTOU** signal, and the process group of the process is orphaned.

[ENOSPC] There was no free space remaining on the device containing the file.

[EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal is also sent to the process.

Additional **errno** values can be set by the underlying **write()** function (see *write(2)).*

WARNINGS

The **putc()** and **putchar()** routines are implemented as both library functions and macros. The macro versions, which are used by default, are defined in **<stdio.h>**. To obtain the library function either use a **#undef** to remove the macro definition or, if compiling in ANSI-C mode, enclose the function name in parentheses or use the function address. The following example illustrates each of these methods:

```
#include <stdio.h>
#undef putc
   ...
main()
{
     int (*put_char()) ();
          ...
     return_val=putc(c,fd);
          ...
     return_val=(putc)(c,fd1);
          ...
     put_char = putchar;
};
```

Line buffering may cause confusion or malfunctioning of programs that use standard I/O routines but use `read()` themselves to read from standard input. When a large amount of computation is done after printing part of a line on an output terminal, it is necessary to `fflush()` (see *fclose*(3S)) the standard output before beginning the computation.

The macro version of `putc()` incorrectly treats the argument *stream* with side effects. In particular, the following call may not work as expected:

```
putc(c, *f++);
```

The function version of `putc()` or `fputc()` should be used instead.

Because of possible differences in word length and byte ordering, files written using `putw()` are machine-dependent, and may not be readable by `getw()` on a different processor.

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), getc(3S), fread(3S), printf(3S), puts(3S), setbuf(3S).

STANDARDS CONFORMANCE

`putc()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
`fputc()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`putchar()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
`putw()`: AES, SVID2, XPG2, XPG3, XPG4

NAME

putenv() - change or add value to environment

SYNOPSIS

```
#include <stdlib.h>

int putenv(const char *string);
```

DESCRIPTION

string points to a string of the form *name=value*. `putenv()` makes the value of the environment variable *name* equal to *value* by altering an existing variable or creating a new one. In either case, the string pointed to by *string* becomes part of the environment, so altering the string changes the environment. The space used by *string* is no longer used once a new string-defining *name* is passed to `putenv()`.

EXTERNAL INFLUENCES

Locale

The `LC_CTYPE` category determines the interpretation of characters in *string* as single- and/or multi-byte characters.

International Code Set Support

Single- and multi-byte character code sets are supported.

DIAGNOSTICS

`putenv()` returns non-zero if it was unable to obtain enough space via `malloc()` for an expanded environment; otherwise it returns zero.

WARNINGS

`putenv()` manipulates the environment pointed to by *environ*, and can be used in conjunction with `getenv()`. However, *envp* (the third argument to *main*) is not changed.

This routine uses `malloc()` to enlarge the environment (see *malloc*(3C)).

After `putenv()` is called, environmental variables are not in alphabetical order.

A potential error is to call `putenv()` with an automatic variable as the argument, then exit the calling function while *string* is still part of the environment.

SEE ALSO

exec(2), getenv(3C), malloc(3C), environ(5).

STANDARDS CONFORMANCE

`putenv()`: AES, SVID2, XPG2, XPG3, XPG4

NAME

putpwent() - write password file entry

SYNOPSIS

```
#include <pwd.h>

int putpwent(const struct passwd *p, FILE *f);
```

DESCRIPTION

`putpwent()` is the inverse of `getpwent()` (see *getpwent*(3C)). Given a pointer to a *passwd* structure as created by `getpwent()` (or `getpwuid()` or `getpwnam()`), `putpwent()` writes a line on the stream *f*, which matches the format of `/etc/passwd`.

`putpwent()` ignores the audit ID and audit flag in the *passwd* structure; and *does not* create the corresponding entries used in the secure password file (`/.secure/etc/passwd`). `putspwent()` which produces entries that match the secure password file format, must be used to create these entries.

DIAGNOSTICS

`putpwent()` returns non-zero if an error was detected during its operation; otherwise it returns zero.

SEE ALSO

getpwent(3C), putspwent(3C), passwd(4), spasswd(4).

STANDARDS CONFORMANCE

`putpwent()`: SVID2, XPG2

NAME

puts(), fputs() - put a string on a stream

SYNOPSIS

```
#include <stdio.h>

int puts(const char *s);

int fputs(const char *s, FILE *stream);
```

DESCRIPTION

`puts()` writes the null-terminated string pointed to by *s*, followed by a new-line character, to the standard output stream `stdout`.

`fputs()` writes the null-terminated string pointed to by *s* to the named output *stream*, but does *not* append a new-line character.

Neither function writes the terminating null character.

RETURN VALUE

Upon successful completion, `puts()` and `fputs()` return a non-negative number. Otherwise they return EOF, set the error indicator for the stream, and set `errno` to indicate the error.

ERRORS

`puts()` and `fputs()` fail if, either the *stream* is unbuffered or *stream*'s buffer needed to be flushed causing an underlying `write()` call to be invoked, and:

[EAGAIN]	The flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation.
[EBADF]	The file descriptor underlying *stream* is not a valid file descriptor open for writing.
[EFBIG]	An attempt was made to write to a file that exceeds the process's file size limit or the maximum file size (see *ulimit*(2)).
[EINTR]	A signal was caught during the `write()` system call.
[EIO]	The process is in a background process group and is attempting to write to its controlling terminal, `TOSTOP` is set, the process is neither ignoring nor blocking the `SIGTTOU` signal, and the process group of the process is orphaned.
[ENOSPC]	There was no free space remaining on the device containing the file.
[EPIPE]	An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A SIGPIPE signal is also sent to the process.

Additional `errno` values may be set by the underlying `write()` function (see *write(2)).*

SEE ALSO

ferror(3S), fopen(3S), fread(3S), printf(3S), putc(3S).

NOTES

`puts()` appends a new-line character; `fputs()` does not.

STANDARDS CONFORMANCE

`puts()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`fputs()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

putspwent() - write secure password file entry

SYNOPSIS

```
#include <pwd.h>

int putspwent(const struct s_passwd *p, FILE *f);
```

DESCRIPTION

`putspwent()` is the inverse of `getspwent()` (see *getspwent*(3C)). Given a pointer to a `s_passwd` structure, as created by `getspwent()`, `putspwent()` writes a line on the stream *f*, which matches the format of `/.secure/etc/passwd`.

RETURN VALUE

`putspwent()` returns non-zero if it detects an error during its operation; otherwise it returns a value of zero.

AUTHOR

`putspwent()` was developed by HP.

SEE ALSO

getpwent(3C), getspwent(3C), putpwent(3C), spasswd(4).

NAME

putwc(), putwchar(), fputwc() - put a wide character on a stream file

SYNOPSIS

```
#include <wchar.h>

wint_t putwc(wint_t wc, FILE *stream);

wint_t putwchar(wint_t wc);

wint_t fputwc(wint_t wc, FILE *stream);
```

Remarks:

These functions are compliant with the XPG4 Worldwide Portability Interface wide-character I/O functions. They parallel the 8-bit character I/O functions defined in *putc(3S)*.

DESCRIPTION

putwc() Writes the character corresponding to the wide character *wc* onto the output *stream* at the position where the file pointer is pointing. **putwchar**(*wc*) is defined as **putwc**(*wc*, **stdout**). **putwc()** and **putwchar()** are defined both as macros and as functions.

fputwc() Behaves like **putwc()**, but is a function rather than a macro, and can therefore be used as an argument.

Output streams, with the exception of the standard error stream **stderr**, are by default buffered if the output refers to a file and line-buffered if the output refers to a terminal. The standard error output stream, **stderr**, is by default unbuffered, but use of **freopen()** (see *fopen*(3S)) causes it to become buffered or line-buffered. **setbuf()** or **setvbuf()** (see *setbuf*(3S)) can be used to change the stream's buffering strategy.

Definitions for these functions, the type *wint_t* and the value WEOF are provided in the **<wchar.h>** header.

EXTERNAL INFLUENCES

Locale

The **LC_CTYPE** category determines how wide character conversions are done.

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUE

On success, **putwc()**, **fputwc()**, and **putwchar()** each return the wide character corresponding to the value they have written. On failure, they return the constant **WEOF**, set the error indicator for the stream, and set **errno** to indicate the error.

ERRORS

putwc(), **putwchar()**, and **fputwc()** fail if either the *stream* is unbuffered, or *stream*'s buffer needed to be flushed causing an underlying **write()** call to be invoked, and:

[EAGAIN] The **O_NONBLOCK** flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation.

[EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for writing.

[EFBIG] An attempt was made to write to a file that exceeds the process's file size limit or the maximum file size (see *ulimit*(2)).

[EINTR] A signal was caught during the **write()** system call.

[EIO] The process is in a background process group and is attempting to write to its controlling terminal, **TOSTOP** is set, the process is neither ignoring nor blocking the **SIGTTOU** signal, and the process group of the process is orphaned.

[ENOSPC] There was no free space remaining on the device containing the file.

[EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal is also sent to the process.

[EILSEQ] The wide character *wc* does not correspond to a valid character.

Additional `errno` values can be set by the underlying `write()` function (see *write(2)*).

WARNINGS

`putwc()` and `putwchar()` are implemented both as library functions and as macros. The macro versions, which are used by default, are defined in `<wchar.h>`. To obtain the library function either use a `#undef` to remove the macro definition or, if compiling in ANSI-C mode, enclose the function name in parentheses or use the function address. The following example illustrates each of these methods:

```
#include <wchar.h>
#undef putwc
   ...
main()
{
     wint_t (*put_wchar()) ();
          ...
     return_val=putwc(wc,fd);
          ...
     return_val=(putwc)(wc,fd1);
          ...
     put_wchar = putwchar;
};
```

Line buffering may cause confusion or malfunctioning of programs that use wide character I/O routines but use `read()` themselves to read from standard input. When a large amount of computation is done after printing part of a line on an output terminal, it is necessary to `fflush()` (see *fclose*(3S)) the standard output before beginning the computation.

The macro version of `putwc()` incorrectly treats the argument *stream* with side effects. In particular, the following call may not work as expected:

```
putwc(wc, *f++);
```

The function version of `putwc()` or `fputwc()` should be used instead.

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), getwc(3C), fread(3S), printf(3S), fputws(3C), setbuf(3S).

STANDARDS CONFORMANCE

`putwc()`: XPG4

`fputwc()`: XPG4

`putwchar()`: XPG4

NAME

qsort() - quicker sort

SYNOPSIS

```
#include <stdlib.h>

void qsort(
    void *base,
    size_t nel,
    size_t size,
    int (*compar)(const void *, const void *)
);
```

DESCRIPTION

`qsort()` is an implementation of the quicker-sort algorithm. It sorts a table of data in place.

base Pointer to the element at the base of the table.

nel Number of elements in the table.

size Size of each element in the table.

compar Name of the comparison function, which is called with two arguments that point to the elements being compared. The function passed as *compar* must return an integer less than, equal to, or greater than zero, according to whether its first argument is to be considered less than, equal to, or greater than the second. `strcmp()` uses this same return convention (see *strcmp*(3C)).

NOTES

The pointer to the base of the table should be of type pointer-to-element, and cast to type pointer-to-void.

The comparison function need not compare every byte; thus, arbitrary data can be contained in the elements in addition to the values being compared.

The order in the output of two items which compare as equal is unpredictable.

SEE ALSO

sort(1), bsearch(3C), lsearch(3C), string(3C).

WARNINGS

If *size* is zero, a divide-by-zero error might be generated.

STANDARDS CONFORMANCE

`qsort()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

rand(), srand() - simple random-number generator

SYNOPSIS

```
#include <stdlib.h>

int rand(void);

void srand(unsigned int seed);
```

DESCRIPTION

`rand()` uses a multiplicative, congruential, random-number generator with period 2^{32} that returns successive pseudo-random numbers in the range from 0 to 2^{15}-1.

`srand()` can be called at any time to reset the random-number generator to a random starting point. The generator is initially seeded with a value of 1.

NOTE

The spectral properties of `rand()` leave a great deal to be desired. `drand48()` provides a much better, though more elaborate, random-number generator (see *drand48*(3C)).

SEE ALSO

drand48(3C).

STANDARDS CONFORMANCE

`rand()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`srand()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

rcmd(), rresvport(), ruserok() - return a stream to a remote command

SYNOPSIS

```
int rcmd(
    char **ahost,
    unsigned short inport,
    const char *locuser,
    const char *remuser,
    const char *cmd,
    int *fd2p);

int rresvport(int *port);

int ruserok(
    const char *rhost,
    int superuser,
    const char *ruser,
    const char *luser);
```

DESCRIPTION

`rcmd()` A routine used by privileged programs to execute *cmd* on the remote host **ahost* using an authentication scheme based on reserved port numbers. `rcmd()` returns a file descriptor for the socket to which the standard input and standard output of *cmd* are attached. A command level interface to `rcmd()` is provided by `remsh` (see *remsh*(1)), which is the same command as BSD `rsh`.

`rresvport()` Returns a descriptor to a socket with an address in the privileged port space.

`ruserok()` Used by servers to authenticate clients requesting service with `rcmd()`.

Any program using `rcmd()` or `rresvport()` must be run as super-user.

The name of the remote host can be either an official host name or an alias as understood by `gethostbyname()`; (see *gethostent*(3N), *named*(1M), and *hosts*(4)). `rcmd()` looks up the host **ahost* using `gethostbyname()`, returning –1 if the host does not exist. Otherwise **ahost* is set to the standard name of the host and a connection is established to a server residing at the Internet port *inport*. If the connection is refused after five tries, or if it was refused for a reason other than the port being in use, `rcmd()` returns –1.

If the call succeeds, a socket of type SOCK_STREAM is returned to the caller, and given to the remote command as `stdin` and `stdout`. If *fd2p* is non-zero, an auxiliary connection to a control process is set up, and a descriptor for it is placed in **fd2p*. The control process returns diagnostic output from the command on this connection, and also accepts bytes on this connection as UNIX signal numbers, to be forwarded to the process group of the command. If the auxiliary port cannot be set up, `rcmd()` returns –1. If *fd2p* is 0, `stderr` of the remote command is made the same as `stdout`, and no provision is made for sending arbitrary signals to the remote process.

The protocol is described in detail by *remshd*(1M).

`rresvport()` is used to obtain a socket with a privileged address bound to it. This socket is suitable for use by `rcmd()` and several other routines. Privileged addresses consist of a port in the range 0 to 1023. Only the super-user is permitted to bind an address of this sort to a socket.

`ruserok()` verifies that *ruser* on *rhost* is authorized to act as *luser* on the local host. The *superuser* parameter to `ruserok()` is an integer flag that should be non-zero if the local user name corresponds to the super-user. If the *superuser* flag is not set, `ruserok()` first checks the file `/etc/hosts.equiv` to authenticate the request for service. If this check succeeds, `ruserok()` returns 0. If the *superuser* flag is set, or if there is no file `/etc/hosts.equiv`, or if the check fails, `ruserok()` then checks a file `.rhosts` (if there is one) in the local user's home directory. `ruserok()` returns 0 if this check succeeds. Otherwise it returns –1.

Typically, the file `/etc/hosts.equiv` contains a list of host names, and users' `.rhosts` files contain host-name/user-name pairs. A remote user is authenticated

by `ruserok()` if the remote host name appears in `/etc/hosts.equiv` and the remote user name and local user name are the same, or if the remote host name and the remote user name appear together in `.rhosts` in the home directory of the local user.

For a complete explanation of the syntax understood by `ruserok()`, see *hosts.equiv*(4).

DIAGNOSTICS

rcmd Diagnostic Messages

`rcmd()` returns the following diagnostic messages:

hostname`: Unknown host`
: *gethostbyname* was unable to find an entry in the hosts database matching the name of the server (see *gethostent*(3N) and *hosts*(4)).

Next step: Have the system administrator of your host check whether the remote host's entry is in the `hosts` database (see *hosts*(4)).

`connect:` *hostname*`: ...`
: Unable to establish a connection to the reserved port. A message that specifies the reason for the failure is appended to this diagnostic message.

`write: Setting up stderr`
: Error writing to the socket connection set up for error message transmission.

system call`: ...`
: Error executing the system call. Appended to this error is a message specifying the reason for the failure.

`socket: Protocol failure in circuit setup`
: Socket connection not established on a reserved port or socket address not of the Internet family type.

`read:` *hostname*`: ...`
: Error in reading information from the standard socket connection. Appended to this error is a message explaining the reason for the error.

`Connection timeout`
: The remote host did not connect within 30 seconds to the secondary socket set up as an error connection.

`Lost connection`
: The program attempted to read from the socket and failed. This means the socket connection with the remote host was lost.

message...
: An error message can be transmitted through the socket connection from the daemon. That message will be sent to `stderr`.

`primary connection shutdown`
: While waiting for the secondary socket to be set up, `rcmd()` had its primary connection shut down. This may have been caused by an *inetd* security failure.

`recv: ...`
: While trying to set up the secondary (`stderr`) socket, `rcmd()` had an error condition on its primary connection.

`accept: Interrupted system call`
: While trying to set up its secondary socket, `rcmd()` ran out of some resource that caused the accept to be timed out.

Next step: Repeat the command.

rcmd and rresvport Diagnostic Messages

The diagnostic messages associated with `rresvport()` can also appear in `rcmd()` since `rcmd()` calls `rresvport()`:

system call : . . .
Error in executing the system call. The error message returned by the system call is appended to the message.

`socket: All ports in use`
All reserved ports in use. If a timeout occurs, check whether the ARPA Services are installed and `inetd` is running.

EXAMPLES

To execute the `date` command on remote host `hpxzgy` using the remote account `chm`, use `rcmd()` as shown below. This program requires super-user privileges, and the remote account must be equivalent (see *hosts.equiv*(4)) to the local account that runs the program.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <pwd.h>

struct passwd *getpwuid();
char *host[] = { "hpxzgy"  };
char *cmd = "date";
char *ruser = "chm";

main(argc,argv)
        int argc;
        char **argv;
{
        struct servent *sp;
        struct passwd *pwd;
        FILE *fp;
        char ch;
        int rem;

        sp = getservbyname("shell","tcp");
        pwd = getpwuid(getuid());
        rem = rcmd(host, sp->s_port, pwd->pw_name, ruser, cmd, 0);
        if (rem < 0)
                exit(1);   /* rcmd outputs its own error messages */
        fp = fdopen(rem, "r");
        while ((ch = getc(fp)) != EOF)
                putchar(ch);
}
```

WARNINGS

There is no way to specify options to the `socket()` call that `rcmd()` makes. Since `rcmd()` replaces the pointer to the hostname (**ahost*) with a pointer to the standard name of the host in a static data area, this value must be copied into the user's data area if it is to be used later. Otherwise unpredictable results will occur.

AUTHOR

`rcmd()` was developed by the University of California, Berkeley.

SEE ALSO

login(1), rlogin(1), remsh(1), named(1M), remshd(1M), rexecd(1M), gethostent(3N), rexec(3N), hosts.equiv(4).

NAME

readopt() - obtain option code and data from NetIPC option buffer

SYNOPSIS

```
#include <sys/ns_ipc.h>

void readopt(
    short opt[],
    short argnum,
    short *optioncode,
    short *datalength,
    short data[],
    short *result);
```

DESCRIPTION

`readopt()` extracts an option from an option buffer and copies it into a user-supplied data buffer.

`readopt()` recognizes the following parameters:

opt (input parameter) The *opt* parameter to be read.

argnum (input parameter) The number of the argument to be obtained. The first argument is number zero.

optioncode (output parameter) The option code or constant definition (C programs only) associated with the argument. These codes are described in each NetIPC call *opt* parameter description.

datalength (input/output parameter) The length of the data buffer into which the argument should be read. On output, this parameter contains the length of the data actually read. The length of the data associated with a particular option code is provided in each NetIPC call *opt* parameter description.

data (output parameter) A data buffer which will contain the data read from the argument.

result (output parameter) The result code returned. Refer to "Diagnostics" below for more information.

RETURN VALUE

None. Errors are returned in the *result* parameter.

ERRORS

`readopt()` fails and sets *result* to the value indicated if any of the following conditions are encountered:

[NSR_ADDR_OPT] The *opt* buffer pointer is null.

[NSR_NO_ERROR] The call was successful.

[NSR_OPT_CANTREAD] Data in the option buffer has been corrupted and cannot be read.

[NSR_OPT_DATA_LEN] The supplied buffer is not large enough to receive the option.

[NSR_OPT_ENTRY_NUM] The option index is negative or larger than the number of options in the *opt* buffer.

AUTHOR

`readopt()` was developed by HP.

SEE ALSO

ipcconnect(2), ipccontrol(2), ipccreate(2), ipcdest(2), ipcerrmsg(3N), ipcgetnodename(2), ipclookup(2), ipcname(2), ipcnamerase(2), ipcrecv(2), ipcrecvcn(2), ipcselect(2), ipcsend(2), ipcsetnodename(2), ipcshutdown(2), addopt(3N), initopt(3N), optoverhead(3N), readopt(3N).

NAME

regcmp(), regex() - compile and execute regular expression

SYNOPSIS

```
#include <stdlib.h>

char *regcmp(
    const char *string1,
    /* string2, */ ...
    /*, (char *)0 */
);

char *regex(const char *re, const char *subject);

extern char *__loc1;
```

Remarks

Features documented in this manual entry are obsolescent and may be removed in a future HP-UX release. Use of *regcomp*(3C) instead is recommended.

DESCRIPTION

regcmp() compiles a regular expression and returns a pointer to the compiled form. *malloc*(3C) is used to create space for the vector. It is the user's responsibility to free unneeded space so allocated. A NULL return from **regcmp()** indicates an incorrect argument.

regex() executes a compiled pattern against the subject string. Additional arguments are passed to receive values back. **regex()** returns NULL on failure, or a pointer to the next unmatched character on success. A global character pointer **__loc1** points to where the match began. **regcmp()** and **regex()** were largely borrowed from the editor, *ed*(1); however, the syntax and semantics have been changed slightly. The following are the valid symbols and their associated meanings:

[]*.^ These symbols retain their current meaning.

$ Matches the end of the string; **\n** matches a new-line.

- Used within brackets the hyphen signifies a character range. For example, **[a-z]** is equivalent to **[abcd...xyz]**. The **-** can represent itself only if used as the first or last character. For example, the character class expression **[]-]** matches the characters **]** and **-**.

+ A regular expression followed by **+** means one or more times. For example, **[0-9]+** is equivalent to **[0-9][0-9]***.

{m} *{m,}* *{m,u}*
Integer values enclosed in **{ }** indicate the number of times the preceding regular expression can be applied. The value *m* is the minimum number and *u* is a maximum number, which must be no greater than 256. The syntax *{m}* indicates the exact number of times the regular expression can be applied. The syntax *{m,}* is analogous to {*m*,infinity}. The plus (+)**and**asterisk (*) operations are equivalent to **{1,}** and **{0,}** respectively.

(...)$*n* The value of the enclosed regular expression is returned. The value is stored in the (*n*+1)th argument following the subject argument. A maximum of ten enclosed regular expressions are allowed. **regex()** makes its assignments unconditionally.

(...) Parentheses are used for grouping. An operator, such as *, +, or **{ }**, can work on a single character or a regular expression enclosed in parentheses. For example, **(a*(cb+)*)$0**.

Since all of the above defined symbols are special characters, they must be escaped to be used as themselves.

regcmp() and **regex()** are kept in **/lib/libPW.a**, and are linked by using the **-lc** and **-lPW** options to the **ld** or **cc** command. See WARNINGS below.

EXAMPLES

Match a leading new-line in the subject string to which the *cursor* points.

```
char *cursor, *newcursor, *ptr;
    ...
newcursor = regex((ptr = regcmp("^\n", 0)), cursor);
free(ptr);
```

Match through the string **Testing3** and return the address of the character after the last matched character (cursor+11). The string **Testing3** will be copied to the character array **ret0**.

```
char ret0[9];
char *newcursor, *name;
    ...
name = regcmp("([A-Za-z][A-Za-z0-9_]{0,7})$0", 0);
newcursor = regex(name, "123Testing321", ret0);
```

WARNINGS

regcmp() and **regex()** are kept in **/lib/libPW.a**. Unfortunately, **/lib/libPW.a** also contains some functions that have the same names as functions contained in the default C library, **/lib/libc.a**. To prevent unexpected results due to these name conflicts, always search **libc** before searching **libPW**. This is done with the **ld** (or **cc**) command line option sequence **-lc -lPW** which satisifies all standard C functions from **libc** then searches **libPW** for the **regcmp()** and **regex()** functions (there is also an implied **-lc** following the explicit **-lPW** to satisfy any additional C functions required by **regcmp()** and **regex()**).

User programs that use **regcmp()** might run out of memory if **regcmp()** is called iteratively without freeing vectors that are no longer required.

SEE ALSO

ed(1), malloc(3C), regcomp(3C).

NAME

regcomp(), regerror(), regexec(), regfree() - regular expression matching routines

SYNOPSIS

```
#include <regex.h>

int regcomp(regex_t *preg, const char *pattern, int cflags);

int regexec(
    const regex_t *preg,
    const char *string,
    size_t nmatch,
    regmatch_t pmatch[],
    int eflags
);

void regfree(regex_t *preg);

size_t regerror(
    int errcode,
    const regex_t *preg,
    char *errbuf,
    size_t errbuf_size
);
```

DESCRIPTION

These functions interpret regular expressions as described in *regexp*(5). They support both *basic* and *extended* regular expressions.

The structures `regex_t` and `regmatch_t` are defined in the header `<regex.h>`.

The `regex_t` structure contains at least the following member (use of other members results in non-portable code):

`size_t re_nsub` Number of parenthesized subexpressions.

The `regmatch_t` structure contains at least the following members:

regoff_t rm_so Byte offset from start of string to start of substring.

regoff_t rm_eo Byte offset from start of string to the first character after the end of the substring.

`regcomp()` compiles the regular expression specified by the *pattern* argument and places the results in the structure pointed to by *preg*. The *cflags* argument is the bit-wise logical OR of zero or more of the following flags (defined in `<regex.h>`):

`REG_EXTENDED` Use extended regular expressions.

`REG_NEWLINE` IF `REG_NEWLINE` is not set in *cflags*, a newline character in *pattern* or *string* is treated as an ordinary character. If `REG_NEWLINE` is set, newlines are treated as ordinary characters except as follows:

1. A newline in *string* is not matched by a period outside of a bracket expression or by any form of a nonmatching list.
2. A circumflex (^) in *pattern*, when used to specify expression anchoring, matches the zero-length string immediately after a newline in *string*, regardless of the setting of `REG_NOTBOL`.
3. A dollar-sign ($) in *pattern*, when used to specify expression anchoring, matches the zero-length string immediately before a newline in *string*, regardless of the setting of `REG_NOTEOL`.

`REG_ICASE`

Ignore case in match. If a character in *pattern* is defined in the current `LC_CTYPE` locale as having one or more opposite-case counterpoints, both the character and any counterpoints match the pattern character. This applies to all portions of the pattern, including a string of characters specified to be matched via a back-reference expression (\n).

Within bracket expressions: Collation ranges, character classes, and equivalence classes are effectively expanded into equivalent lists of collation elements and characters. Opposite-case counterpoints are then generated for each collation element or character to form the complete matching list or non-matching list for the bracket expression. Opposite-case counterpoints for a multi-character collating element include all possible combinations of opposite-case counterpoints for each individual character comprising the collating element. These are then combined to form new valid multi-character collating elements. For example, the opposite-case counterpoints for `[.ch.]` could be `[.Ch.]`, `[.cH.]`, and `[.CH.]`.

REG_NOSUB
Report only success/fail in **regexec()**.

The default regular expression type for *pattern* is Basic Regular Expression. The application can specify Extended Regular Expressions by using the **REG_EXTENDED** *cflags* value.

If the function **regcomp()** succeeds, it returns zero; otherwise it returns a non-zero value indicating the error.

If **regcomp()** succeeds, and if the **REG_NOSUB** flag was not set in *cflags*, **regcomp()** sets **re_nsub** to the number of parenthesized subexpressions (delimited by `\(` and `\)` in basic regular expressions or `(` and `)` in extended regular expressions) found in *pattern*.

regexec() matches the null-terminated string specified by *string* against the compiled regular expression *preg* initialized by a previous call to **regcomp()**. If it finds a match, **regexec()** returns zero; otherwise it returns non-zero indicating either no match or an error. The *eflags* argument is the bit-wise logical OR of the following flags:

REG_NOTBOL The first character of the string pointed to by *string* is not the beginning of the line. Therefore, the circumflex character (^), when taken as a special character, never matches.

REG_NOTEOL The last character of the string pointed to by *string* is not the end of the line. Therefore, the dollar sign ($), when taken as a special character, never matches.

If *nmatch* is not zero, and **REG_NOSUB** was not set in the *cflags* argument to **regcomp()**, then **regexec()** fills in the *pmatch* array with byte offsets to the substrings of *string* that correspond to the parenthesized subexpressions of *pattern*: *pmatch*[*i*].*rm_so* is the byte offset of the beginning and *pmatch*[*i*].*rm_eo* is the byte offset one byte past the end of the substring *i*. (Subexpression *i* begins at the *i*th matched left parenthesis, counting from 1). Offsets in *pmatch*[0] identify the substring that corresponds to the entire regular expression. Unused elements of *pmatch* are set to –1. If there are more than *nmatch* subexpressions in *pattern* (*pattern* itself counts as a subexpression), **regexec()** still does the match, but only records the first *nmatch* substrings.

When matching a regular expression, any given parenthesized subexpression of *pattern* might participate in the match of several different substrings of *string*, or it might not match any substring, even though the pattern as a whole did match. The following explains which substrings are reported in *pmatch* when matching regular expressions:

1. If subexpression *i* in a regular expression is not contained within another subexpression, and it participated in the match several times, the byte offsets in *pmatch*[*i*] delimit the last such match.
2. If subexpression *i* is not contained within another subexpression, and it did not participate in an otherwise successful match (because either `*`, `?`, or `|` was used), then the byte offsets in *pmatch*[*i*] are –1.
3. If subexpression *i* is contained in subexpression *j*, and a match of subexpression *j* is reported in *pmatch*[*j*], the match or no-match reported in *pmatch*[*i*] is the last one that occurred within the substring in *pmatch*[*j*].
4. If subexpression *i* is contained in subexpression *j*, and the offsets in *pmatch*[*j*] are –1, the offsets in *pmatch*[*i*] will also be –1.
5. If subexpression *i* matched a zero-length string, both offsets in *pmatch*[*i*] refer to the character immediately following the zero-length substring.

If REG_NOSUB was set in *cflags* in the call to **regcomp()**, and *nmatch* is not zero in the call to **regexec()**, the content of the *pmatch* array is unspecified.

regfree() frees any memory allocated by **regcomp()** associated with *preg*.

If the *preg* argument to **regexec()** or **regfree()** is not a compiled regular expression returned by **regcomp()**, the result is undefined. A *preg* can no longer be treated as a compiled regular expression after it is given to **regfree()**.

regerror() provides a mapping from error codes returned by **regcomp()** and **regexec()** to printable strings. **regerror()** generates a string corresponding to the value of the *errcode* parameter, which was the last non-zero value returned by **regcomp()** or **regexec()** with the given value of *preg*. The *errcode* parameter can take on any of the error values defined in <**regex.h**>. If *errbuf_size* is not zero, **regerror()** copies an appropriate error message into the buffer specified by *errbuf*. If the error message (including the terminating null) cannot fit in the buffer, it is truncated to *errbuf_size* – 1 bytes and null terminated.

If *errbuf_size* is zero, the *errbuf* parameter is ignored, but the return value is as defined below.

regerror() returns the size of the buffer (including terminating null) that is required to hold the entire error message.

EXTERNAL INFLUENCES

Locale

The LC_COLLATE category determines the collating sequence used in compiling and executing regular expressions.

The LC_CTYPE category determines the interpretation of text as single and/or multi-byte characters, the characters matched by character-class expressions in regular expressions, and the opposite-case counterpart for each character.

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUE

regcomp() returns zero for success and non-zero for an invalid expression or other failure. **regexec()** returns zero if it finds a match and non-zero for no match or other failure.

ERRORS

If **regcomp()** or **regexec()** detects one of the error conditions listed below, it returns the corresponding non-zero error code. The error codes are defined in the header <**regex.h**>.

REG_BADBR	Contents of \{ \} invalid: Not a number, number too large, more than two numbers, or first larger than second.
REG_BADPAT	Invalid regular expression.
REG_BADRPT	*, +, or + not preceded by valid regular expression.
REG_EBRACE	\{ \} imbalance.
REG_EBRACK	[ ] imbalance.
REG_ECOLLATE	Invalid collation element referenced.
REG_ECTYPE	Invalid character class type named.
REG_EDUPOPER	Duplication operator in illegal position.
REG_EESCAPE	Trailing \ in pattern.
REG_EMEM	Out of memory while matching expression.
REG_ENEWLINE	new-line character found before end of pattern and REG_NEWLINE flag not set.
REG_ENOEXPR	No expression within ( ) or on one side of a |.
REG_ENOSEARCH	No remembered search string.
REG_EPAREN	\(\) imbalance in basic regular expression or ( ) imbalance in extended regular expression.

REG_ERANGE	Invalid endpoint in range statement.
REG_ESPACE	Out of memory for compiled pattern.
REG_ESUBREG	Number in *\digit* invalid or in error.
REG_NOMATCH	regexec() failed to match.
REG_NSUB	Too many parenthesis pairs or nesting level too deep.

EXAMPLES

```
/* match string against the extended regular expression in pattern,
treating errors as no match.  Return 1 for match, 0 for no match.
Print an error message if an error occurs. */

int
match(string, pattern)
char *string;
char *pattern;
{
    int i;
    regex_t re;
    char buf[256];

    i=regcomp(&re, pattern, REG_EXTENDED|REG_NOSUB);
    if (i != 0) {
        (void)regerror(i,&re,buf,sizeof buf);
        printf("%s\n",buf);
        return(0);                          /* report error */
    }
    i = regexec(&re, string, (size_t) 0, NULL, 0);
    regfree(&re);
    if (i != 0) {
        (void)regerror(i,&re,buf,sizeof buf);
        printf("%s\n",buf);
        return(0);                          /* report error */
    }
    return(1);
}
```

The following demonstrates how the REG_NOTBOL flag could be used with regexec() to find all substrings in a line that match a pattern supplied by a user.

```
(void) regcomp(&re, pattern, 0);
/* look for first match at start of line */
error = regexec(&re, &buffer[0], 1, &pm, 0);
while (error == 0) {                /* while matches found */
    /* find next match on line */
    error = regexec(&re, &buffer[pm.rm_eo], 1, &pm, REG_NOTBOL);
}
```

AUTHOR

regcomp(), regerror(), regexec(), and regfree() were developed by HP.

SEE ALSO

regexp(5).

STANDARDS CONFORMANCE

regcomp(): XPG4, POSIX.2
regerror(): XPG4, POSIX.2
regexec(): XPG4, POSIX.2
regfree(): XPG4, POSIX.2

NAME

compile(), step(), advance() - regular expression compile and match routines

SYNOPSIS

```
#define INIT declarations
#define GETC() getc statements
#define PEEKC() peekc statements
#define UNGETC(c) ungetc statements
#define RETURN(pointer) return statements
#define ERROR(val) error statements

#include <regexp.h>

char *compile(
    const char *instring,
    char *expbuf,
    const char *endbuf,
    int eof
);

int step(const char *string, const char *expbuf);

int advance(const char *string, const char *expbuf);

extern char *loc1, *loc2, *locs;

extern int circf, sed, nbra;
```

Remarks

Features documented in this manual entry are obsolescent and may be removed in a future HP-UX release. Use of *regcomp*(3C) functions instead is recommended.

DESCRIPTION

These functions are general-purpose regular expression matching routines to be used in programs that perform Basic Regular Expression (see *regexp*(5)) matching. These functions are defined in **<regexp.h>**.

The functions **step()** and **advance()** do pattern matching given a character string and a compiled regular expression as input. **compile()** takes a Basic Regular Expression as input and produces a compiled expression that can be used with **step()** and **advance()**.

The interface to this file is unpleasantly complex. Programs that include this file must have the following five macros declared before the **#include <regexp.h>** statement. These macros are used by the **compile()** routine.

GETC() Return the value of the next byte in the regular expression pattern. Successive calls to **GETC()** should return successive bytes of the regular expression.

PEEKC() Return the next byte in the regular expression. Successive calls to **PEEKC()** should return the same byte (which should also be the next byte returned by **GETC()**.

UNGETC(*c***)** Cause the argument *c* to be returned by the next call to **GETC()** (and **PEEKC()**). No more than one byte of pushback is ever needed, and this byte is guaranteed to be the last byte read by **GETC()**. The value of the macro **UNGETC(***c***)** is always ignored.

RETURN(*pointer***)**
This macro is used on normal exit of the **compile()** routine. The value of the argument *pointer* is a pointer to the character after the last character of the compiled regular expression. This is useful to programs that must manage memory allocation.

ERROR(*val***)** This is the abnormal return from the **compile()** routine. The argument *val* is an error number (see table below for meanings). This call should never return.

Error	Meaning
11	Range endpoint too large.
16	Bad number.
25	"*digit*" out of range.
36	Illegal or missing delimiter.

41	No remembered search string.
42	\(\) imbalance.
43	Too many \(.
44	More than 2 numbers given in \{ \}.
45	} expected after \.
46	First number exceeds second in \{ \}.
49	[ ] imbalance.
50	Regular expression overflow.

The syntax of the **compile()** routine is as follows:

compile(*instring***, expbuf, endbuf, eof)**

The first parameter *instring* is never used explicitly by the **compile()** routine, but is useful for programs that pass down different pointers to input characters. It is sometimes used in the **INIT** declaration (see below). Programs that call functions to input characters or have characters in an external array can pass down a value of **((char *) 0)** for this parameter.

The next parameter *expbuf* is a character pointer. It points to the location where the compiled regular expression will be placed.

The parameter *endbuf* is one more than the highest address where the compiled regular expression can be placed. If the compiled expression cannot fit in (*endbuf*–*expbuf*) bytes, a call to **ERROR(50)** is made.

The parameter *eof* is the character which marks the end of the regular expression. For example, in *ed*(1), this character is usually a **/**.

Each program that includes this file must have a **#define** statement for **INIT**. This definition is placed right after the declaration for the function **compile()** and the opening curly brace **{**. It is used for dependent declarations and initializations. Most often it is used to set a register variable to point to the beginning of the regular expression so that this register variable can be used in the declarations for **GETC()**, **PEEKC()**, and **UNGETC()**. Otherwise it can be used to declare external variables that might be used by **GETC()**, **PEEKC()**, and **UNGETC()**. See the example below of the declarations taken from *grep*(1).

step() also performs actual regular expression matching in this file. The call to *step* is as follows:

step(*string***, expbuf)**

The first parameter to **step()** is a pointer to a string of characters to be checked for a match. This string should be null-terminated.

The second parameter *expbuf* is the compiled regular expression that was obtained by a call to **compile()**.

step() returns non-zero if the given string matches the regular expression, and zero if the expressions do not match. If there is a match, two external character pointers are set as a side effect to the call to **step()**. The variable set in **step()** is **loc1**. This is a pointer to the first character that matched the regular expression. The variable **loc2**, which is set by the function **advance()**, points to the character after the last character that matches the regular expression. Thus, if the regular expression matches the entire line, **loc1** points to the first character of *string* and **loc2** points to the null at the end of *string*.

step() uses the external variable **circf()**, which is set by **compile()** if the regular expression begins with **^**. If this is set, **step()** tries to match the regular expression to the beginning of the string only. If more than one regular expression is to be compiled before the first is executed, the value of *circf* should be saved for each compiled expression and *circf* should be set to that saved value before each call to **step()**.

advance() is called from **step()** with the same arguments as **step()**. The purpose of **step()** is to step through the *string* argument and call **advance()** until **advance()** returns non-zero, which indicates a match, or until the end of *string* is reached. To constrain *string* to beginning-of-line in all cases, **step()** need not be called; simply call **advance()**.

When **advance()** encounters a ***** or **\{ \}** sequence in the regular expression, it advances its pointer to the string to be matched as far as possible and recursively calls itself, trying to match the rest of the

string to the rest of the regular expression. As long as there is no match, *advance* backs up along the string until it finds a match or reaches the point in the string that initially matched the * or \{ \}. It is sometimes desirable to stop this backing up before the initial point in the string is reached. If the external character pointer **locs** is equal to the point in the string at sometime during the backing up process, **advance()** breaks out of the loop that backs up and returns zero. This is used by *ed*(1) and *sed*(1) for substitutions done globally (not just the first occurrence, but the whole line) so, for example, expressions such as **s/y*//g** do not loop forever.

The additional external variables **sed** and **nbra** are used for special purposes.

EXTERNAL INFLUENCES

Locale

The **LC_COLLATE** category determines the collating sequence used in compiling and executing regular expressions.

The **LC_CTYPE** category determines the interpretation of text as single and/or multi-byte characters, and the characters matched by character class expressions in regular expressions.

International Code Set Support

Single- and multi-byte character code sets are supported.

EXAMPLES

The following is an example of how the regular expression macros and calls look from *grep*(1):

```
#define INIT            register char *sp = instring;
#define GETC()          (*sp++)
#define PEEKC()         (*sp)
#define UNGETC(c)       (--sp)
#define RETURN(c)       return;
#define ERROR(c)        regerr()

#include <regexp.h>
        ...
(void) compile(*argv, expbuf, &expbuf[ESIZE], '\0');
        ...
if (step(linebuf, expbuf))
    succeed();
```

SEE ALSO

grep(1), regcomp(3C), setlocale(3C), regexp(5).

STANDARDS CONFORMANCE

advance(): AES, SVID2, XPG2, XPG3, XPG4
compile(): AES, SVID2, XPG2, XPG3, XPG4

loc1: AES, SVID2, XPG2, XPG3, XPG4
loc2: AES, SVID2, XPG2, XPG3, XPG4

locs: AES, SVID2, XPG2, XPG3, XPG4
step(): AES, SVID2, XPG2, XPG3, XPG4

NAME

reltimer - relatively arm a per-process timer

SYNOPSIS

```
#include <sys/timers.h>

int reltimer(
    timer_t timerid,
    struct itimerspec *value,
    struct itimerspec *ovalue,
);
```

DESCRIPTION

`reltimer()` sets the *it_value* of the specified timer to an offset from the current clock setting.

If `reltimer()` specifies a *value* argument with the `it_value` member equal to zero, the timer is disabled. `reltimer()` updates the *it_interval* value of the timer to the value specified. Time values smaller than the resolution of the specified timer are rounded up to its resolution; timer values larger than the maximum value of the specified timer are rounded down to the maximum value (see *mktimer*(3C)).

`reltimer()` returns in the *ovalue* parameter a value representing the previous amount of time before the timer would have expired or zero if the timer was disabled, together with the previous interval timer period. The members of *ovalue* are subject to the resolution of the timer, and are the same values that would be returned by a `gettimer()` call.

The behavior of this function is undefined if *value* is NULL.

RETURN VALUE

Upon successful completion, `reltimer()` returns zero; otherwise, it returns –1 and sets `errno` to indicate the error.

ERRORS

reltimer() fails if any of the following conditions are encountered:

[EINVAL]	*timerid* does not correspond to an ID returned by `mktimer()` or the value structure specified a nanosecond value less than zero or greater than or equal to 1000 million.
[EIO]	An error occurred while accessing the clock device.

SEE ALSO

gettimer(3C), mktimer(3C), <sys/timers.h>.

STANDARDS CONFORMANCE

`reltimer()`: AES

NAME

remove() - remove a file

SYNOPSIS

```
#include <stdio.h>

int remove(const char *path);
```

DESCRIPTION

remove() removes the file named by *path*. If *path* does not name a directory, **remove**(*path*) is equivalent to **unlink**(*path*). If *path* names a directory, **remove**(*path*) is equivalent to **rmdir**(*path*).

SEE ALSO

rmdir(2), unlink(2).

STANDARDS CONFORMANCE

remove(): AES, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

res_query(), res_search(), res_mkquery(), res_send(), res_init(), dn_comp(), dn_expand(), herror() - resolver routines

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_query(
     char *dname,
     int class,
     int type,
     u_char *answer,
     int anslen
);

int res_search(
     char *dname,
     int class,
     int type,
     u_char *answer,
     int anslen
);

int res_mkquery(
     int op,
     char *dname,
     int class,
     int type,
     char *data,
     int datalen,
     struct rrec *newrr,
     char *buf,
     int buflen
);

int res_send(char *msg, int msglen, char *answer, int anslen);

void res_init();

int dn_comp(
     char *exp_dn,
     char *comp_dn,
     int length,
     char **dnptrs,
     char **lastdnptr
);

int dn_expand(
     char *msg,
     char *eomorig,
     char *comp_dn,
     char exp_dn,
     int  length
);

extern int h_errno;

void herror(char *s);
```

DESCRIPTION

These routines are used for making, sending and interpreting query and reply messages with Internet domain name servers.

Global configuration and state information used by the resolver routines is kept in the structure `_res`. Most of the values have reasonable defaults and can be ignored. Options stored in `_res.options` are defined in <`resolv.h`> and are as follows. Options are stored as a simple bit mask containing the bitwise OR of the options enabled.

`RES_INIT` True if the initial name server address and default domain name are initialized (i.e., `res_init()` has been called).

`RES_DEBUG` Print debugging messages.

`RES_AAONLY` Accept authoritative answers only. With this option, `res_send()` should continue until it finds an authoritative answer or finds an error. Currently this is not implemented.

`RES_PRIMARY` Query the primary server only. Currently this is not implemented.

`RES_USEVC` Use TCP connections for queries instead of UDP datagrams.

`RES_STAYOPEN` Used with `RES_USEVC` to keep the TCP connection open between queries. This is useful only in programs that regularly do many queries. UDP should be the normal mode used.

`RES_IGNTC` The name server will set the truncation bit if all of the data does not fit into the response datagram packet. If `RES_IGNTC` is set, `res_send()` will not retry the query with TCP (i.e., ignore truncation errors).

`RES_RECURSE` Set the recursion-desired bit in queries. This is the default. (`res_send()` does not do iterative queries and expects the name server to handle recursion.)

`RES_DEFNAMES` If set, `res_search()` appends the default domain name to single-component names (those that do not contain a dot). This option is enabled by default.

`RES_DNSRCH` If this option is set, `res_search()` searches for host names in the current domain and in parent domains; see *hostname*(5). This is used by the standard host lookup routine `gethostbyname()` (see *gethostbyname*(3N)). This option is enabled by default.

Primary Routines

`res_init()` Reads the configuration file, `/etc/resolv.conf`, to get the default domain name, search list, and the Internet address of the local name server(s). If no server is configured, the host running the resolver is tried. The current domain name is defined by the hostname if not specified in the configuration file; it can be overridden by the environment variable `LOCALDOMAIN`. Initialization normally occurs on the first call to one of the following routines. If there are errors in the configuration file, they are silently ignored.

`res_query()` Provides an interface to the server query mechanism. It constructs a query, sends it to the local server, awaits a response, and makes preliminary checks on the reply. The query requests information of the specified *type* and *class* for the specified fully-qualified domain name *dname*. The reply message is left in the *answer* buffer with length *anslen* supplied by the caller.

`res_search()` Makes a query and awaits a response much like `res_query()`, but in addition, it implements the default and search rules controlled by the `RES_DEFNAMES` and `RES_DNSRCH` options. It returns the first successful reply.

Other Routines

Routines described here are lower-level routines used by `res_query()`.

`res_mkquery()` Constructs a standard query message and places it in *buf*. It returns the size of the query, or −1 if the query is larger than *buflen*. The query type *op* is usually `QUERY`, but can be any of the query types defined in <`arpa/nameser.h`>. The domain name for the query is given by *dname*. *class* can be any of the query classes defined in <`arpa/nameser.h`>. *type* can be any of the query types defined in <`arpa/nameser.h`>. *data* is the data for an inverse query (`IQUERY`). *newrr* is currently unused but is intended for making update messages.

`res_send()` Sends a pre-formatted query and returns an answer. It calls `res_init()` if `RES_INIT` is not set, sends the query to the local name server, and handles timeouts and retries. `res_send()` returns the length of the reply message, or –1 if there were errors.

`dn_comp()` Compresses the domain name *exp_dn* and stores it in *comp_dn*. The size of the compressed name is returned or –1 if there were errors. *length* is the size of the array pointed to by *comp_dn*. The compression uses an array of pointers *dnptrs* to previously compressed names in the current message. The first pointer points to to the beginning of the message and the list ends with NULL. The limit to the array is specified by *lastdnptr*. A side effect of `dn_comp()` is to update the list of pointers for labels inserted into the message as the name is compressed. If *dnptr* is NULL, names are not compressed. If *lastdnptr* is NULL, the list of labels is not updated.

`dn_expand()` Expands the compressed domain name *comp_dn* to a full domain name. The compressed name is contained in a query or reply message; *msg* is a pointer to the beginning of the message. The uncompressed name is placed in the buffer indicated by *exp_dn* which is of size *length*. The size of compressed name is returned or –1 if there was an error.

RETURN VALUE

Error return status from `res_search()` is indicated by a return value of –1. The external integer `h_errno` can then be checked to see whether this is a temporary failure or an invalid or unknown host. The routine `herror()` can be used to print an error message describing the failure. The argument string *s* is printed first, followed by a colon, a blank, the message, and a new-line.

ERRORS

`h_errno` can have the following values:

`HOST_NOT_FOUND` No such host is known.

`TRY_AGAIN` This is usually a temporary error and means that the local server did not receive a response from an authoritative server. A retry at some later time may succeed.

`NO_RECOVERY` Some unexpected server failure was encountered. This is a non-recoverable error.

`NO_DATA` The name is known to the name server, but there is no data of the requested type associated with this name; this is not a temporary error. Another type of request to the name server using this domain name will result in an answer.

AUTHOR

These resolver routines were developed by the University of California, Berkeley.

FILES

`/etc/resolv.conf` resolver configuration file

SEE ALSO

named(1m), gethostent(3N), resolver(4), hostname(5), RFC1034, RFC1035.

NAME

rexec() - return stream to a remote command

SYNOPSIS

```
int rexec(
    char **ahost,
    int inport,
    const char *user,
    const char *passwd,
    const char *cmd,
    int *fd2p);
```

DESCRIPTION

rexec() arranges for the remote execution of *cmd* on the host ******ahost* as *user*, who is authenticated with *passwd*. It returns a file descriptor for the socket to which the standard input and standard output of *cmd* are attached. A command-level interface to **rexec()** is provided by the **rexec** command (see *remsh*(1)).

rexec() looks up host ******ahost* using **gethostbyname()** (see *gethostbyname*(3N)) and returns –1 if the host does not exist. The host name can be either the official name or an alias. If the **gethostbyname()** call succeeds, ******ahost* is set to the standard name of the host. **rexec()** passes a username and password to the remote host for authentication. These can be specified in the *user* and *passwd* parameters to **rexec()**. If either is NULL, **rexec()** searches for the appropriate information in the **.netrc** file (see *netrc*(4)) in the users's home directory. If this fails, **rexec()** prompts the user for the remote user name and password, defaulting to the local user name and a NULL password.

inport specifies which TCP port to use for the connection; it is normally the value returned by **getservbyname("exec", "tcp")** (see *getservent*(3N)). The protocol used by **rexec()** is described in detail in *rexecd*(1M).

If the call succeeds, a socket of type **SOCK_STREAM** is returned to the caller, and given to the remote command as **stdin** and **stdout**. If the connection to the socket is refused after five tries, or if it was refused for a reason other than the port being in use, **rexec()** returns –1. If *fd2p* is non-zero, an auxiliary connection to a control process is set up and a file descriptor for it is placed in ******fd2p*. The control process returns diagnostic output from the command on this connection and accepts bytes on this connection, interpreting them as UNIX signal numbers to be forwarded to the process group of the command. If the auxiliary port cannot be set up, **rexec()** returns –1. If *fd2p* is 0, **stderr** of the remote command is made the same as **stdout** and no provision is made for sending arbitrary signals to the remote process.

DIAGNOSTICS

rexec() produces the following diagnostic messages:

hostname: **Unknown host**
: The remote host name was not found by **gethostbyname()**.

system call:...
: Error in executing the system call. A message specifying the cause of the failure is appended to this message.

connect: *hostname*:...
: Error in connecting to the socket obtained for **rexec()**. A message specifying the cause of the failure is appended to this diagnostic.

Secondary socket:...
: Error in creating a secondary socket for error transmission to be used by **rexec()**.

read: *hostname*:...
: Error in reading information transmitted over the socket. A message specifying the cause of the failure is appended to this diagnostic.

Connection timeout
: The remote host did not connect within 30 seconds to the secondary socket set up as an error connection.

Lost connection
: The program attempts to read from the socket and fails. This means the socket connection with the remote host was lost.

`.netrc:...`
Error in opening `.netrc` file in the home directory for a reason other than the file not existing.

`Error- .netrc file not correct mode.`
`Remove password or correct mode.`

The `.netrc` file is readable, writable or executable by anyone other than the user.

Next step: Check whether `.netrc` has been modified by someone else and change the mode of `.netrc` (**chmod 400 .netrc**).

`Unknown .netrc option ...`
An unrecognized keyword has been found in `.netrc` (see *netrc*(4)).

Next step: Correct keyword in `.netrc`.

`primary connection shutdown`
While waiting for the secondary socket to be set up, `rexec()` had its primary connection shut down. This may have been caused by the `inetd` security failure.

`recv:...`
While trying to set up the secondary (`stderr`) socket, `rexec()` had an error condition on its primary connection.

`accept: Interrupted system call`
While trying to set up a secondary socket, `rexec()` ran out of a resource, which caused the accept to be timed out.

Next step: Repeat the command. If a timeout occurs, check whether the ARPA Services are installed and `inetd` is running.

EXAMPLE
To execute the `date` command on remote host `hpxzgy` using the remote account `chm`, `rexec()` could be used as follows:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

char *host[] = { "hpxzgy" };
char *user = "chm";
char *passwd = "password";
char *cmd = "date";

main(argc, argv)
      char **argv;
      int argc;
{
      char ch;
      struct servent *servent;
      FILE *fp;
      int sd;

      servent = getservbyname("exec", "tcp");
      sd = rexec(host, servent->s_port, user, passwd, cmd, 0);
      fp = fdopen(sd, "r");
      while ((ch = getc(fp)) != EOF)
            putchar(ch);
}
```

WARNINGS
There is no way to specify options to the `socket()` call that `rexec()` makes.

A program using **rexec()** should not be put in the background when **rexec()** is expected to prompt for a password or user name. If it is put in the background it will compete with the shell for input.

Since **rexec()** replaces the pointer to the hostname (**ahost*) with a pointer to the standard name of the host in a static data area, this value must be copied into the user's data area if it is to be used later.

The password is sent unencrypted through the socket connection.

AUTHOR

rexec() was developed by the University of California, Berkeley.

SEE ALSO

remsh(1), rexecd(1M), gethostent(3N), getservent(3N), rcmd(3N), netrc(4).

NAME

rmtimer - free a per-process timer

SYNOPSIS

```
#include <sys/timers.h>

int rmtimer(timer_t timerid);
```

DESCRIPTION

`rmtimer()` is used to free a previously allocated timer (returned by `mktimer()`. Any pending timer event to be generated by this timer has been cancelled when the call returns.

RETURN VALUE

Upon successful completion, `rmtimer()` returns zero; otherwise, it returns –1 and sets `errno` to indicate the error.

ERRORS

rmtimer() fails if the following condition is encountered:

[EINVAL] *timerid* is not a valid timer ID.

SEE ALSO

mktimer(3C), reltimer(3C), <sys/timers.h>

STANDARDS CONFORMANCE

`rmtimer()`: AES

NAME

rnusers(), rusers() - return information about users on remote machines

SYNOPSIS

```
#include <utmp.h>
#include <rpcsvc/rusers.h>

int rnusers(char *host);

int rusers(char *host, struct utmpidlearr *up);
```

DESCRIPTION

`rnusers()` returns the number of users logged in on *host* or −1 if it cannot determine that number. The *host* string is either the official name of the host or an alias for it. See *hosts*(4) for more information regarding host names.

`rusers()` fills in the `utmpidlearr` structure with data about *host* and returns 0 if successful. The `ut_line` field is limited to eight characters on Berkeley systems, so the HP-UX XDR routine truncates from 12 characters to 8. The `nonuser()` macro does not exist in the HP-UX `utmp.h` file; therefore, HP-UX windows appear as separate users.

The relevant structures are:

```
struct utmparr {                    /* RUSERSVERS_ORIG */
      struct utmp **uta_arr;
      int uta_cnt;
};

struct utmpidle {
      struct utmp ui_utmp;
      unsigned ui_idle;
};

struct utmpidlearr {          /* RUSERSVERS_IDLE */
      struct utmpidle **uia_arr;
      int uia_cnt;
};
```

RPC Information

program number:
RUSERSPROG

xdr routines:

```
int xdr_utmp(xdrs, up)
        XDR *xdrs;
        struct utmp *up;
int xdr_utmpidle(xdrs, ui)
        XDR *xdrs;
        struct utmpidle *ui;
int xdr_utmpptr(xdrs, up)
        XDR *xdrs;
        struct utmp **up;
int xdr_utmpidleptr(xdrs, up)
        XDR *xdrs;
        struct utmpidle **up;
int xdr_utmparr(xdrs, up)
        XDR *xdrs;
        struct utmparr *up;
int xdr_utmpidlearr(xdrs, up)
        XDR *xdrs;
        struct utmpidlearr *up;
```

procs:
RUSERSPROC_NUM

No arguments, returns number of users as an *unsigned long*.

RUSERSPROC_NAMES
No arguments, returns *utmparr* or *utmpidlearr*, depending on version number.

RUSERSPROC_ALLNAMES
No arguments, returns *utmparr* or *utmpidlearr*, depending on version number. Returns listing even for *utmp* entries satisfying *nonuser()* in *utmp.h*.

versions:
RUSERSVERS_ORIG
RUSERSVERS_IDLE

structures:

WARNING

User applications that call this routine must be linked with `/usr/include/librpcsvc.a`. For example,

```
cc my_source.c -lrpcsvc
```

AUTHOR

`rnusers()` was developed by Sun Microsystems, Inc.

SEE ALSO

rusers(1).

NAME

rpc() - library routines for remote procedure calls

DESCRIPTION

These routines allow C programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a data packet to the server. Upon receipt of the packet, the server calls a dispatch routine to perform the requested service and then sends back a reply. Finally, the procedure call returns to the client.

Functions

`auth_destroy()`	Destroy authentication information handle.
`authnone_create()`	Return RPC authentication handle with no checking.
`authunix_create()`	Return RPC authentication handle with UNIX permissions.
`authunix_create_default()`	Return default UNIX authentication handle.
`callrpc()`	Call remote procedure, given [prognum,versnum,procnum].
`clnt_broadcast()`	Broadcast remote procedure call everywhere .
`clnt-call()`	Call remote procedure associated with client handle.
`clnt_control()`	Change or retrieve information associated with a client handle.
`clnt_create()`	Create RPC client using the transport specified by the caller.
`clnt_destroy()`	Destroy client's RPC handle.
`clnt_freeres()`	Free data allocated by RPC/XDR system when decoding results.
`clnt_geterr()`	Copy error information from client handle to error structure.
`clnt_pcreateerror()`	Print message to stderr about why client handle creation failed.
`clnt_perrno()`	Print message to stderr corresponding to condition given.
`clnt_perror()`	Print message to stderr explaining why an RPC call failed.
`clnt_spcreateerror()`	Return a pointer to a null-delimited string telling why the client handle creation failed.
`clnt_sperrno()`	Return a pointer to a null-delimited string containing a message corresponding to the error value passed to this function.
`clnt_sperror()`	Return a pointer to a null-delimited string telling why an RPC call failed.
`clntraw_create()`	Create toy RPC client for simulation.
`clnttcp_create()`	Create RPC client using TCP transport.
`clntudp_create()`	Create RPC client using UDP transport.
`get_myaddress()`	Get the machine's IP address.
`gettransient()`	Get a program number in the transient range.
`pmap_getmaps()`	Return list of RPC program-to-port mappings.
`pmap_getport()`	Return port number on which waits supporting service.
`pmap_rmtcall()`	Instruct portmapper to make an RPC call.
`pmap_set()`	Establish mapping between [prognum,versnum,procnum] and port.
`pmap_unset()`	Destroy mapping between [prognum,versnum,procnum] and port.
`registerrpc()`	Register procedure with RPC service package.
`rpc_createerr()`	Global variable indicating reason why client creation failed.
`svc_destroy`	Destroy RPC service transport handle.

`svc_fdset()`	Global array with RPC service file descriptor mask; can handle up to NOFILE socket descriptors (NOFILE defined in header file <`sys/parm.h`>).
`svc_freeargs()`	Free data allocated by RPC/XDR system when decoding arguments.
`svc_getargs()`	Decode the arguments of an RPC request.
`svc_getcaller()`	Get the network address of the caller of a procedure.
`svc_getreqset()`	Return when all associated sockets have been serviced.
`svc_register()`	Associate *prognum* and *versnum* with service dispatch procedure.
`svc_run()`	Wait for RPC requests to arrive and call appropriate service.
`svc_sendreply()`	Send back results of a remote procedure call.
`svc_unregister()`	Remove mapping of [prognum,versnum] to dispatch routines.
`svcerr_auth()`	Called when refusing service because of authentication error.
`svcerr_decode()`	Called when service cannot decode its parameters.
`svcerr_noproc()`	Called when service hasn't implemented the desired procedure.
`svcerr_noprog()`	Called when program is not registered with RPC package.
`svcerr_progvers()`	Called when version is not registered with RPC package.
`svcerr_systemerr()`	Called when service detects system error.
`svcerr_weakauth()`	Called when refusing service because of insufficient authentication.
`svcfd_create()`	Create an RPC service from an existing socket.
`svcraw_create()`	Create a toy RPC service transport for testing.
`svctcp_create()`	Create an RPC service based on TCP transport.
`svcudp_create()`	Create an RPC service based on UDP transport.
`xdr_accepted_reply()`	Generate RPC-style replies without using RPC package.
`xdr_authunix_parms()`	Generate UNIX credentials without using RPC package.
`xdr_callhdr()`	Generate RPC-style headers without using RPC package.
`xdr_callmsg()`	Generate RPC-style messages without using RPC package.
`xdr_opaque_auth()`	Describe RPC messages, externally.
`xdr_pmap()`	Describe parameters for portmap procedures, externally.
`xdr_pmaplist()`	Describe a list of port mappings, externally.
`xdr_rejected_reply()`	Generate RPC-style rejections without using RPC package.
`xdr_replymsg()`	Generate RPC-style replies without using RPC package.
`xprt_register()`	Register RPC service transport with RPC package.
`xprt_unregister()`	Unregister RPC service transport from RPC package

AUTHOR

`rpc` was developed by Sun Microsystems, Inc.

SEE ALSO

Programming and Protocols for NFS Services.

NAME

rstat(), havedisk() - get performance data from remote kernel

SYNOPSIS

```
#include <time.h>
#include <rpcsvc/rstat.h>

int havedisk(char *host);

int rstat(char *host, struct statstime *statp);
```

DESCRIPTION

havedisk() returns 1 if *host* has a disk, 0 if it does not, and –1 if this cannot be determined. The *host* string is either the official name of the host or an alias for it. See *hosts*(4) for more information regarding host names.

rstat() fills in the *statstime* structure for *host*, and returns 0 if it was successful. The relevant structures are:

```
struct stats {                  /* RSTATVERS_ORIG */
    int cp_time[CPUSTATES];     /* the time spent in each CPU state */
    int dk_xfer[DK_NDRIVE];     /* total number of disk transfers
                                   on each of the disk interfaces */
    unsigned v_pgpgin;          /* total VM pages paged in */
    unsigned v_pgpgout;         /* total VM pages paged out */
    unsigned v_pswpin;          /* total VM pages paged swapped in */
    unsigned v_pswpout;         /* total VM pages paged swapped out */
    unsigned v_intr;            /* total interrupts */
    int if_ipackets;            /* inbound packets on all interfaces */
    int if_ierrors;             /* inbound errors on all interfaces */
    int if_opackets;            /* outbound packets on all interfaces */
    int if_oerrors;             /* outbound errors on all interfaces */
    int if_collisions;          /* collisions seen on all interfaces */
};

struct statsswtch {             /* RSTATVERS_SWTCH */
    int cp_time[CPUSTATES];     /* the time spent in each CPU state */
    int dk_xfer[DK_NDRIVE];     /* total number of disk transfers
                                   on each of the disk interfaces */
    unsigned v_pgpgin;          /* total VM pages paged in */
    unsigned v_pgpgout;         /* total VM pages paged out */
    unsigned v_pswpin;          /* total VM pages paged swapped in */
    unsigned v_pswpout;         /* total VM pages paged swapped out */
    unsigned v_intr;            /* total interrupts */
    int if_ipackets;            /* inbound packets on all interfaces */
    int if_ierrors;             /* inbound errors on all interfaces */
    int if_opackets;            /* outbound packets on all interfaces */
    int if_oerrors;             /* outbound errors on all interfaces */
    int if_collisions;          /* collisions seen on all interfaces */
    unsigned v_swtch;           /* total context switches */
    long avenrun[3];            /* average number of running jobs */
    struct timeval boottime;    /* time of last boot */
};

struct statstime {              /* RSTATVERS_TIME */
    int cp_time[CPUSTATES];     /* the time spent in each CPU state */
    int dk_xfer[DK_NDRIVE];     /* total number of disk transfers
                                   on each of the disk interfaces */
    unsigned v_pgpgin;          /* total VM pages paged in */
    unsigned v_pgpgout;         /* total VM pages paged out */
    unsigned v_pswpin;          /* total VM pages paged swapped in */
    unsigned v_pswpout;         /* total VM pages paged swapped out */
```

```
        unsigned v_intr;          /* total interrupts */
        int if_ipackets;          /* inbound packets on all interfaces */
        int if_ierrors;           /* inbound errors on all interfaces */
        int if_opackets;          /* outbound packets on all interfaces */
        int if_oerrors;           /* outbound errors on all interfaces */
        int if_collisions;        /* collisions seen on all interfaces */
        unsigned v_swtch;         /* total context switches */
        long avenrun[3];          /* average number of running jobs */
        struct timeval boottime; /* time of last boot */
        struct timeval curtime;  /* current system time */
};
```

RPC Info

program number:
RSTATPROG

xdr routines:
int xdr_stats(xdrs, stat)
XDR *xdrs;
struct stats *stat;
int xdr_statsswtch(xdrs, stat)
XDR *xdrs;
struct statsswtch *stat;
int xdr_statstime(xdrs, stat)
XDR *xdrs;
struct statstime *stat;
int xdr_timeval(xdrs, tv)
XDR *xdrs;
struct timeval *tv;

procs:
RSTATPROC_HAVEDISK
Takes no arguments, returns *long*
which is true if remote host has a disk.
RSTATPROC_STATS
Takes no arguments, return *struct statsxxx*,
depending on version.

versions:
RSTATVERS_ORIG
RSTATVERS_SWTCH
RSTATVERS_TIME

WARNING

User applications that call this routine must be linked with `/usr/include/librpcsvc.a`. For example,

```
cc my_source.c -lrpcsvc
```

AUTHOR

`rstat()` was developed by Sun Microsystems, Inc.

SEE ALSO

rup(1), rstatd(1M).

NAME

rwall() - write to specified remote machines

SYNOPSIS

```
#include <rpcsvc/rwall.h>

int rwall(char *host, char *msg);
```

DESCRIPTION

`rwall()` causes *host* to print the string *msg* to all its users. It returns 0 if successful.

RPC Info

program number:

WALLPROG

procs:

WALLPROC_WALL

Takes string as argument (wrapstring), returns no arguments. Executes *wall* on remote host with string.

versions:

RSTATVERS_ORIG

WARNING

User applications that call this routine must be linked with `/usr/include/librpcsvc.a`. For example,

```
cc my_source.c -lrpcsvc
```

AUTHOR

`rwall()` was developed by Sun Microsystems, Inc.

SEE ALSO

rwall(1M), rwalld(1M), shutdown(1M).

NAME

scandir(), alphasort() - scan a directory

SYNOPSIS

```
#include <dirent.h>

int scandir(
    const char *dirname,
    struct dirent **namelist,
    int (*select)(const struct dirent * const *),
    int (*compar)(
        const struct dirent * const *,
        const struct dirent * const *
    )
);

int alphasort(
    const struct dirent * const *d1,
    const struct dirent * const *d2
);
```

DESCRIPTION

`scandir()` reads the directory *dirname* and builds an array of pointers to directory entries using `malloc()` (see *malloc*(3C)). It returns the number of entries in the array and a pointer to the array through *namelist*.

The *select* parameter is a pointer to a user-supplied subroutine which is called by `scandir()` to select which entries are to be included in the array. The select routine is passed a pointer to a directory entry and should return a non-zero value if the directory entry is to be included in the array. If *select* is null, then all the directory entries will be included.

The *compar* parameter is a pointer to a user-supplied subroutine which is passed to *qsort*(3C) to sort the completed array. If this pointer is null, the array is not sorted. `alphasort()` is a routine which can be used for the *compar* parameter to sort the array alphabetically.

The memory allocated for the array can be deallocated with `free()` (see *malloc*(3C)) by freeing each pointer in the array and the array itself.

EXTERNAL INFLUENCES

Locale

The `LC_COLLATE` category determines the collation ordering used by `alphasort()`. See *hpnls*(5) for a description of supported collation features.

The `LC_CTYPE` category determines the interpretation of bytes in the file name portion of directory entries as single- and/or multi-byte characters by the `alphasort()` function.

Results are undefined if the locales specified by the `LC_COLLATE` and `LC_CTYPE` categories use different code sets.

International Code Set Support

Single- and multi-byte character code sets are supported for `alphasort()`.

RETURN VALUE

`scandir()` returns -1 if the directory cannot be opened for reading or if `malloc()` cannot allocate enough memory to hold all the data structures.

EXAMPLE

The example program below scans the `/tmp` directory. It does not exclude any entries since *select* is NULL. The contents of `namelist` are sorted by `alphasort()`. It prints out how many entries are in `/tmp` and the sorted entries of the `/tmp` directory. The memory used by `scandir()` is returned using `free()`.

```
#include <sys/types.h>
#include <stdio.h>
#include <dirent.h>
```

```
extern int scandir();
extern int alphasort();

main()
{
    int num_entries, i;
    struct dirent **namelist, **list;

    if ((num_entries = scandir("/tmp", &namelist, NULL, alphasort)) < 0) {
        fprintf(stderr, "Unexpected error\n");
        exit(1);
    }
    printf("Number of entries is %d\n", num_entries);
    if (num_entries) {
        printf("Entries are:");
        for (i=0, list=namelist; i<num_entries; i++) {
            printf(" %s", (*list)->d_name);
            free(*list);
            *list++;
        }
        free(namelist);
        printf("\n");
    }
    printf("\n");
    exit(0);
}
```

SEE ALSO

directory(3C), malloc(3C), qsort(3C), string(3C), dirent(5), hpnls(5).

NAME

scanf, fscanf, sscanf, nl_scanf, nl_fscanf, nl_sscanf - formatted input conversion, read from stream file

SYNOPSIS

```
#include <stdio.h>
int scanf(const char *format, /* [pointer,] */ ...);
int fscanf(FILE *stream, const char *format, /* [pointer,] */ ...);
int sscanf(const char *s, const char *format, /* [pointer,] */ ...);
int nl_scanf(const char *format, /* [pointer,] */ ...);
int nl_fscanf(FILE *stream, const char *format, /* [pointer,] */ ...);
int nl_sscanf(const char *s, const char *format, /* [pointer,] */ ...);
```

DESCRIPTION

`scanf()` and `nl_scanf()` read from the standard input stream *stdin*.

`fscanf()` and `nl_fscanf()` read from the named input *stream*.

`sscanf()` and `nl_sscanf()` read from the character string *s*.

Each function reads characters, interprets them according to the control string *format* argument, and stores the results in its *pointer* arguments. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are ignored. The control string contains conversion specifications and other characters used to direct interpretation of input sequences. The control string contains:

- White-space characters (blanks, tabs, newlines, or formfeeds) that cause input to be read up to the next non-white-space character (except in two cases described below).
- An ordinary character (not `%`) that must match the next character of the input stream.
- Conversion specifications, consisting of the character `%`, an optional assignment suppressing character `*`, an optional numerical maximum-field width, an optional `l` (ell), `h` or `L` indicating the size of the receiving variable, and a conversion code.
- The conversion specification can alternatively be prefixed by the character sequence `%`*n*`$` instead of the character `%`, where *n* is a decimal integer in the range (`1`–`{NL_ARGMAX}`) (`NL_ARGMAX` is defined in `<limits.h>`). The `%`*n*`$` construction indicates that the value of the next input field should be placed in the *n*th argument, rather than to the next unused one. The two forms of introducing a conversion specification, `%` and `%`*n*`$`, must not be mixed within a single *format* string with the following exception: Skip fields (see below) can be designated as `%*` or `%`*n*`$*`. In the latter case, *n* is ignored.

Unless the specification contains the `n` conversion character (described below), a conversion specification directs the conversion of the next input field. The result of a conversion specification is placed in the variable to which the corresponding argument points, unless `*` indicates assignment suppression. Assignment suppression provides a way to describe an input field to be skipped. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted. For all descriptors except `[` and `c`, white space leading an input field is ignored.

The conversion code indicates the interpretation of the input field; the corresponding pointer argument must be of a restricted type. For a suppressed field, no pointer argument is given. The following conversion codes are legal:

`%`	A single `%` is expected in the input at this point; no assignment is done.
`d`	A decimal integer is expected; the corresponding argument should be an integer pointer.
`u`	An unsigned decimal integer is expected; the corresponding argument should be an unsigned integer pointer.
`o`	An octal integer is expected; the corresponding argument should be an unsigned integer pointer.

x, X A hexadecimal integer is expected; the corresponding argument should be an unsigned integer pointer. The **x** and **X** conversion characters are equivalent.

i An integer is expected; the corresponding argument should be an integer pointer. The value of the next input item, interpreted according to C conventions, will be stored; a leading **0** implies octal, a leading **0x** implies hexadecimal; otherwise, decimal is assumed.

n Cause the total number of bytes (including white space) scanned since the function call to be stored; the corresponding argument should be an integer pointer. No input is consumed. The function return value does not include *%n* assignments in the count of successfully matched and assigned input items.

e,E,f,g,G A floating-point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a *float*. The input format for floating-point numbers is an optionally signed string of digits, possibly containing a radix character, followed by an optional exponent field consisting of an **E** or an **e**, followed by an optional **+**, **-**, or space, followed by an integer. The conversion characters **E** and **G** behave the same as, respectively, **e** and **g**.

C A character is expected; the corresponding argument should be a **wchar_t** pointer. The normal skip-over-white-space is suppressed in this case; to read the next non-space character, use **%1S**. The character is read and converted into a wide character according to the setting of **LC_CTYPE**. If a field width is given, the corresponding argument refers to a wide character array; the indicated number of characters is read and converted.

c A character is expected; the corresponding argument should be a character pointer. The normal skip-over-white-space is suppressed in this case; to read the next non-space character, use **%1s**. If a field width is given, the corresponding argument refers to a character array; the indicated number of characters is read.

S A character string is expected; the corresponding argument should be a **wchar_t** pointer pointing to an array of wide characters large enough to accept the string and a terminating **(wchar_t)0**, which is added automatically. Characters are read and converted into wide characters according to the setting of **LC_CTYPE**. The input field is terminated by a white-space character. **scanf()** cannot read a null string.

s A character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating **\0**, which is added automatically. The input field is terminated by a white-space character. **scanf()** cannot read a null string.

[Indicates string data and the normal skip-over-leading-white-space is suppressed. The left bracket is followed by a set of characters, called the *scanset*, and a right bracket; the input field is the maximal sequence of input characters consisting entirely of characters in the scanset. The circumflex (**^**), when it appears as the first character in the scanset, serves as a complement operator and redefines the scanset as the set of all characters *not* contained in the remainder of the scanset string. Construction of the *scanset* follows certain conventions. A range of characters may be represented by the construct *first-last*, enabling [0123456789] to be expressed [0-9]. Using this convention, *first* must be lexically less than or equal to *last*; otherwise, the dash stands for itself. The dash also stands for itself when it is the first or the last character in the scanset. To include the right square bracket as an element of the scanset, it must appear as the first character (possibly preceded by a circumflex) of the scanset, in which case it will not be interpreted syntactically as the closing bracket. The corresponding argument must point to a character array large enough to hold the data field and the terminating **\0**, which are added automatically. At least one character must match for this conversion to succeed.

p A sequence of unsigned hexadecimal numbers is expected. This sequence may be produced by the **p** conversion character of **printf()**. The corresponding argument shall be a pointer to a pointer to **void** into which the value represented by the hexadecimal sequence is stored. The behavior of this conversion is undefined for any input

item other than a value converted earlier during the same program execution.

The conversion characters **d**, **i**, and **n** can be preceded by **l** or **h** to indicate that a pointer to a **long int** or **short int** rather than to an **int** is in the argument list. Similarly, the conversion characters **u**, **o**, **x**, and **X** can be preceded by **l** or **h** to indicate that a pointer to **unsigned long int** or **unsigned short int** rather than to an **unsigned int** is in the argument list. Finally, the conversion characters **e**, **E**, **f**, **g**, and **G** can be preceded by **l** or **L** to indicate that a pointer to a **double** or **long double** rather than to a **float** is in the argument list. The **l**, **L** or **h** modifier is ignored for other conversion characters.

The **scanf()** functions terminate their conversions at EOF, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

EXTERNAL INFLUENCES

Locale

The **LC_CTYPE** category determines the interpretation of ordinary characters within format strings as single and/or multi-byte characters. Field width is given in terms of bytes. Characters received from the input stream are interpreted as single- or multi-byte characters as determined by the **LC_TYPE** category and the field width is decremented by the length of the character.

The **LC_NUMERIC** category determines the radix character expected within floating-point numbers.

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUES

If the input ends before the first conflict or conversion, EOF is returned. Otherwise, these functions return the number of successfully assigned input items. This number is a short count, or even zero if a conflict ensues between an input character and the control string.

ERRORS

scanf(), **fscanf()**, **nl_scanf()**, and **nl_fscanf()** fail if data needs to be read into the *stream*'s buffer, and:

[EAGAIN] The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the process would be delayed in the read operation.

[EBADF] The file descriptor underlying *stream* is not a valid file descriptor open for reading.

[EINTR] The read operation was terminated due to the receipt of a signal, and either no data was transferred or the implementation does not report partial transfer for this file.

[EIO] The process is a member of a background process and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the **SIGTTIN** signal or the process group of the process is orphaned.

Additional **errno** values can be set by the underlying **read()** function (see *read*(2)).

EXAMPLES

The call:

```
int i, n; float x; char name[50];
n = scanf("%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E-1 thompson
```

assigns to *n* the value 3, to *i* the value 25, to *x* the value 5.432, and *name* contains **thompson\0**. Or:

```
int i; float x; char name[50];
(void) scanf("%2d%f%*d %[0-9]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

assigns 56 to *i*, 789.0 to *x*, skips 0123, and places the string 56\0 in *name*. The next call to **getchar()** (see *getc*(3S)) returns **a**.

For another example, to create a language-independent date scanning routine, use:

```
char month[20]; int day, year;
(void) scanf(format, month, &day, &year);
```

For American usage, *format* would point to a string:

```
%1$s %2$d %3$d
```

The input:

```
July 3 1986
```

would assign `July` to *month*, `3` to *day* and `1986` to *year*.

For German usage, *format* would point to a string:

```
%2$d %1$s %3$d
```

The input:

3 Juli 1986

would assign `Juli` to *month*, `3` to *day* and `1986` to *year*.

The success of literal matches and suppressed assignments can be determined with the `%n` conversion specification. Here is an example that checks the success of literal matches:

```
int i, n1, n2, n3, n4;
n1 = n2 = n3 = n4 = -1;"
scanf( "%nBEGIN%n %d %nEND%n", &n1, &n2, &i, &n3, &n4);
if (n2 - n1 == 5) puts( "matched BEGIN");
if (n4 - n3 == 3) puts( "matched END");
```

Here is an example that checks the success of suppressed assignments:

```
int i, n1, n2;
n1 = n2 = -1;
scanf( "%d %n%*s%n", &i, &n1, &n2);
if (n2 > n1)
    printf( "successful assignment suppression of %d chars\n", n2 - n1);
```

WARNINGS

Trailing white space (including a newline) is left unread unless matched in the control string.

Truncation of multi-byte characters may occur if a field width is used with the conversion character.

`nl_scanf()`, `nl_fscanf()`, and `nl_sscanf()` are provided for historical reasons only. Their use is not recommended. Use `scanf()`, `fscanf()`, and `sscanf()` instead.

AUTHOR

`scanf()` was developed by AT&T and HP.

SEE ALSO

getc(3S), setlocale(3C), printf(3S), strtod(3C), strtol(3C).

STANDARDS CONFORMANCE

`scanf()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
`fscanf()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`nl_fscanf()`: XPG2
`nl_scanf()`: XPG2

`nl_sscanf()`: XPG2
`sscanf()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

setaclentry(), fsetaclentry() - add, modify, or delete one entry in file's access control list (ACL)

SYNOPSIS

```
#include <unistd.h>
#include <acllib.h>

int setaclentry(const char *path, int uid, int gid, int mode);

int fsetaclentry(int fd, int uid, int gid, int mode);
```

DESCRIPTION

Both forms of this call add, modify, or delete one entry in a file's access control list (ACL). **setaclentry()** and **fsetaclentry()** take a path name (*path*) or open file descriptor (*fd*) and an entry identifier (*uid*, *gid*). They change the indicated entry's access mode bits to the given value (*mode*), meanings of which are defined in **<unistd.h>**. *modes* are represented as **R_OK**, **W_OK**, and **X_OK**. Irrelevant bits in *mode* values must be zero.

If the file's ACL does not have an entry for the given *uid* and *gid*, the entry is created and added to the ACL. If *mode* is **MODE_DEL** (defined in **<acllib.h>**), the matching entry is deleted from the file's ACL if it is an optional entry, or its mode bits are set to zero (no access) if it is a base entry.

uid or *gid* can be **ACL_NSUSER** or **ACL_NSGROUP** (defined in **<sys/acl.h>**), respectively, to represent non-specific entries *u*.%, %.*g*, or %.% . The file's *u*.% or %.*g* base entries can be referred to using **ACL_FILEOWNER** or **ACL_FILEGROUP** (defined in **<acllib.h>**), for the file's owner or group ID, respectively.

setaclentry() and **fsetaclentry()** read the file's ACL with **getacl()** or **fgetacl()** and modify it with **setacl()** or **fsetacl()**, respectively.

RETURN VALUE

If successful, **setaclentry()** and **fsetaclentry()** return zero.

ERRORS

If an error occurs, **setaclentry()** and **fsetaclentry()** return the following negative values and set **errno**:

–1 Unable to perform **getacl()** or **fgetacl()** on the file. **errno** indicates the cause.

-2 Unable to perform **stat()** or **fstat()** on the file. **errno** indicates the cause.

-3 Cannot add a new entry because the ACL already has **NACLENTRIES** (defined in **<sys/acl.h>**) entries.

-4 Cannot delete a nonexisting entry.

-5 Unable to perform **setacl()** or **fsetacl()** on the file. **errno** indicates the cause.

EXAMPLES

The following code fragment adds an entry to file "work/list" for user ID 115, group ID 32, or modifies the existing entry for that user and group, if any, with a new access mode of read only. It also changes the owner base entry to have all access rights, and deletes the entry, if any, for any user in group 109.

```
#include <unistd.h>
#include <acllib.h>

char *filename = "work/list";

setaclentry (filename, 115, 32, R_OK);
setaclentry (filename, ACL_FILEOWNER, ACL_NSGROUP, R_OK | W_OK | X_OK);
setaclentry (filename, ACL_NSUSER, 109, MODE_DEL);
```

DEPENDENCIES

NFS **setaclentry()** and **fsetaclentry()** are not supported on remote files.

AUTHOR

setaclentry() and **fsetaclentry()** were developed by HP.

SEE ALSO

getacl(2), setacl(2), stat(2), acltostr(3C), cpacl(3C), chownacl(3C), strtoacl(3C), acl(5).

NAME

setbuf(), setvbuf() - assign buffering to a stream file

SYNOPSIS

```
#include <stdio.h>

void setbuf(FILE *stream, char *buf);

int setvbuf(FILE *stream, char *buf, int type, size_t size);
```

DESCRIPTION

`setbuf()` can be used after a stream has been opened but before it is read or written. It causes the array pointed to by *buf* to be used instead of an automatically allocated buffer. If *buf* is the NULL pointer input/output will be completely unbuffered.

A constant `BUFSIZ`, defined in the `<stdio.h>` header file, tells how big an array is needed:

```
char buf[BUFSIZ];
```

`setvbuf()` can be used after a stream has been opened but before it is read or written. *type* determines how *stream* is to be buffered. Legal values for *type* (defined in `<stdio.h>`) are:

`_IOFBF` causes input/output to be fully buffered.

`_IOLBF` causes output to be line buffered; the buffer will be flushed when a newline is written, the buffer is full, or input is requested.

`_IONBF` causes input/output to be completely unbuffered.

When an output stream is unbuffered, information is queued for writing on the destination file or terminal as soon as written; when it is buffered, many characters are saved up and written as a block. When the output stream is line-buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a new-line character is written or terminal input is requested). `fflush()` can also be used to explicitly write the buffer.

If *buf* is not the NULL pointer, the array it points to is used for buffering instead of an automatically allocated buffer (from `malloc()`). *size* specifies the size of the buffer to be used. The constant `BUFSIZ` in `<stdio.h>` is suggested as a good buffer size. If input/output is unbuffered, *buf* and *size* are ignored.

By default, output to a terminal is line buffered and all other input/output is fully buffered.

SEE ALSO

fopen(3S), getc(3S), malloc(3C), putc(3S), stdio(3S).

DIAGNOSTICS

If an illegal value for *type* or *size* is provided, `setvbuf()` returns a non-zero value. Otherwise, the value returned will be zero.

NOTE

A common source of error is allocating buffer space as an "automatic" variable in a code block, then failing to close the stream in the same block.

STANDARDS CONFORMANCE

`setbuf()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`setvbuf()`: AES, SVID2, XPG2, XPG3, XPG4, ANSI C

NAME

setclock - set value of system-wide clock

SYNOPSIS

```
#include <sys/timers.h>

int setclock(int clock_type, struct timespec *tp);
```

DESCRIPTION

`setclock()` sets the current value *tp* of the specified system-wide clock, *clock_type*.

`setclock()` supports a *clock_type* of **TIMEOFDAY**, defined in <sys/timers.h>, which represents the time-of-day clock for the system. For this clock, the values returned by `setclock()` represent the amount of time since the Epoch.

The calling process must have appropriate privileges to set the **TIMEOFDAY** clock.

RETURN VALUE

`setclock()` returns a value of zero if successful; otherwise it returns –1 and sets `errno` to indicate the error.

ERRORS

setclock() fails if any of the following conditions are encountered:

[EINVAL]	*clock_type* does not specify a known system-wide clock, or *tp* either is outside the range for a given clock type, or specifies a nanosecond value less than zero or greater than or equal to 1000 million.
[EIO]	An error occurred while accessing the clock device
[EPERM]	The requesting process does not have the required appropriate privileges to set the specified clock.

SEE ALSO

gettimer(3C), getclock(3C), <sys/timers.h>

STANDARDS CONFORMANCE

`setclock()`: AES

NAME

setjmp(), longjmp(), sigsetjmp(), siglongjmp() - non-local goto

SYNOPSIS

```
#include <setjmp.h>

int setjmp(jmp_buf env);

void longjmp(jmp_buf env, int val);

int _setjmp(jmp_buf env);

void _longjmp(jmp_buf env, int val);

int sigsetjmp(sigjmp_buf env, int savemask);

void siglongjmp(sigjmp_buf env, int val);
```

DESCRIPTION

`setjmp()` and `longjmp()` are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program. They exist in three variant forms: `setjmp()` and `longjmp()`; `_setjmp()` and `_longjmp()`; `sigsetjmp()` and `siglongjmp()`. Unless indicated otherwise, references to `setjmp()` and `longjmp()` apply to all three versions.

`setjmp()` saves its stack environment in *env* (whose type, `jmp_buf`, is defined in the `<setjmp.h>` header file) for later use by `longjmp()`. It returns the value `0`.

`longjmp()` restores the environment saved by the last call of `setjmp()` with the corresponding *env* argument. After `longjmp()` is completed, program execution continues as if the corresponding call of `setjmp()` (which must not itself have returned in the interim) had just returned the value *val*. `longjmp()` cannot cause `setjmp()` to return the value `0`. If `longjmp()` is invoked with a second argument of `0`, `setjmp()` returns `1`. All accessible data values are valid as of the time `longjmp()` is called.

Upon the return from a `setjmp()` call caused by a `longjmp()`, the values of any non-static local variables belonging to the routine from which `setjmp()` was called are undefined. Code which depends on such values is not guaranteed to be portable.

Variant Forms

The following functions behave the same as `setjmp()` and `longjmp()` except in the handling of the process' signal mask (see *sigaction*(2) and *sigvector*(2)). This distinction is only significant for programs which use `sigaction()`, `sigprocmask()`, `sigvector()`, `sigblock()`, and/or `sigsetmask()`.

`setjmp()`
`longjmp()` These always save and restore the signal mask.

`_setjmp()`
`_longjmp()` These never manipulate the signal mask.

`sigsetjmp()` Saves the signal mask if and only if *savemask* is non-zero.

`siglongjmp()` Restores the signal mask if and only if it is saved by `sigsetjmp()`.

Programming Considerations

If a `longjmp()` is executed and the environment in which the `setjmp()` is executed no longer exists, errors can occur. The conditions under which the environment of the `setjmp()` no longer exists include exiting the procedure that contains the `setjmp()` call, and exiting an inner block with temporary storage (such as a block with declarations in C or a `with` statement in Pascal). This condition might not be detectable, in which case the `longjmp()` occurs and, if the environment no longer exists, the contents of the temporary storage of an inner block are unpredictable. This condition might also cause unexpected process termination. If the procedure has been exited the results are unpredictable.

Passing `longjmp()` a pointer to a buffer not created by `setjmp()`, passing `_longjmp()` a pointer to a buffer not created by either `setjmp()` or `_setjmp()`, passing `siglongjmp()` a pointer to a buffer not created by `sigsetjmp()` or passing any of these three functions a buffer that has been modified by the user, can cause all the problems listed above, and more.

Some implementations of Pascal support a "try/recover" mechanism, which also creates stack marker information. If a `longjmp()` operation occurs in a scope which is nested inside a try/recover, and the corresponding `setjmp()` is not inside the scope of the try/recover, the recover block will not be executed and the currently active recover block will become the one enclosing the `setjmp()`, if one exists.

WARNINGS

A call to `longjmp()` to leave the guaranteed stack space reserved by `sigspace()` might remove the guarantee that the ordinary execution of the program will not extend into the guaranteed space. It might also cause the program to forever lose its ability to automatically increase the stack size, and the program might then be limited to the guaranteed space.

The result of using `setjmp()` within an expression can be unpredictable.

If `longjmp()` is called even though *env* was never primed by a call to `setjmp()`, or when the last such call was in a function that has since returned, total chaos is guaranteed.

AUTHOR

`setjmp()` was developed by AT&T and HP.

SEE ALSO

sigaction(2), sigblock(2), signal(5), sigprocmask(2), sigsetmask(2), sigspace(2), sigsuspend(2), sigvector(2).

STANDARDS CONFORMANCE

`setjmp()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
`longjmp()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`siglongjmp()`: AES, XPG3, XPG4, FIPS 151-2, POSIX.1
`sigsetjmp()`: AES, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

setlocale(), getlocale() - set and get the locale of a program

SYNOPSIS

```
#include <locale.h>

char *setlocale(int category, const char *locale);

struct locale_data *getlocale(int type);
```

DESCRIPTION

`setlocale()` sets, queries or restores that aspect of a program's locale as specified by the *category* argument. A program's locale refers to those areas of the program's Native Language Support (NLS) environment for which the following values of *category* have been defined:

`LC_ALL`	Affects behavior of all categories below as well as all *nl_langinfo*(3C) items. Note that some *nl_langinfo* items are only affected by the setting of the `LC_ALL` category.
`LC_COLLATE`	Affects behavior of regular expressions and the NLS string collation functions (see *string*(3C) and *regexp*(5)).
`LC_CTYPE`	Affects behavior of regular expressions, character classification and conversion functions (see *ctype*(3C), *conv*(3C), and *regexp*(5)). The `LC_CTYPE` category also affects the behavior of all routines that process multibyte characters (see *multibyte*(3C) and *nl_tools_16*(3C)).
`LC_MESSAGES`	Affects the language in which messages are displayed and the processing of affirmative and negative responses.
`LC_MONETARY`	Affects behavior of functions that handle monetary values (see *localeconv*(3C)).
`LC_NUMERIC`	Affects handling of the radix character in the formatted input/output functions (see *printf*(3C), *scanf*(3C) and *vprintf*(3C)) and the string conversion functions (see *ecvt*(3C) and *strtod*(3C)). `LC_NUMERIC` also affects the numeric values found in the *localeconv* structure.
`LC_TIME`	Affects the behavior of time conversion functions (see *getdate*(3C) and *strftime*(3C)).

All *nl_langinfo*(3C) items are affected by the setting of one of the categories listed above. See *langinfo*(5) to determine which categories affect each item.

The value of the *locale* argument determines the action taken by `setlocale()`. *locale* is a pointer to a character string.

Setting the Locale of a Program

To set the program's locale for *category*, `setlocale()` accepts one of the following values as the *locale* argument: *locale name*, "C", or "" (the empty string). The actions prescribed by these values are as follows :

locale name	If *locale* is a valid locale name (see *lang(5)*), `setlocale()` sets that part of the NLS environment associated with *category* as defined for that locale.
"C"	If the value of *locale* is set to "C", `setlocale()` sets that part of the NLS environment associated with *category* as defined for the "C" locale (see *lang(5)*). The "C" locale is the default prior to successfully calling `setlocale()`.
POSIX	Same as "C"
""	If the value of *locale* is the empty string, the setting of that part of the NLS environment associated with *category* depends on the setting of the following environment variables in the user's environment (see *environ*(5)):

`LANG`	`LC_MESSAGES`
`LC_ALL`	`LC_MONETARY`
`LC_COLLATE`	`LC_NUMERIC`
`LC_CTYPE`	`LC_TIME`

If *category* is any defined value other than LC_ALL, setlocale() sets that category as specified by the value of the LC_ALL environment or if LC_ALL is not set to the corresponding environment variable. If the environment variable is not set or set to the empty string, setlocale() sets the category as specified by the value of the LANG environment variable. If LANG is not set or is set to the empty string, then setlocale() sets the category to the "C" locale. For example, setlocale(LC_TIME,"") sets the program's NLS environment associated with the LC_TIME *category* to the value specified by the user's LC_TIME environment variable. All other aspects of the NLS environment are unaffected.

If *category* is LC_ALL, then all categories are set corresponding to the value of LC_ALL if LC_ALL is set, or LANG if LC_ALL is not set, except for those categories in which the corresponding environment variable is set to a valid language name (see *lang(5)*). In this case the value of the environment variable overrides the values of LC_ALL and LANG for that category. If the values of both LC_ALL and LANG are not set or are set to the empty string, then the "C" locale is used.

The following usage of setlocale() results in the program's locale being set according to the the user's language requirements:

```
setlocale(LC_ALL,"");
```

Querying the Locale of a Program

setlocale() queries the current NLS environment pertaining to *category* if the value of *locale* is NULL. The query operation does not change the environment. The purpose of performing a query is to save that aspect of the user's current NLS environment associated with *category*, in the value returned by setlocale(), such that it can be restored with a subsequent call to setlocale().

Restoring the Locale of a Program

To restore a category within the program locale, a setlocale() call is made with the same *category* argument and the return string of the previous setlocale() call given as the *locale* argument.

getlocale() returns a pointer to a locale_data structure (see /usr/include/locale.h). The members of the locale_data structure contain information about the setting of each setlocale category. *type* determines what information is contained in the locale_data structure. Defined values of *type* and their behaviour are :

LOCALE_STATUS	The structure member corresponding to each category contains a string with the name of the locale currently set for that category. The string does not include modifier information.
MODIFIER_STATUS	The structure member corresponding to each category contains a string with the name of the modifier currently set for that category. If no modifier is set then the entry contains an empty string.
ERROR_STATUS	The structure member contains information about errors which occurred during the previous call to setlocale(). If setlocale() could not satisfy a request corresponding to a particular *category*, the structure member for that category contains a string with the name of the invalid locale. In all other cases the member for the category contains an empty string.

RETURN VALUE

If the pointer to a string is given for *locale* and the selection can be honored, the setlocale() function returns a pointer to the string associated with the specified *category* for the new locale. The maximum length of this string is LC_BUFSIZ bytes (see <locale.h>). If the selection cannot be honored, the setlocale() function returns a null pointer and the program's locale is not changed.

A null pointer for *locale* causes setlocale() to return a string associated with the *category* for the program's current locale.

The string returned by setlocale() is such that a subsequent call with that string as the *locale* argument and its associated *category* restores that part of the program's locale.

ERRORS

If a language name given through the *locale* argument does not identify a valid language name or the

language is not available on the system (see *lang(5)*) a null pointer is returned and the program's locale is not changed. The same behavior occurs when the call :

```
setlocale(LC_ALL, "");
```

is made and any category related environment variable in the user's environment identifies an invalid language name or a language that is not available on the system.

If the *category* argument is not a defined category value, a null pointer is returned and the program's locale is not changed.

`setlocale()` returns a string which reflects the current setting of that aspect of the NLS environment corresponding to the *category* argument. If this return string is used in a subsequent `setlocale()` call and the *category* arguments of the two calls do not match, the locale remains unchanged and a null pointer is returned.

WARNINGS

Using `getenv()` as the *locale* argument is not recommended. An example of such incorrect usage is :

```
setlocale(LC_ALL, getenv("LANG"));
```

`getenv()` returns a character string which can be a language name, an empty string, or a null pointer; depending on the setting of the user's `LANG` environment variable. Each of these values as the *locale* argument define a specific action to be taken by `setlocale()`. Therefore the action taken by `setlocale()` depends upon the value returned from the `getenv()` call. To ensure that `setlocale()` sets the program's locale based upon the setting of the user's environment variables the following usage is recommended:

```
setlocale(LC_ALL, "");
```

The value returned by `setlocale()` points to a static area that is overwritten during the next call to `setlocale()`. Be sure to copy these values to another area if they are to be used after a subsequent `setlocale()` call.

The structure returned through a call to `getlocale()` is overwritten during the next call to `getlocale()`. Be sure to save these values if they are to be used after a subsequent `getlocale()` call.

Any program which calls `setlocale()` before `catopen()` may behave differently in this release than on previous releases because of the addition of `LC_MESSAGES` to `XPG4` . In the past, `catopen()` was directed to the desired language by `LANG` . Now, `catopen()` is controlled by `LC_MESSAGES` . `Setlocale()` can modify the `LC_MESSAGES` category.

For example, if the environment variables are set as follows :

```
LC_MESSAGES="french"
```

and the following call to `setlocale()` is made:

```
setlocale(LC_ALL, "german");
```

which is followed by a call to `catopen()` . `Catopen()` will open the message catalogs for `german` rather than `french`.

EXAMPLES

To set a program's entire locale based on the language requirements specified via the user's environment variables :

```
setlocale(LC_ALL, "");
```

If, in the previous example, the user's environment variables were set as follows :

```
LANG ="german"
LC_COLLATE ="spanish@nofold"
LC_MONETARY =""
LC_TIME ="american"
```

the `LC_ALL`, `LC_CTYPE`, `LC_MONETARY`, and `LC_NUMERIC` category items would be set to correspond to the `german` language definition, the `LC_COLLATE` category items would be set to correspond to the `spanish` language definition for unfolded collation (see *hpnls*(5)) and the `LC_TIME` category items would be set corresponding to the `american` language definition.

Using the same example, if the following call was made:

```
struct locale_data *locale_info=getlocale(LOCALE_STATUS);
```

the contents of `*locale_info` would be :

```
locale_info->LC_ALL_D="german"
locale_info->LC_COLLATE_D="spanish"
locale_info->LC_CTYPE_D="german"
locale_info->LC_MESSAGES_D="german"
locale_info->LC_MONETARY_D="german"
locale_info->LC_NUMERIC_D="german"
locale_info->LC_TIME_D="american"
```

Continuing with the same example, if the following call was made :

```
struct locale_data *modifier_info=getlocale(MODIFIER_STATUS);
```

the contents of *modifier_info would now be :

```
modifier_info->LC_ALL_D=""
modifier_info->LC_COLLATE_D="nofold"
modifier_info->LC_CTYPE_D=""
modifier_info->LC_MESSAGES_D=""
modifier_info->LC_MONETARY_D=""
modifier_info->LC_NUMERIC_D=""
modifier_info->LC_TIME_D=""
```

The calls :

```
setlocale(LC_ALL, "");
struct locale_data *error_info=getlocale(ERROR_STATUS);
```

with the following settings in the users environment :

```
LANG=german
LC_COLLATE=junk
```

where `junk` is an invalid language, would result in the contents of `*error_info` being:

```
_error_info->LC_ALL_D=""
_error_info->LC_COLLATE_D="junk"
_error_info->LC_CTYPE_D=""
_error_info->LC_MESSAGES_D=""
_error_info->LC_MONETARY_D=""
_error_info->LC_NUMERIC_D=""
_error_info->LC_TIME_D=""
```

An example showing the precedence of the `LC_ALL` environment variable :

```
setlocale(LC_ALL, "");
```

with the following settings in the users environment :

```
LANG=german
LC_ALL=french
```

All categories will be loaded with `french.`

Another example showing the precedence of the `LC_ALL` environment variable :

```
setlocale(LC_CTYPE, "");
```

with the following settings in the users environment :

```
LANG=turkish
LC_ALL=danish
LC_CTYPE=russian
```

The `LC_CTYPE` category will be loaded with `danish.`

Another example with the LC_ALL environment variable :

```
setlocale(LC_TIME, "polish");
```

with the following settings in the users environment :

```
LANG=italian
LC_ALL=dutch
```

The LC_TIME category will be set to polish.

To set the date/time formats to French :

```
setlocale(LC_TIME, "french");
```

To set the collating sequence to the "C" locale :

```
setlocale( LC_COLLATE, "C");
```

To set monetary handling to the value of the user's LC_MONETARY environment variable :

```
setlocale( LC_MONETARY, "");
```

(Note that if the LC_MONETARY environment variable is not set or empty, the value of the user's LANG environment variable is used.)

To query a user's locale:

```
char *ch = setlocale( LC_ALL, NULL);
```

To restore the locale saved in the above example :

```
setlocale(LC_ALL, ch);
```

To query just that part of the user's locale pertaining to the LC_NUMERIC category :

```
char *ch = setlocale(LC_NUMERIC, NULL);
```

To restore the LC_NUMERIC category of the user's locale saved in the above example :

```
setlocale(LC_NUMERIC, ch);
```

AUTHOR

setlocale() was developed by HP.

SEE ALSO

nlsinfo(1), buildlang(1M), conv(3C), ctype(3C), ecvt(3C), getdate(3C), langinfo(3C), multibyte(3C), nl_tools_16(3C), printf(3S), scanf(3S), strcoll(3C), strftime(3C), string(3C), strtod(3C), vprintf(3S), wconv(3X), wctype(3X), wstring(3X), hpnls(5), environ(5), langinfo(5), strerror(3C), <langinfo.h>, <locale.h>.

STANDARDS CONFORMANCE

setlocale(): AES, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

shl_load(), shl_definesym(), shl_findsym(), shl_gethandle(), shl_getsymbols(), shl_unload(), shl_get() - explicit load of shared libraries

SYNOPSIS

```
#include <dl.h>

shl_t shl_load(const char *path, int flags, long address);

int shl_findsym(
    shl_t *handle,
    const char *sym,
    short type,
    void *value
);

int shl_definesym(
    const char *sym,
    short type,
    long value,
    int flags
);
int shl_getsymbols(
    shl_t handle,
    short type,
    int flags,
    void *(*memory) (),
    struct shl_symbol **symbols,
);

int shl_unload(shl_t handle);

int shl_get(int index, struct shl_descriptor **desc);

int shl_gethandle(shl_t handle, struct shl_descriptor **desc);
```

DESCRIPTION

These routines can be used to programmatically load and unload shared libraries, and to obtain information about the libraries (such as the addresses of symbols defined within them). The routines themselves are accessed by specifying the `-ldld` option on the command line with the `cc` or `ld` command (see *cc*(1) and *ld*(1)). In addition, the `-E` option to the `ld` command can be used to ensure that all symbols defined in the program are available to the loaded libraries.

Shared libraries are created by compiling source files with the `+z` (position-independent code) option, and linking the resultant object files with the `-b` (create shared library) option.

`shl_load()` Attaches the shared library named by *path* to the process. The library is mapped at the specified *address*. If *address* is `0L`, the system chooses an appropriate address for the library. This is the recommended practice because the system has the most complete knowledge of the address space (see DEPENDENCIES). The flags argument is made up of several fields. One of the following must be specified:

`BIND_IMMEDIATE`	Resolve symbol references when the library is loaded.
`BIND_DEFERRED`	Delay code symbol resolution until actual reference.

Zero or more of the following can be specified by doing a bitwise OR operation:

`BIND_FIRST`	Place the library at the head of the symbol search order.
`BIND_NONFATAL`	Default `BIND_IMMEDIATE` behavior is to treat all unsatisfied symbols as fatal. This flag allows binding of unsatisfied code symbols to be deferred until use.
`BIND_NOSTART`	Do not call the initializer for the shared library when the library is loaded, nor on a future call to `shl_unload()`.

BIND_VERBOSE Print verbose messages concerning possible unsatisfied symbols.

If successful, **shl_load()** returns a handle which can be used in subsequent calls to **shl_findsym()**, **shl_unload()**, or **shl_gethandle()**; otherwise NULL is returned.

shl_findsym()
Obtains the address of an exported symbol *sym* from a shared library. The *handle* argument should be a pointer to the handle of a loaded shared library that was returned from a previous call to **shl_load()** or **shl_get()**. If a pointer to NULL is passed for this argument, **shl_findsym()** searches all currently loaded shared libraries to find the symbol; otherwise **shl_findsym()** searches only the specified shared library. The return value of handle will be NULL if the symbol found was generated via **shl_definesym()**. Otherwise the handle of the library where the symbol was found is returned. The special handle **PROG_HANDLE** can be used to refer to the program itself, so that symbols exported from the program can also be accessed dynamically. The *type* argument specifies the expected type for the symbol, and should be one of the defined constants **TYPE_PROCEDURE**, **TYPE_DATA**, or **TYPE_UNDEFINED**. The latter value suppresses type checking. The address of the symbol is returned in the variable pointed to by *value*. If a shared library contains multiple versions of the requested symbol, the latest version is returned. This routine returns 0 if successful; otherwise –1 is returned.

shl_definesym()
Adds a symbol to the shared library symbol table for the current process making it the most visible definition. If the *value* falls in the range of a currently loaded library, an association will be made and the symbol is undefined once the associated library is unloaded. The defined symbol can be overridden by a subsequent call to this routine or by loading a more visible library that provides a definition. Symbols overridden in this manner may become visible again if the overriding definition is removed.

Possible symbol types include:

TYPE_PROCEDURE Symbol is a function.

TYPE_DATA Symbol is data.

Possible flag values include: None defined at the present time. Zero should be passed in to prevent conflicts with future uses of this flag.

shl_getsymbols()
Provides an array of symbol records, allocated using the supplied memory allocator, that are associated with the library specified by *handle*. If the *handle* argument is a pointer to NULL, symbols defined using **shl_definesym()** are returned. If multiple versions of the same symbol have been defined within a library or with **shl_definesym()**, only the version from the specified symbol information source that would be considered for symbol binding is returned. The *type* argument is used to restrict the return information to a specific type. Values of **TYPE_PROCEDURE** and **TYPE_DATA** can be used to limit the returned symbols to be either code or data respectively. The constant **TYPE_UNDEFINED** can be used to return all symbols, regardless of type. The *flags* argument must have one of the following values:

IMPORT_SYMBOLS
Return symbols found on the import list.

EXPORT_SYMBOLS
Return symbols found on the export list. All symbols defined via **shl_definesym()** are export symbols.

Zero or more of the following can be specified by doing a bitwise OR operation:

NO_VALUES Only makes sense when combined with **EXPORT_SYMBOLS**. Do not calculate the value field in the return structure to avoid symbol binding by the loader to resolve symbol dependencies. If only a few symbol values are needed, **shl_findsym()** can be used to find the values of interesting symbols. Not to be used with **GLOBAL_VALUES**.

GLOBAL_VALUES
Only makes sense when combined with **EXPORT_SYMBOLS**. Use the name and

type information of each return symbol and find the most visible occurrence using all symbol information sources. The value and handle fields in the symbol return structure reflect where the most visible occurrence was found. Not to be used with NO_VALUES.

The *memory* argument should point to a function with the same interface as `malloc()` (see *malloc*(3C)).

The return information consists of an array of the following records (defined in <dl.h>):

```
struct shl_symbol {
    char *name,
    short type,
    void  value,
    shl_t handle,
};
```

The *type* field in the return structure can have the values TYPE_PROCEDURE, TYPE_DATA, or TYPE_STORAGE, where TYPE_STORAGE is a subset of TYPE_DATA. The *value* and *handle* fields are only valid if export symbols are requested and the NO_VALUES flag is not specified. The *value* field contains the address of the symbol, while the *handle* field is the handle of the library that defined the symbol, or NULL for symbols defined via the `shl_definesym()` routine and is useful in conjunction with the GLOBAL_VALUES flag.

If successful, `shl_getsymbols()` returns the number of symbols found; otherwise it returns –1.

`shl_unload()`
Can be used to detach a shared library from the process. The *handle* argument should be the handle returned from a previous call to `shl_load()`. `shl_unload()` returns 0 if successful; otherwise –1 is returned. All explicitly loaded libraries are detached automatically on process termination.

`shl_get()`
Returns information about currently loaded libraries, including those loaded implicitly at startup time. The index argument is the ordinal position of the shared library in the shared library search list for the process. A subsequent call to `shl_unload()` decrements the index values of all libraries having an index greater than the unloaded library. The index value –1 refers to the dynamic loader. The *desc* argument is used to return a pointer to a statically allocated buffer containing a descriptor for the shared library. The format of the descriptor is implementation dependent; to examine its format, look at the contents of file `/usr/include/dl.h`. Information common to all implementations includes the library handle, pathname, and the range of addresses the library occupies. The buffer for the descriptor used by `shl_get()` is static; the contents should be copied elsewhere before a subsequent call to the routine. The routine returns 0 normally, or –1 if an invalid *index* is given.

`shl_gethandle()`
Returns information about the library specified by the handle argument. The special handle PROG_HANDLE can be used to refer to the program itself. The descriptor returned is the same as the one returned by the `shl_get()` routine. The buffer for the descriptor used by `shl_gethandle()` is static; the contents should be copied elsewhere before a subsequent call to the routine. The routine returns 0 normally, or –1 on error.

DIAGNOSTICS

If a library cannot be loaded, `shl_load()` returns NULL and sets `errno` to indicate the error. All other functions return –1 on error and set `errno`.

If `shl_findsym()` cannot find the indicated symbol, `errno` is set to zero. If `shl_findsym()` finds the indicated symbol but cannot resolve all the symbols it depends on, `errno` is set to ENOSYM.

If a call to `shl_load()` or `shl_findsym()` fails with ENOSYM, the process may be left in an inconsistent state. Some symbol resolutions may have occurred before the failure, and these may be invalid. The program should probably be terminated if this occurs.

ERRORS

Possible values for `errno` include:

[ENOEXEC]	The specified file is not a shared library, or a format error was detected.
[ENOSYM]	Some symbol required by the shared library could not be found.
[EINVAL]	The specified handle or index is not valid or an attempt was made to load a library at an invalid address.
[ENOMEM]	There is insufficient room in the address space to load the library.
[ENOENT]	The specified library does not exist.
[EACCES]	Read or execute permission is denied for the specified library.

WARNINGS

`shl_unload()` detaches the library from the process and frees the memory allocated for it, but does not break existing symbolic linkages into the library. In this respect, an unloaded shared library is much like a block of memory deallocated via `free()` (see *free*(3C)).

Some implementations may not, by default, export all symbols defined by a program (instead exporting only those symbols that are imported by a shared library seen at link time). Therefore the `-E` option to *ld*(1) should be used when using these routines if the loaded libraries are to refer to program symbols.

All symbol information returned by `shl_getsymbols()`, including the name field, become invalid once the associated library is unloaded by `shl_unload()`.

DEPENDENCIES

Series 300/400:

`shl_definesym()` and `shl_getsymbols()` are not implemented on Series 300 and 400 systems.

When using `shl_findsym()`, keep in mind that the compilers place an underscore at the beginning of all external names.

Series 700/800:

The only value for the *address* field is `0L`. Any other value is treated as if it had been specified as `0L`.

The following additional values for the *flags* argument can be used with `shl_load()` on Series 700 and 800 systems:

`BIND_RESTRICTED`
: Restrict symbols visible by the library to those present at library load time.

`DYNAMIC_PATH`
: Allow the loader to dynamically search for the library specified by the *path* argument. The directories to be searched are determined by the `+s` and `+b` options of the `ld` command used when the program was linked.

AUTHOR

shl_load(3X) and related functions were developed by HP.

SEE ALSO

ld(1), dld.sl(5).

NAME

sigemptyset(), sigfillset(), sigaddset(), sigdelset(), sigismember() - initialize, manipulate, and test signal sets

SYNOPSIS

```
#include <signal.h>

int sigemptyset(sigset_t *set);

int sigfillset(sigset_t *set);

int sigaddset(sigset_t *set, int signo);

int sigdelset(sigset_t *set, int signo);

int sigismember(const sigset_t *set, int signo);
```

DESCRIPTION

`sigemptyset()` initializes the signal set pointed to by *set,* to exclude all signals supported by HP-UX.

`sigfillset()` initializes the signal set pointed to by *set,* to include all signals supported by HP-UX.

Applications must call either `sigemptyset()` or `sigfillset()` at least once for each object of type `sigset_t` before using that object for anything else, including cases where the object is returned from a function (for example, the *oset* argument to `sigprocmask()` — see *sigprocmask*(2)).

`sigaddset()` adds the signal specified by *signo* to the signal set pointed to by *set*.

`sigdelset()` deletes the signal specified by *signo* from the signal set pointed to by *set.*

`sigismember()` tests whether the signal specified by *signo* is a member of the signal set pointed to by *set*.

RETURN VALUE

Upon successful completion, `sigismember()` returns a value of 1 if the specified signal is a member of the specified set, or a value of 0 if it is not. The other functions return a value of 0 upon successful completion. For all of the above functions, if an error is detected, a value of –1 is returned and `errno` is set to indicate the error.

ERRORS

`sigaddset()`, `sigdelset()`, and `sigismember()` fail if the following is true:

[EINVAL] The value of the *signo* argument is out of range. The reliable detection of this error is not guaranteed.

WARNINGS

The above functions do not detect a bad address passed in for the *set* argument. A segmentation fault is the most likely result.

AUTHOR

`sigfillset()`, `sigemptyset()`, `sigaddset()`, `sigdelset()`, and `sigismember()` were derived from the *IEEE Standard POSIX 1003.1-1988* .

SEE ALSO

sigaction(2), sigsuspend(2), sigpending(2), sigprocmask(2), signal(5).

STANDARDS CONFORMANCE

`sigaddset()`: AES, XPG3, XPG4, FIPS 151-2, POSIX.1
`sigdelset()`: AES, XPG3, XPG4, FIPS 151-2, POSIX.1

`sigemptyset()`: AES, XPG3, XPG4, FIPS 151-2, POSIX.1
`sigfillset()`: AES, XPG3, XPG4, FIPS 151-2, POSIX.1

`sigismember()`: AES, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

sinh(), cosh(), tanh(), sinhf(), coshf(), tanhf() - hyperbolic functions

SYNOPSIS

```
#include <math.h>

double sinh(double x);

double cosh(double x);

double tanh(double x);

float sinhf(float x);

float coshf(float x);

float tanhf(float x);
```

DESCRIPTION

`sinh()`, `cosh()`, and `tanh()` return respectively the hyberbolic sine, cosine, and tangent of their argument.

When x is ±INFINITY, `sinh()` returns ±INFINITY respectively.

When x is ±INFINITY, `cosh()` returns +INFINITY.

When x is ±INFINITY, `tanh()` returns ±1.0 respectively.

`sinhf()`, `coshf()`, and `tanhf()` are `float` versions of these functions. Their performance is significantly faster than that of the `double` versions. Programs must be compiled in ANSI mode (use the `-Aa` option) in order to use these functions; otherwise, the compiler promotes the `float` arguments to `double`, and the functions return incorrect results.

DEPENDENCIES

Series 300/400

`sinhf()`, `coshf()`, and `tanhf()` are not supported on Series 300/400 systems.

Series 700/800

`sinhf()`, `coshf()`, and `tanhf()` are not specified by any standard (they are, however, named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard). They are provided in the PA1.1 versions of the math library only. The `+DA1.1` option (the default on Series 700 systems) links in a PA1.1 version automatically. A PA1.1 library can be linked in explicitly. For more information, see the *HP-UX Floating-Point Guide*.

ERRORS

/lib/libm.a

`sinh()` and `cosh()` return HUGE_VAL (and `sinh()` may return –HUGE_VAL for negative x) and set `errno` to ERANGE when the correct value would overflow.

`sinh()`, `cosh()` and `tanh()` return NaN and set `errno` to EDOM when x is NaN. In addition, a message indicating DOMAIN error is printed on the standard error output.

These error-handling procedures can be changed with the `matherr()` function (see *matherr*(3M)).

/lib/libM.a

No error messages are printed on the standard error output.

`sinh()` and `cosh()` return HUGE_VAL (and `sinh()` may return –HUGE_VAL for negative x) and set `errno` to ERANGE when the correct value would overflow.

`sinh()`, `cosh()` and `tanh()` return NaN and set `errno` to EDOM when x is NaN.

These error-handling procedures can be changed by using the `_matherr()` function (see *_matherr*(3M)). Note that `_matherr()` is provided in order to assist in migrating programs from `libm.a` to `libM.a` and is *not* a part of XPG3, ANSI C, or POSIX.

SEE ALSO

isinf(3M), isnan(3M), matherr(3M).

STANDARDS CONFORMANCE

`sinh()` in libm.a: AES, SVID2, XPG2, XPG3, FIPS 151-2, POSIX.1

`sinh()` in libM.a: AES, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`cosh()` in libm.a: AES, SVID2, XPG2, XPG3, FIPS 151-2, POSIX.1
`cosh()` in libM.a: AES, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`tanh()` in libm.a: AES, SVID2, XPG2, XPG3, FIPS 151-2, POSIX.1
`tanh()` in libM.a: AES, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

sleep() - suspend execution for interval

SYNOPSIS

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

DESCRIPTION

`sleep()` suspends the current process from execution for the number of *seconds* specified by the argument.

Actual suspension time can be less than that requested for two reasons:

- Scheduled wakeups occur at fixed 1-second intervals (on the second, according to an internal clock), and
- Any caught signal terminates the *sleep* following execution of that signal's catching routine.

Suspension time can be an arbitrary amount longer than requested due to the scheduling of other activity in the system. The value returned by `sleep()` is the "unslept" amount (the requested time minus the time actually slept) in case the caller had an alarm set to go off earlier than the end of the requested `sleep()` time, or premature arousal due to another caught signal.

`sleep()` is implemented by setting an alarm signal and pausing until it (or some other signal) occurs. The previous state of the alarm signal is saved and restored. The calling program may have set up an alarm signal before calling `sleep()`. If the `sleep()` time exceeds the time until such an alarm signal, the process sleeps only until the alarm signal would have occurred. The caller's alarm catch routine is executed just before the `sleep()` routine returns. If the `sleep()` time is less than the time till such alarm, the prior alarm time is reset to go off at the same time it would have without the intervening `sleep()`.

seconds must be less than 2^{32}.

SEE ALSO

alarm(2), pause(2), signal(5).

STANDARDS CONFORMANCE

`sleep()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

spray - scatter data in order to check the network

SYNOPSIS

```
#include <time.h>
#include <rpcsvc/spray.h>
```

DESCRIPTION

This reference page describes the data structures and XDR routines used by the *spray*(1M) program. A `spray()` function call does not exist. Refer to *spray*(1M) for more information.

RPC Info

program number:
SPRAYPROG

xdr routines:

```
xdr_sprayarr(xdrs, arr);
        XDR *xdrs;
        struct sprayarr *arr;
xdr_spraycumul(xdrs, cumul);
        XDR *xdrs;
        struct spraycumul *cumul;
```

procs:

SPRAYPROC_SPRAY
Takes no arguments, returns no value.
Increments a counter in server daemon.
The server does not return this call,
so the caller should have a timeout of 0.
The sprayarr is only used by the caller,
to vary the size of the UDP packets sent.

SPRAYPROC_GET
Takes no arguments, returns *struct spraycumul*
with the values of counter and clock set to
reflect the number of SPRAYPROC_SPRAY calls,
and the total time (seconds and microseconds)
elapsed since the last SPRAYPROC_CLEAR request.

SPRAYPROC_CLEAR
Takes no arguments and returns no value.
Zeros out counter and clock in preparation
for calls to SPRAYPROC_SPRAY.

versions:
SPRAYVERS_ORIG

structures:

```
struct spraycumul {
        unsigned counter;
        struct timeval clock;
};
struct sprayarr {
        int *data;
        int lnth;
};
```

WARNING

User applications that call this routine must be linked with `/usr/include/librpcsvc.a`. For example,

```
cc my_source.c -lrpcsvc
```

AUTHOR

`spray` was developed by Sun Microsystems, Inc.

SEE ALSO

spray(1M), sprayd(1M).

INTERNATIONAL SUPPORT
8-bit data, 16-bit data, messages

NAME

sputl(), sgetl() - access long integer data in a machine-independent fashion

SYNOPSIS

```
#include <unistd.h>

void sputl(long int value, char *buffer);
```

DESCRIPTION

`sputl()` Take the four bytes of the long integer *value* and place them in memory starting at the address pointed to by *buffer*. The ordering of the bytes is the same across all machines.

`sgetl()` Retrieve the four bytes in memory starting at the address pointed to by *buffer* and return the long integer value in the byte ordering of the host machine.

The combination of `sputl()` and `sgetl()` provides a machine-independent way of storing long numeric data in a file in binary form without conversion to characters.

Any program that uses these functions must be loaded with the object-file access-routine library `libld.a`.

STANDARDS CONFORMANCE

`sputl()`: SVID2

`sgetl()`: SVID2

NAME

ssignal(), gsignal() - software signals

SYNOPSIS

```
#include <signal.h>

int (*ssignal(int sig, int (*action)(int)))(int);

int gsignal(int sig);
```

DESCRIPTION

`ssignal()` and `gsignal()` implement a software facility similar to *signal*(5). This facility is used by the Standard C Library to enable users to indicate the disposition of error conditions, and is also made available to users for their own purposes.

Software signals made available to users are associated with integers in the inclusive range 1 through 15. A call to `ssignal()` associates a procedure, *action*, with the software signal *sig*; the software signal, *sig*, is raised by a call to `gsignal()`. Raising a software signal causes the action established for that signal to be taken.

The first argument to `ssignal()` is a number identifying the type of signal for which an action is to be established. The second argument defines the action; it is either the name of a (user-defined) *action function* or one of the manifest constants `SIG_DFL` (default) or `SIG_IGN` (ignore). `ssignal()` returns the action previously established for that signal type; if no action has been established or the signal number is illegal, `ssignal()` returns `SIG_DFL`.

`gsignal()` raises the signal identified by its argument, *sig*:

- If an action function has been established for *sig*, that action is reset to `SIG_DFL` and the action function is entered with argument *sig*. `gsignal()` returns the value returned to it by the action function.
- If the action for *sig* is `SIG_IGN`, `gsignal()` returns the value 1 and takes no other action.
- If the action for *sig* is `SIG_DFL`, `gsignal()` returns the value 0 and takes no other action.
- If *sig* has an illegal value or no action was ever specified for *sig*, `gsignal()` returns the value 0 and takes no other action.

SEE ALSO

signal(5).

NOTES

Some additional signals with numbers outside the range 1 through 15 are used by the Standard C Library to indicate error conditions. Those signal numbers outside the range 1 through 15 are legal, although their use may interfere with the operation of the Standard C Library.

STANDARDS CONFORMANCE

`ssignal()`: SVID2, XPG2

`gsignal()`: SVID2, XPG2

NAME

statfsdev, fstatfsdev - get file system statistics

SYNOPSIS

```
#include <sys/vfs.h>

int statfsdev(const char *path, struct statfs *buf);

int fstatfsdev(int fildes, struct statfs *buf);
```

DESCRIPTION

statfsdev() returns information about the file system on the file specified by *path*.

buf is a pointer to a **statfs** structure into which information is placed concerning the file system. The contents of the structure pointed to by *buf* include the following members:

```
long  f_bavail    /* free blocks available to non-superuser */
long  f_bfree     /* free blocks */
long  f_blocks    /* total blocks in file system */
long  f_bsize     /* fundamental file system block size in bytes */
long  f_ffree     /* free file nodes in file system */
long  f_files     /* total file nodes in file system */
long  f_type      /* type of info, zero for now */
fsid_t  f_fsid    /* file system ID.  f_fsid[1] is MOUNT_UFS,
                                      MOUNT_NFS, or MOUNT_CDFS */
```

Fields that are undefined for a particular file system are set to –1 .

fstatfsdev() returns the same information as above, but about the open file referred to by file descriptor *fildes*.

RETURN VALUE

Upon successful completion, **statfsdev()** and **fstatfsdev()** return zero. Otherwise, they return -1 and set the global variable **errno** to indicate the error.

ERRORS

statfsdev() fails if one or more of the following conditions are encountered:

[EACCES] Search permission is denied for a component of the path prefix.

[EAGAIN] The file exists, enforcement mode file/record locking is set, and there are outstanding record locks on the file.

[EFAULT] *path* points to an invalid address.

[ELOOP] Too many symbolic links are encountered in translating the path name.

[EMFILE] The maximum number of file descriptors allowed are currently open.

[ENAMETOOLONG] The length of the specified path name exceeds **PATH_MAX** bytes, or the length of a component of the path name exceeds **NAME_MAX** bytes while **_POSIX_NO_TRUNC** is in effect.

[ENFILE] The system file table is full.

[ENOENT] The named file does not exist.

[ENOTDIR] A component of the path prefix is not a directory.

[ENXIO] The device specified by the named special file does not exist.

fstatfsdev() fails if one or more of the following is true:

[EBADF] *fildes* is not a valid open file descriptor.

[ESPIPE] *filedes* points to an invalid address.

Both **fstatfsdev()** and **statfsdev()** fail if one or more of the following is true:

[EAGAIN] Enforcement-mode record locking was set, and there was a blocking write lock.

[EDEADLK]	A resource deadlock would occur as a result of this operation.
[EINTR]	A system call was interrupted by a signal.
[EINVAL]	The file specified by *path* or *filedes* does not contain a file system of any known type.
[ENOLOCK]	The system lock table was full, so the read could not go to sleep until the blocking write lock was removed.

AUTHOR

`statfsdev()` and `fstatfsdev()` were developed by HP.

FILES

`/usr/include/sys/mount.h`

SEE ALSO

bdf(1M), df(1M), stat(2), statfs(2).

NAME

stdio() - standard buffered input/output stream file package

SYNOPSIS

```
#include <stdio.h>
```

DESCRIPTION

The Standard I/O functions described in the subsection (3S) entries of this manual constitute an efficient, user-level I/O buffering scheme. The `getc()` and `putc()` functions handle characters quickly. The following funtions all use or act as if they use `getc()` and `putc()`, and can be freely intermixed:

`fgetc()`	`fputs()`	`getchar()`	`putchar()`
`fgets()`	`fread()`	`gets()`	`puts()`
`fprintf()`	`fscanf()`	`getw()`	`putw()`
`fputc()`	`fwrite()`	`printf()`	`scanf()`

A file with associated buffering is called a *stream* and is declared to be a pointer to a defined type `FILE`. `fopen()` creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Section (3S) library routines operate on this stream.

At program startup, three streams, *standard input*, *standard output*, and *standard error*, are predefined and do not need to be explicitly opened. When opened, the standard input and standard output streams are fully buffered if the output refers to a file and line-buffered if the output refers to a terminal. The standard error output stream is by default unbuffered. These three streams have the following constant pointers declared in the `<stdio.h>` header file :

`stdin`	standard input file
`stdout`	standard output file
`stderr`	standard error file

A constant, `NULL`, (0) designates a nonexistent pointer.

An integer-constant, `EOF`, (-1) is returned upon end-of-file or error by most integer functions that deal with streams (see individual descriptions for details).

An integer constant `BUFSIZ` specifies the size of the buffers used by the particular implementation (see *setbuf*(3S)).

Any program that uses this package must include the header file of pertinent macro definitions as follows:

```
#include <stdio.h>
```

The functions and constants mentioned in subsection (3S) entries of this manual are declared in that header file and need no further declaration.

A constant `_NFILE` defines the default maximum number of open files allowed per process. To increase the open file limit beyond this default value, see *setrlimit*(2).

SEE ALSO

close(2), lseek(2), open(2), pipe(2), read(2), setrlimit(2), write(2), ctermid(3S), cuserid(3S), fclose(3S), ferror(3S), fgetpos(3S), fileno(3S), fopen(3S), fread(3S), fseek(3S), fsetpos(3S), getc(3S), gets(3S), popen(3S), printf(3S), putc(3S), puts(3S), scanf(3S), setbuf(3S), system(3S), tmpfile(3S), tmpnam(3S), ungetc(3S).

ERRORS

Invalid *stream* pointers usually cause grave disorder, possibly including program termination. Individual function descriptions describe the possible error conditions.

STANDARDS CONFORMANCE

`stderr`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`stdin`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`stdout`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

ftok() - standard interprocess communication package

SYNOPSIS

```
#include <sys/ipc.h>

key_t ftok(const char *path, int id);
```

DESCRIPTION

All interprocess communication facilities require the user to supply a key to be used by the **msgget()**, **semget()**, and **shmget()** system calls to obtain interprocess communication identifiers (see *msgget*(2), *semget*(2), and *shmget*(2)). One suggested method for forming a key is to use the **ftok()** routine described below. Another way to compose keys is to include the project ID in the most significant byte, and use the remaining portion as a sequence number. There are many other ways to form keys, but it is necessary for each system to define standards for forming them. If some standard is not adhered to, it will be possible for unrelated processes to unintentionally interfere with each other's operation. Therefore, it is strongly suggested that the most significant byte of a key in some sense refer to a project so that keys do not conflict across a given system.

ftok() returns a key based on *path* and *id* that is usable in subsequent **msgget()**, **semget()**, and **shmget()** system calls. *path* must be the path name of an existing file that is accessible to the process. *id* is a character that uniquely identifies a project. Note that **ftok()** returns the same key for linked files when called with the same *id* and that it returns different keys when called with the same file name but different *id*s.

RETURN VALUE

ftok() returns (**key_t**) – 1 if *path* does not exist or if it is not accessible to the process.

EXAMPLES

The following call to **ftok**() returns a key associated with the file *myfile* and id **A**:

```
key_t mykey;
      mykey = ftok ("myfile", 'A');
```

WARNINGS

If the file whose *path* is passed to **ftok()** is removed when keys still refer to the file, future calls to **ftok()** with the same *path* and *id* will return an error. If the same file is recreated, **ftok()** is likely to return a different key than it did the original time it was called.

In an HP Clustered environment, **ftok()** can return a different key (using the same file name) when executed on different members of the cluster if any component of the file path name is a context-dependent file.

SEE ALSO

intro(2), msgget(2), semget(2), shmget(2), cdf(4).

NAME

strftime() - convert date and time to string

SYNOPSIS

```
#include <time.h>

size_t strftime(
    char *s,
    size_t maxsize,
    const char *format,
    const struct tm *timeptr
);
```

DESCRIPTION

`strftime()` converts the contents of a `tm` structure (see *ctime*(3C)) to a formatted date and time string.

`strftime()` places characters into the array pointed to by *s* as controlled by the string pointed to by *format*. The *format* string consists of zero or more directives and ordinary characters. A directive consists of a `%` character, an optional field width and precision specification, and a terminating character that determines the directive's behavior. All ordinary characters (including the terminating null character are copied unchanged into the array. No more than *maxsize* characters are placed into the array. Each directive is replaced by the appropriate characters as described in the following list. The appropriate characters are determined by the program's locale, by the values contained in the structure pointed to by *timeptr*, and by the `TZ` environment variable (see External Influences below).

Directives

The following directives, shown without the optional field width and precision specification, are replaced by the indicated characters:

`%a`	Locale's abbreviated weekday name.
`%A`	Locale's full weekday name.
`%b`	Locale's abbreviated month name.
`%B`	Locale's full month name.
`%c`	Locale's appropriate date and time representation.
`%d`	Day of the month as a decimal number [01,31].
`%E`	Locale's combined Emperor/Era name and year.
`%H`	Hour (24-hour clock) as a decimal number [00,23].
`%I`	Hour (12-hour clock) as a decimal number [01,12].
`%j`	Day of the year as a decimal number [001,366].
`%m`	Month as a decimal number [01,12].
`%M`	Minute as a decimal number [00,59].
`%n`	New-line character.
`%N`	Locale's Emperor/Era name.
`%o`	Locale's Emperor/Era year.
`%p`	Locale's equivalent of either AM or PM.
`%S`	Second as a decimal number [00,61].
`%t`	Tab character.
`%U`	Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.
`%w`	Weekday as a decimal number [0(Sunday),6].
`%W`	Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.
`%x`	Locale's appropriate date representation.
`%X`	Locale's appropriate time representation.
`%y`	Year without century as a decimal number [00,99].
`%Y`	Year with century as a decimal number.
`%Z`	Time zone name (or by no characters if no time zone exists).
`%%`	%

The following directives are provided for backward compatibility with the directives supported by *date*(1) and the *ctime*(3C) functions. It is recommended that the directives above be used in preference to those below.

%D	Date in usual U.S. format (%m/%d/%y) (use **%x** instead).
%F	Locale's full month name (use **%B** instead).
%h	Locale's abbreviated month name (use **%b** instead).
%r	Time in 12-hour U.S. format (%I:%M:%S [AM|PM]) (use **%X** instead).
%T	Time in 24-hour U.S. format (%H:%M:%S) (use **%X** instead).
%z	Time zone name (or by no characters if no time zone exists) (use **%Z** instead).

If a directive is not one of the above, the behavior is undefined.

Field Width and Precision

An optional field width and precision specification can immediately follow the initial **%** of a directive in the following order:

[-|0]*w*	the decimal digit string *w* specifies a minimum field width in which the result of the conversion is right- or left-justified. It is right-justified (with space padding) by default. If the optional flag '-' is specified, it is left-justified with space padding on the right. If the optional flag '0' is specified, it is right-justified and padded with zeros on the left.
.*p*	the decimal digit string *p* specifies the minimum number of digits to appear for the **d**, **H**, **I**, **j**, **m**, **M**, **o**, **S**, **U**, **w**, **W**, **y** and **Y** directives, and the maximum number of characters to be used from the **a**, **A**, **b**, **B**, **c**, **D**, **E**, **F**, **h**, **n**, **N**, **p**, **r**, **t**, **T**, **x**, **X**, **z**, **Z**, and **%** directives. In the first case, if a directive supplies fewer digits than specified by the precision, it will be expanded with leading zeros. In the second case, if a directive supplies more characters than specified by the precision, excess characters will truncated on the right.

If no field width or precision is specified for a **d**, **H**, **I**, **m**, **M**, **S**, **U**, **W**, **y**, or **j** directive, a default of **.2** is used for all but **j** for which **.3** is used.

EXTERNAL INFLUENCES

Locale

The **LC_TIME** category determines the characters to be substituted for those directives described above as being from the locale.

The **LC_CTYPE** category determines the interpretation of the bytes within *format* as single and/or multi-byte characters.

The **LC_NUMERIC** category determines the characters used to form numbers for those directives that produce numbers in the output. If **ALT_DIGITS** (see *langinfo*(5)) is defined for the locale, the characters so specified are used in place of the default ASCII characters.

Environment Variables

TZ determines the time zone name substituted for the **%Z** and **%z** directives. The time zone name is determined by calling the function **tzset()** which sets the external variable **tzname** (see *ctime*(3C)).

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUE

If the total number of resulting characters including the terminating null character is not more than **maxsize**, **strftime()** returns the number of characters placed into the array pointed to by *s*, not including the terminating null character. Otherwise, zero is returned and the contents of the array are indeterminate.

EXAMPLES

If the *timeptr* argument contains the following values:

```
timeptr→tm_sec = 4;
timeptr→tm_min = 9;
timeptr→tm_hour = 15;
timeptr→tm_mday = 4;
timeptr→tm_mon = 6;
```

```
timeptr→tm_year = 88;
timeptr→tm_wday = 1;
timeptr→tm_yday = 185;
timeptr→tm_isdst = 1;
```

the following combinations of the **LC_TIME** category and format strings produce the indicated output:

LC_TIME	format string	output
american	%x	Mon, Jul 4, 1988
german	%x	Mo., 4. Juli 1988
american	%X	03:09:04 PM
french	%X	15h09 04
any†	%H:%M:%S	15:09:04
any†	%.1H:%.1M:%.1S	15:9:4
any†	%2.1H:%-3M:%03.1S	15:9 :004

† The directives used in these examples are not affected by the **LC_TIME** category of the locale.

WARNINGS

If the arguments *s* and *format* are defined such that they overlap, the behavior is undefined.

The function **tzset()** is called upon every invocation of **strftime()** (whether or not the time zone name is copied to the output array).

The range of values for **%S** ([0,61]) extends to 61 to allow for the occasional one or two leap seconds. However, the system does not accumulate leap seconds and the **tm** structure generated by the functions **localtime()** and **gmtime()** (see *ctime*(3C)) never reflects any leap seconds.

Results are undefined if values contained in the structure pointed to by *timeptr* exceed the ranges defined for the **tm** structure (see *ctime*(3C)) or are not consistent (such as if the **tm_yday** element is set to 0, indicating the first day of January, while the **tm_mon** element is set to 11, indicating a day in December).

AUTHOR

strftime() was developed by HP.

SEE ALSO

date(1), ctime(3C), getdate(3C), setlocale(3C), environ(5), langinfo(5), hpnls(5).

STANDARDS CONFORMANCE

strftime(): AES, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

strcat(), strncat(), strcmp(), strncmp(), strcasecmp(), strncasecmp(), strcpy(), strncpy(), strdup(), strlen(), strchr(), strrchr(), strpbrk(), strspn(), strcspn(), strstr(), strrstr(), strtok(), strcoll(), strxfrm(), nl_strcmp(), nl_strncmp(), index(), rindex() - character string operations

SYNOPSIS

```
#include <string.h>
#include <strings.h>

char *strcat(char *s1, const char *s2);

char *strncat(char *s1, const char *s2, size_t n);

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

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

int strcasecmp(const char *s1, const char *s2);

int strncasecmp(const char *s1, const char *s2, size_t n);

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

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

char *strdup(const char *s);

size_t strlen(const char *s);

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

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

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

size_t strspn(const char *s1, const char *s2);

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

char *strstr(const char *s1, const char *s2);

char *strrstr(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);

int nl_strcmp(const char *s1, const char *s2);

int nl_strncmp(const char *s1, const char *s2, size_t n);

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

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

Remarks:

All functions except **index()** and **rindex()** are declared in both headers, so only one of the two headers needs to be included.

The functions **index()** and **rindex()** are declared only in **<strings.h>**, They and **<strings.h>** are provided solely for portability of BSD applications, and are not recommended for new applications where portability is important. For portable applications, use **<string.h>**, **strchr()**, and **strrchr()** instead.

index() and **rindex()** and **<strings.h>** are provided solely for portability of BSD applications, and are not recommended for new applications where portability is important. For portable applications, use **strchr()** and **strrchr()** instead.

DESCRIPTION

Arguments *s1*, *s2*, and *s* point to strings (arrays of characters terminated by a null byte).

Definitions for all these functions, the type `size_t`, and the constant `NULL` are provided in the **<string.h>** header.

`strcat()` Appends a copy of string *s2* to the end of string *s1*. `strncat()` appends a maximum of *n* characters. It copies fewer if *s2* is shorter than *n* characters. Each returns a pointer to the null-terminated result (the value of *s1*).

`strcmp()` Compares its arguments and returns an integer less than, equal to, or greater than zero, depending on whether *s1* is lexicographically less than, equal to, or greater than *s2*. The comparison of corresponding characters is done as if the type of the characters were `unsigned char`. Null pointer values for *s1* and *s2* are treated the same as pointers to empty strings. `strncmp()` makes the same comparison but examines a maximum of *n* characters (*n* less than or equal to zero yields equality). `strcasecmp()` and `strncasecmp()` are identical in function to `strcmp()` and `strncmp()` respectively, but characters are folded by `_tolower()` (see *conv*(3C)) prior to comparison. The returned lexicographic difference reflects the folding to lowercase.

`strcpy()` Copies string *s2* to *s1*, stopping after the null byte has been copied. `strncpy()` copies exactly *n* characters, truncating *s2* or adding null bytes to *s1* if necessary, until a total of *n* have been written. The result is not null-terminated if the length of *s2* is *n* or more. Each function returns *s1*. Note that should not be used to copy *n* bytes of an arbitrary structure. If that structure contains a null byte anywhere, `strncpy()` copies fewer than *n* bytes from the source to the destination and fills the remainder with null bytes. Use the `memcpy()` function (see *memory*(3C)) to copy arbitrary binary data.

`strdup()` Returns a pointer to a new string which is a duplicate of the string to which *s1* points. The space for the new string is obtained using the `malloc()` function (see *malloc*(3C)).

`strlen()` Returns the number of characters in *s*, not including the terminating null byte.

`strchr()` (`strrchr()`) Returns a pointer to the first (last) occurrence of character *c* in string *s*, or a null pointer if *c* does not occur in the string. The null byte terminating a string is considered to be part of the string. `index()` (`rindex()`) is identical to `strchr()` (`strrchr()`), and is provided solely for portability of BSD applications.

`strpbrk()` Returns a pointer to the first occurrence in string *s1* of any character from string *s2*, or a null pointer if no character from *s2* exists in *s1*.

`strspn()` (`strcspn()`) Returns the length of the maximum initial segment of string *s1*, which consists entirely of characters from (not from) string *s2*.

`strstr()` (`strrstr()`) Returns a pointer to the first (last) occurrence of string *s2* in string *s1*, or a NULL pointer if *s2* does not occur in the string. If *s2* points to a string of zero length, `strstr()` (`strrstr()`) returns *s1*.

`strtok()` Considers the string *s1* to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *s2*. The first call (with a non-null pointer *s1* specified) returns a pointer to the first character of the first token, and writes a null byte into *s1* immediately following the returned token. The function keeps track of its position in the string *s1* between separate calls, so that subsequent calls made with the first argument a null pointer work through the string immediately following that token. In this way subsequent calls work through the string *s1* until no tokens remain. The separator string *s2* can be different from call to call. When no token remains in *s1*, a null pointer is returned.

`strcoll()` Returns an integer greater than, equal to, or less than zero, according to whether the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2*. The comparison is based on strings interpreted as appropriate to the program's locale (see Locale below). In the "C" locale `strcoll()` works like `strcmp()`. `nl_strcmp()` is provided for historical reasons only and is equivalent to `strcoll()`. `nl_strncmp()`, also provided only for historical reasons, makes the same comparisons as `strcoll()`, but looks at a maximum of *n* characters (*n* less than or equal to zero yields equality).

`strxfrm()` Transforms the string pointed to by *s2* and places the resulting string into the array pointed to by *s1*. The transformation is such that if the `strcmp()` function is applied to

two transformed strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the **strcoll()** function applied to the same two original strings. No more than *n* bytes are placed into the resulting string, including the terminating null character. If the transformed string fits in no more than *n* bytes, the length of the resulting string is returned (not including the terminating null character). Otherwise the return value is the number of bytes that the *s1* string would occupy (not including the terminating null character), and the contents of the array are indeterminate.

strcoll() has better performance with respect to **strxfrm()** in cases where a given string is compared to other strings only a few times, or where the strings to be compared are long but a difference in the strings that determines their relative ordering usually comes among the first few characters. **strxfrm()** offers better performance in, for example, a sorting routine where a number of strings are each transformed just once and the transformed versions are compared against each other many times.

EXTERNAL INFLUENCES

Locale

The **LC_CTYPE** category determines the interpretation of the bytes within the string arguments to the **strcoll()**, **strxfrm()**, **nl_strcmp()**, and **nl_strncmp()** functions as single and/or multi-byte characters. It also determines the case conversions to be done for the **strcasecmp()** and **strncasecmp()** functions.

The **LC_COLLATE** category determines the collation ordering used by the **strcoll()**, **strxfrm()**, **nl_strcmp()**, and **nl_strncmp()** functions. See *hpnls*(5) for a description of supported collation features. Use **nlsinfo** (see *nlsinfo*(1)) to view the collation used for a particular locale.

International Code Set Support

Single- and multi-byte character code sets are supported for the **strcoll()**, **strxfrm()**, **nl_strcmp()**, and **nl_strncmp()** functions. All other functions support only single-byte character code sets.

WARNINGS

The functions **strcat()**, **strncat()**, **strcpy()**, **strncpy()**, and **strtok()** alter the contents of the array to which *s1* points. They do not check for overflow of the array.

Null pointers for destination strings cause undefined behavior.

Character movement is performed differently in different implementations, so moves involving overlapping source and destination strings may yield surprises.

The transformed string produced by **strxfrm()** for a language using an 8-bit code set is usually at least twice as large as the original string and may be as much four times as large (ordinary characters occupy two bytes each in the transformed string, 1-to-2 characters four bytes, 2-to-1 characters two bytes per original pair, and don't-care characters no bytes). Each character of a multi-byte code set (Asian languages) occupies three bytes in the transformed string.

For functions **strcoll()**, **strxfrm()**, **nl_strcmp()**, and **nl_strncmp()**, results are undefined if the languages specified by the **LC_COLLATE** and **LC_CTYPE** categories use different code sets.

AUTHOR

string was developed by AT&T, HP, and the University of California, Berkeley.

SEE ALSO

nlsinfo(1), conv(3C), malloc(3C), malloc(3X), memory(3C), setlocale(3C), hpnls(5).

STANDARDS CONFORMANCE

nl_strcmp(): XPG2
nl_strncmp(): XPG2

strcat(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
strchr(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

strcmp(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
strcoll(): AES, XPG3, XPG4, ANSI C

strcpy(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
strcspn(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

strdup(): SVID2
strlen(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

strncat(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
strncmp(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

strncpy(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
strpbrk(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

strrchr(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
strspn(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

strstr(): AES, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C
strtok(): AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

strxfrm(): AES, XPG3, XPG4, ANSI C

NAME

strord - convert string data order

SYNOPSIS

```
#include <nl_types.h>

char *strord(char *s1, const char *s2, nl_mode m);
```

DESCRIPTION

The text orientation (mode) of a file can be right-to-left (non-Latin) or left-to-right (Latin). This text orientation can affect the way data is arranged in the file. The data arrangements that result are called screen order and keyboard order (see *hpnls*(5) for more details).

`strord()` converts the order of characters in *s2* from screen to keyboard order or vice versa and places the result in *s1*. The arguments *s1* and *s2* point to strings (arrays of characters terminated by a null character). `strord()` returns *s1*.

`strord()` performs the conversion based on mode information indicated by the argument *m*. The argument *m* is of type *nl_mode* found in the header file <nl_types.h>. The mode argument can have two possible values: **NL_LATIN** and **NL_NONLATIN**.

If the mode argument is **NL_LATIN**, the text orientation is left-to-right, and all non-Latin sub-strings are reversed. Non-Latin sub-strings are any number of contiguous right-to-left language characters. Non-Latin sub-strings are delimited by ASCII characters.

Similarly, if the mode argument is **NL_NONLATIN**, the text orientation is right-to-left and all Latin sub-strings are reversed. Latin sub-strings are any number of contiguous printable ASCII characters. Latin sub-strings are delimited by right-to-left language characters and ASCII control codes.

Some right-to-left languages have a duplicate set of digits called alternative numbers. Alternative numbers always have a left-to-right orientation.

EXTERNAL INFLUENCES

Locale

The `LC_NUMERIC` category determines whether a right-to-left language has alternative numbers.

International Code Set Support

Single-byte character code sets are supported.

WARNINGS

`strord()` does not check for overflow of the array pointed to by *s1*.

AUTHOR

`strord()` was developed by HP.

SEE ALSO

nl_init(3C), hpnls(5), environ(5), forder(1), nljust(1).

NAME

strtoacl(), strtoaclpatt(), aclentrystart() - convert exact or pattern string form to access control list (ACL) structure

SYNOPSIS

```
#include <acllib.h>

int strtoacl(
    const char *string,
    int nentries,
    int maxentries,
    struct acl_entry *acl,
    int fuid,
    int fgid
);

int strtoaclpatt(
    const char *string,
    int maxentries,
    struct acl_entry_patt *acl
);
```

DESCRIPTION

`strtoacl()` converts an access control list from exact symbolic (string) representation to structure form. It parses the input string and verifies its validity. Optionally it applies the entries in the string as a series of changes to an existing ACL.

`strtoaclpatt()` converts an access control list pattern from symbolic (string) representation to structure form. It parses the input string and verifies its validity.

The external array `aclentrystart[]`, only valid until the next call of either routine, is useful for error reporting. See ERRORS below.

The "operator" and "short" symbolic forms of ACLs and ACL patterns (described in *acl*(5)) are acceptable as input strings. If the first non-whitespace character in *string* is `(`, the ACL or ACL pattern in *string* must be in short form. Otherwise operator form is assumed.

`strtoacl()` takes a pointer to the string to be converted, and a pointer to the first element of an array of ACL entries (`acl[]`) initially containing the indicated number (*nentries*) of valid entries (zero or more). This array can grow to the indicated number of entries (*maxentries*). `strtoacl()` also takes file user ID (*fuid*) and group ID (*fgid*) values to substitute for `@` characters in *string* and returns the resulting number of entries in `acl[]`.

Redundant entries (identical user ID and group ID values after processing `@` characters) are combined, so that `acl[]` contains unique entries in the order encountered. If a new entry is mentioned, it is added to the end of the `acl` array.

strtoaclpatt()

`strtoaclpatt()` differs from `strtoacl()` because it processes an ACL pattern instead of an ACL. Since modification of an existing initial ACL is not useful, it is not supported.

Entries with matching user and group ID values are not combined. Each entry input yields one entry in the returned array.

The `@` character for user and group IDs (see *acl*(5)) is converted to special values (`ACL_FILEOWNER` or `ACL_FILEGROUP`, respectively, defined in <`acllib.h`>), not to specific user or group names provided by the caller. Thus, `strtoaclpatt()` need not be called to reparse the ACL pattern for each file, but the caller must handle the special values when comparing an ACL pattern to an ACL.

Wildcard user names, group names, and mode values are supported, as are absent mode parts; see *acl*(5).

`strtoaclpatt()` returns a different structure than `strtoacl()`. The *acl_entry_patt* structure contains *onmode* and *offmode* masks rather than a single *mode* value.

In operator form input, operators have a different effect on `strtoaclpatt()`:

= Sets bits in both the *onmode* and *offmode* fields appropriately, replacing existing bits in the entry, including any set by earlier operators.

\+ Sets bits in *onmode* and clears the same bits in *offmode*.

\- Sets bits in *offmode* and clears the same bits in *onmode*.

In short form input, the mode is treated like the = operator in operator form.

For both routines, a non-specific user or group ID of `%` is converted to **ACL_NSUSER** or **ACL_NSGROUP**, respectively. For `strtoaclpatt()` only, a wildcard user or group ID of `*` is converted to **ACL_ANYUSER** or **ACL_ANYGROUP**, respectively. The values are defined in <acllib.h>.

Entries can appear in *string* in any order. *string* can contain redundant entries, and in operator form only, redundant + and - operators for ACL entry mode modifications (in exact form) or mode bit inclusions/exclusions (in patterns). Entries or modifications are applied left to right.

Suggested Use

To build a new ACL (ACL pattern) array using `strtoacl()` (`strtoaclpatt()`), define `acl[]` with as many entries as desired. Pass it to `strtoacl()` (`strtoaclpatt()`) with *nentries* set to zero (`strtoacl()` only) and *maxentries* set to the number of elements in `acl[]`.

To have `strtoacl()` modify a file's existing ACL, define `acl[]` with the maximum possible number of entries (**NACLENTRIES**; see <sys/acl.h>). Call `getacl()` (see *getacl*(2)) to read the file's ACL and `stat()` (see *stat*(2)) to get the file's owner and group IDs. Then pass the current number of entries, the current ACL, and the ID values to `strtoacl()` with *maxentries* set to **NACLENTRIES**.

If `strtoacl()` succeeds, the resulting ACL can be passed safely to `setacl()` (see *setacl*(2)) because all redundancies (if any) have been resolved. However, note that since neither `strtoacl()` nor `strtoaclpatt()` validate user and group ID values, if the values are not acceptable to the system, `setacl()` fails.

Performance Trick

Normally `strtoacl()` replaces user and group names of `@` with specific user and group ID values, and also combines redundant entries. Therefore, calling `stat()` and `strtoacl()` for each of a series of files to which an ACL is being applied is simplest, although time consuming.

If *string* contains no `@` character, or if the caller merely wants to compare one ACL against another (and will handle the special case itself), it is sufficient to call `strtoacl()` once, and pointless to call `stat()` for each file. To determine this, call `strtoacl()` the first time with *fuid* set to **ACL_FILEOWNER** and *fgid* set to **ACL_FILEGROUP**. Repeated calls with file-specific *fuid* and *fgid* values are needed only if the special values of *fuid* and *fgid* appear in `acl[]` and the caller needs an exact ACL to set on each file; see EXAMPLES below.

If `@` appears in *string* and `acl[]` will be used later for a call to `setacl()`, it is necessary to call `strtoacl()` again to reparse the ACL string for each file. It is possible that not all redundant entries were combined the first time because the `@` names were not resolved to specific IDs. This also complicates comparisons between two ACLs. Furthermore, the caller cannot do the combining later because operator information from operator form input might be lost.

RETURN VALUE

If `strtoacl()` (`strtoaclpatt()`) succeeds, it returns the number of entries in the resulting ACL (ACL pattern), always equal to or greater than *nentries* (zero).

`strtoaclpatt()` also sets values in global array `aclentrystart[]` to point to the start of each pattern entry it parsed in *string*, in some cases including leading or trailing whitespace. It only sets a number of pointers equal to its return value plus one (never more than **NACLENTRIES** + 1). The last valid element points to the null character at the end of *string*. After calling `strtoaclpatt()`, an entry pattern's corresponding input string can be used by the caller for error reporting by (temporarily) putting a null at the start of the next entry pattern in *string*.

ERRORS

If an error occurs, `strtoacl()` and `strtoaclpatt()` return a negative value and the content of `acl` is undefined (was probably altered). To help with error reporting in this case, `aclentrystart[0]` and `aclentrystart[1]` are set to point to the start of the current and next entries, respectively, being parsed when the error occurred. If the current entry does not start with (, `aclentrystart[1]` points

to the next null character or comma at or after `aclentrystart[0]`. Otherwise, it points to the next null, or to the character following the next).

The following values are returned in case of error:

-1 Syntax error: entry doesn't start with (as expected in short form.

-2 Syntax error: entry doesn't end with) as expected in short form.

-3 Syntax error: user name is not terminated by a dot.

-4 (`strtoacl()` only) Syntax error: group name is not terminated by an operator in operator-form input or a comma in short-form input.

-5 Syntax error: user name is null.

-6 Syntax error: group name is null.

-7 Invalid user name (not found in `/etc/passwd` file and not a valid number).

-8 Invalid group name (not found in `/etc/group` file and not a valid number).

-9 Syntax error: invalid mode character, other than `0..7`, `r`, `w`, `x`, - (allowed in short form only), `*` (allowed in patterns only), , (to end an entry in operator form), or) (to end an entry in short form). Or, `0..7` or `*` is followed by other mode characters.

-10 The resulting ACL would have more than *maxentries* entries.

EXAMPLES

The following code fragment converts an ACL from a string to an array of entries using an *fuid* of 103 for the file's owner and *fgid* of 45 for the file's group.

```
#include <acllib.h>

int nentries;
struct acl_entry acl [NACLENTRIES];

if ((nentries = strtoacl (string, 0, NACLENTRIES, acl, 103, 45)) < 0)
     error (...);
```

The following code gets the ACL, *fuid*, and *fgid* for file `../myfile`, modifies the ACL using a description string, and changes the ACL on file `../myfile2` to be the new version.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <acllib.h>

struct stat statbuf;
int nentries;
struct acl_entry acl [NACLENTRIES];

if (stat ("../myfile", & statbuf) < 0)
     error (...);

if ((nentries = getacl ("../myfile", NACLENTRIES, acl)) < 0)
     error (...);

if ((nentries = strtoacl (string, nentries, NACLENTRIES, acl,
    statbuf.st_uid, statbuf.st_gid)) < 0)
{
    error (...);
}

if (setacl ("../myfile2", nentries, acl) < 0)
     error (...);
```

The following code fragment calls `strtoacl()` with special values of *fuid* and *fgid*, then checks to see if they show up in `acl[]`.

```
#include <acllib.h>
```

```
int  perfile = 0;  /* need to stat() and reparse per file? */
int  entry;

if ((nentries = strtoacl (string, 0, NACLENTRIES, acl,
     ACL_FILEOWNER, ACL_FILEGROUP)) < 0)
{
     error (...);
}

for (entry = 0; entry < nentries; entry++)
{
     if ((acl [entry] . uid == ACL_FILEOWNER)
      || (acl [entry] . gid == ACL_FILEGROUP))
     {
             perfile = 1;
             break;
     }
}
```

The following code fragment converts an ACL pattern from a string to an array of pattern entries.

```
#include <acllib.h>

int nentries;
struct acl_entry_patt acl [NACLENTRIES];

if ((nentries = strtoaclpatt (string, NACLENTRIES, acl)) < 0)
     error (...);
```

The following code fragment inside a `for` loop checks an entry pattern (`p*`, `onmask`, and `offmask` variable names) against an entry in a file's ACL (`a*` variable names) using the file's user and group IDs (`f*` variable names).

```
include <unistd.h>
if (((puid == ACL_FILEOWNER) && (fuid != auid))
 || ((puid != ACL_ANYUSER)   && (puid != auid)))
{
    continue;
}

if (((pgid == ACL_FILEGROUP) && (fgid != agid))
 || ((pgid != ACL_ANYGROUP)  && (pgid != agid)))
{
    continue;
}

if ((((  amode) & MODEMASK & onmask ) != onmask)
 || (((~ amode) & MODEMASK & offmask) != offmask))
{
    continue;
}
```

AUTHOR

`strtoacl()` and `strtoaclpatt()` were developed by HP.

FILES

`/etc/passwd`
`/etc/group`

SEE ALSO

getacl(2), setacl(2), acltostr(3C), cpacl(3C), chownacl(3C), setaclentry(3C), acl(5).

NAME

strtod, atof, nl_strtod, nl_atof - convert string to double-precision number

SYNOPSIS

```
#include <stdlib.h>

double strtod(const char *str, char **ptr);

double atof(const char *str);

double nl_strtod(const char *str, char **ptr, int langid);

double nl_atof(const char *str, int langid);
```

DESCRIPTION

`strtod()` returns, as a double-precision floating-point number, the value represented by the character string pointed to by *str*. The string is scanned (leading white-space characters as defined by `isspace()` in *ctype*(3C) are ignored) up to the first unrecognized character. If no conversion can take place, zero is returned.

`strtod()` recognizes characters in the following sequence:

1. An optional string of "white-space" characters which are ignored,
2. An optional sign,
3. A string of digits optionally containing a radix character,
4. An optional `e` or `E` followed by an optional sign or space, followed by an integer.

The radix character is determined by the loaded NLS environment (see *setlocale*(3C)). If `setlocale()` has not been called successfully, the default NLS environment, "C", is used (see *lang*(5)). The default environment specifies a period (.) as the radix character.

If the value of *ptr* is not `(char **)NULL` , the variable to which it points is set to point at the character after the last number, if any, that was recognized. If no number can be formed, **ptr* is set to *str*, and zero is returned.

`atof(`*str*`)` is equivalent to `strtod (`*str*`, (char **)NULL)`.

`nl_strtod()` and `nl_atof()` are similar to the above routines, but first call `langinit()` (see *nl_init*(3C)) to load the NLS environment specified by `langid`.

EXTERNAL INFLUENCES

Locale

The `LC_NUMERIC` category determines the value of the radix character within the currently loaded NLS environment.

RETURN VALUE

If the correct value would cause overflow, `+HUGE_VAL` or `-HUGE_VAL` is returned (according to the sign of the value), and `errno` is set to ERANGE.

If the correct value would cause underflow, zero is returned and `errno` is set to ERANGE.

WARNINGS

`nl_strtod()` and `nl_atof()` are provided for historical reasons only. Their use is not recommended. Use `strtod()` and `atof()` instead.

AUTHOR

`strtod()` was developed by AT&T and HP.

SEE ALSO

ctype(3C), setlocale(3C), scanf(3S), strtol(3C), hpnls(5), lang(5).

STANDARDS CONFORMANCE

`strtod()`: AES, SVID2, XPG2, XPG3, XPG4, ANSI C

`atof()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

strtol, atol, atoi, strtoul - convert string to integer

SYNOPSIS

```
#include <stdlib.h>

long strtol(const char *str, char **ptr, int base);

long atol(const char *str);

int atoi(const char *str);

unsigned long strtoul(const char *str, char **ptr, int base);
```

DESCRIPTION

`strtol()` (`strtoul()`) converts the character string pointed to by *str* to `long int` (`unsigned long int`) representation. The string is scanned up to the first character inconsistent with the base. Leading "white-space" characters (as defined by `isspace()` in *ctype*(3C)) are ignored. If no conversion can take place, zero is returned.

If *base* is greater than or equal to 2 and less than or equal to 36, it is used as the base for conversion. After an optional leading sign, leading zeros are ignored, and `0x` or `0X` is ignored if *base* is 16.

If *base* is zero, the string itself determines the base as follows: After an optional leading sign, a leading zero indicates octal conversion; a leading `0x` or `0X` hexadecimal conversion. Otherwise, decimal conversion is used.

If the value of *ptr* is not `(char **)NULL`, a pointer to the character terminating the scan is returned in the location pointed to by *ptr*. If no integer can be formed, the location pointed to by *ptr* is set to *str*, and zero is returned.

`atol(`*str*`)` is equivalent to `strtol(`*str*`, (char **)NULL, 10)`.

`atoi(`*str*`)` is equivalent to `int strtol(`*str*`, (char **)NULL, 10)`.

RETURN VALUE

Upon successful completion, all functions return the converted value, if any. If the correct value would cause overflow, `strtol()` returns `LONG_MAX` or `LONG_MIN` (according to the sign of the value), and sets `errno` to ERANGE; `strtoul()` returns `ULONG_MAX` and sets `errno` to ERANGE. Overflow conditions are ignored by `atol()` and `atoi()`.

For all other errors, zero is returned and `errno` is set to indicate the error.

ERRORS

`strtol()` and `strtoul()` fail and `errno` is set if any of the following conditions are encountered:

[EINVAL]	The value of *base* is not supported.
[ERANGE]	The value to be returned would have caused overflow.

SEE ALSO

ctype(3C), strtod(3C), scanf(3S).

STANDARDS CONFORMANCE

`strtol()`: AES, SVID2, XPG2, XPG3, XPG4, ANSI C

`atoi()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`atol()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`strtoul()`: AES, XPG4, ANSI C

NAME

strtold() - convert string to long double-precision number

SYNOPSIS

```
#include <stdlib.h>

long_double strtold(const char *str, char **ptr);
```

DESCRIPTION

`strtold()` returns as a long double-precision number the value represented by the character string pointed to by *str*. The string is scanned up to the first unrecognized character.

`strtold()` recognizes an optional string of "white-space" characters (as defined by `isspace()` in *ctype*(3C)), then an optional sign, then a string of digits optionally containing a radix character, then an optional `e` or `E` followed by an optional sign or space, followed by an integer. The radix character is determined by the loaded NLS environment (see *nl_init*(3C)). If `nl_init()` has not been called successfully, the default NLS environment, "C" (see *lang*(5)), is used. The default environment specifies a period (`.`) as the radix character.

If the value of *ptr* is not (`char **`)NULL, the variable to which it points is set to point at the character after the last number, if any, that was recognized. If no number can be formed, **ptr* is set to *str*, and zero is returned.

EXTERNAL INFLUENCES

International Code Set Support

Single-byte character code sets are supported.

RETURN VALUE

If the correct value would cause overflow, `+_MAXLDBL` or `-_MAXLDBL` is returned (according to the sign of the value), and `errno` is set to ERANGE.

If the correct value would cause underflow, zero is returned and `errno` is set to ERANGE.

AUTHOR

`strtold()` was developed by HP.

SEE ALSO

ctype(3C), nl_init(3C), scanf(3S), hpnls(5), lang(5).

NAME

swab() - swap bytes

SYNOPSIS

```
#include <unistd.h>

void swab(const void *from, void *to, ssize_t nbytes);
```

DESCRIPTION

`swab()` copies *nbytes* bytes pointed to by *from* to the array pointed to by *to*, exchanging adjacent even and odd bytes. It is useful for carrying binary data between byte-swapped and non-byte-swapped machines. *nbytes* should be even and non-negative. If *nbytes* is odd and positive `swab()` uses *nbytes*-1 instead. If *nbytes* is negative, `swab()` does nothing.

STANDARDS CONFORMANCE

`swab()`: AES, SVID2, XPG2, XPG3, XPG4

NAME

syslog(), openlog(), closelog(), setlogmask() - control system log

SYNOPSIS

```
#include <syslog.h>

int syslog(int priority, const char *message, int parameters, ...);

int openlog(const char *ident, int logopt, int facility);

int closelog(void);

int setlogmask(int maskpri);
```

DESCRIPTION

syslog() writes a message onto the system log maintained by **syslogd** (see *syslogd*(1M)). The message is tagged with *priority*. The *message* is similar to a *printf*(3S) format string except that **%m** is replaced by the error message associated with the current value of **errno**. A trailing newline is added if needed.

This message is read by **syslogd** and written to the system console, log files, selected users' terminals, or forwarded to **syslogd** on another host as appropriate.

priority is encoded as the logical OR of a *level* and a *facility*. The *level* signifies the urgency of the message, and *facility* signifies the subsystem generating the message. *facility* can be encoded explicitly in *priority*, or a default *facility* can be set with **openlog()** (see below).

level is selected from an ordered list:

LOG_EMERG	A panic condition. This is normally broadcast to all users.
LOG_ALERT	A condition that should be corrected immediately, such as a corrupted system database.
LOG_CRIT	Critical conditions, such as hard device errors.
LOG_ERR	Errors.
LOG_WARNING	Warning messages.
LOG_NOTICE	Conditions that are not error conditions, but should possibly be handled specially.
LOG_INFO	Informational messages.
LOG_DEBUG	Messages that contain information normally of use only when debugging a program.

syslog() does not log a message that does not have a *level* set.

If **syslog()** cannot pass the message to **syslogd**, it attempts to write the message on **/dev/console** if the **LOG_CONS** option is set (see below).

openlog()

can be called to initialize the log file, if special processing is needed. *ident* is a string that precedes every message. *logopt* is a mask of bits, logically OR'ed together, indicating logging options. The values for *logopt* are:

LOG_PID	Log the process ID with each message; useful for identifying instantiations of daemons.
LOG_CONS	Force writing messages to the console if unable to send it to **syslogd**. This option is safe to use in daemon processes that have no controlling terminal because **syslog()** forks before opening the console.
LOG_NDELAY	Open the connection to **syslogd** immediately. Normally, the open is delayed until the first message is logged. This is useful for programs that need to manage the order in which file descriptors are allocated.
LOG_NOWAIT	Do not wait for children forked to log messages on the console. This option should be used by processes that enable notification of child termination

via SIGCLD, because syslog() might otherwise block, waiting for a child whose exit status has already been collected.

facility encodes a default facility to be assigned to all messages written subsequently by syslog() with no explicit facility encoded.

LOG_KERN	Messages generated by the kernel. These cannot be generated by any user processes.
LOG_USER	Messages generated by random user processes. This is the default facility identifier if none is specified.
LOG_MAIL	The mail system.
LOG_DAEMON	System daemons, such as *inetd*(1M), *ftpd*(1M), etc.
LOG_AUTH	The authorization system: *login*(1), *su*(1), *getty*(1M), etc.
LOG_LPR	The line printer spooling system: *lp*(1), *lpsched*(1M), etc.
LOG_LOCAL0	Reserved for local use. Similarly for LOG_LOCAL1 through LOG_LOCAL7.

closelog()
closes the log file.

setlogmask()
sets the log priority mask to *maskpri* and returns the previous mask. Calls to syslog() with a priority not set in *maskpri* are rejected. The mask for an individual priority *pri* is calculated by the macro LOG_MASK(*pri*); the mask for all priorities up to and including *toppri* is given by the macro LOG_UPTO(*toppri*). By default, all priorities are logged.

RETURN VALUE

syslog() returns zero if it is successful in writing to the system log or if *priority* is masked out. It returns −1 if it is unable to write to the system log or if *priority* is out of range.

EXAMPLES

who logs a message regarding some sort of unexpected and serious error:

```
syslog(LOG_ALERT, "who: internal error 23");/s0
```

ftpd uses openlog() to arrange to log its process ID, to log to the console if necessary, and to log in the name of the *daemon* facility:

```
openlog("ftpd", LOG_PID|LOG_CONS, LOG_DAEMON);
```

Arrange to log messages only at levels LOG_ERR and lower:

```
setlogmask(LOG_UPTO(LOG_ERR));
```

Typical usage of syslog() to log a connection:

```
syslog(LOG_INFO, "Connection from host %d", CallingHost);
```

If the facility has not been set with openlog(), it defaults to LOG_USER.

Explicitly set the facility for this message:

```
syslog(LOG_INFO|LOG_LOCAL2, "foobar error: %m");
```

WARNINGS

A call to syslog() has no effect unless the syslog daemon (*syslogd*(1M)) is running. openlog() does not copy and store the *ident* string internally; it stores only a character pointer. Therefore it is the responsibility of the programmer to make sure that the *ident* argument points to the correct string until the log file is closed.

AUTHOR

syslog() was developed by the University of California, Berkeley.

SEE ALSO
logger(1), syslogd(1M).

NAME

system() - issue a shell command

SYNOPSIS

```
#include <stdlib.h>

int system(const char *command);
```

DESCRIPTION

`system()` executes the command specified by the string pointed to by *command*. The environment of the executed command is as if a child process were created using `fork()` (see *fork*(2)), and the child process invoked the *sh-posix*(1) utility via a call to `execl()` (see *execl*(2)) as follows:

```
execl("/bin/posix/sh", "sh", "-c", command, 0);
```

`system()` ignores the `SIGINT` and `SIGQUIT` signals, and blocks the `SIGCHLD` signal, while waiting for the command to terminate. If this might cause the application to miss a signal that would have killed it, the application should examine the return value from `system()` and take whatever action is appropriate to the application if the command terminated due to receipt of a signal.

`system()` does not affect the termination status of any child of the calling processes other than the process or processes it itself creates.

`system()` does not return until the child process has terminated.

Application Usage

If the return value of `system()` is not –1, its value can be decoded through the use of the macros described in <`sys/wait.h`>. For convenience, these macros are also provided in <`stdlib.h`>.

Note that, while `system()` must ignore `SIGINT` and `SIGQUIT` and block `SIGCHLD` while waiting for the child to terminate, the handling of signals in the executed command is as specified by *fork*(2) and *exec*(2). For example, if `SIGINT` is being caught or is set to `SIG_DFL` when `system()` is called, the child is started with `SIGINT` handling set to `SIG_DFL`.

Ignoring `SIGINT` and `SIGQUIT` in the parent process prevents coordination problems (such as two processes reading from the same terminal) when the executed command ignores or catches one of the signals.

RETURN VALUE

If *command* is null, `system()` returns non-zero.

If *command* is not null, `system()` returns the termination status of the command language interpreter in the format specified by *waitpid*(2). The termination status of the command language interpreter is as specified for *sh-posix*(1), except that if some error prevents the command language interpreter from executing after the child process is created, the return value from `system()` is as if the command language interpreter had terminated using `_exit(127)`. If a child process cannot be created, or if the termination status for the command language interpreter cannot be obtained, `system()` returns –1 and sets `errno` to indicate the error.

DIAGNOSTICS

`system()` forks to create a child process which, in turn, `exec()`s `/bin/posix/sh` in order to execute *string*. If the fork fails, `system()` returns –1 and sets `errno`. If the exec fails, `system()` returns the status value returned by `waitpid()` (see *waitpid*(2)) for a process that terminates with a call of `exit(127)`.

ERRORS

If errors are encountered, `system()` sets `errno` values as described by *fork*(2).

FILES

`/bin/posix/sh`

SEE ALSO

sh(1), fork(2), exec(2), waitpid(2).

STANDARDS CONFORMANCE

`system()`: AES, SVID2, XPG2, XPG3, XPG4, POSIX.2, ANSI C

NAME

tcgetattr(), tcsetattr() - control tty device

SYNOPSIS

```
#include <termios.h>

int tcgetattr(int fildes, struct termios *termios_p);

int tcsetattr(
    int fildes,
    int optional_actions,
    const struct termios *termios_p
);
```

DESCRIPTION

`tcgetattr()` gets the parameters associated with *fildes* and stores them in the *termios* structure referenced by *termios_p*. If the terminal device does not support split baud rates, the input baud rate stored in the *termios* structure is zero. This function is allowed from a background process (see *termio*(7)). However, the terminal attributes can be subsequently changed by a foreground process.

`tcsetattr()` sets the parameters associated with *fildes* (unless support is required from underlying hardware that is not available) from the *termios* structure referenced by *termios_p* as follows:

- If *optional_actions* is **TCSANOW**, the change is immediate.
- If *optional_actions* is **TCSADRAIN**, the change occurs after all output written to *fildes* is transmitted.
- If *optional_actions* is **TCSAFLUSH**, the change occurs after all output written to *fildes* is transmitted, and all input that has been received but not read is discarded.

RETURN VALUE

Upon successful completion, a value of zero is returned. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

ERRORS

`tcgetattr()` and `tcsetattr()` fail if any of the following conditions are encountered:

[EBADF]	*fildes* is not a valid file descriptor.
[EINVAL]	The *optional_actions* argument is not a proper value.
[ENOTTY]	The file associated with *fildes* is not a terminal.

WARNINGS

A request to set a hardware parameter to a value that is not supported by the hardware being used is ignored. Any remaining parameter values in the request that are supported or that do not affect hardware are set as requested. For any hardware that does not support separate input and output baud rates, the requested output baud rate is used to set the actual hardware baud rate. `tcgetattr()` always returns the actual values set in hardware.

SEE ALSO

cfspeed(3C), tccontrol(3C), termio(7).

STANDARDS CONFORMANCE

`tcgetattr()`: AES, XPG3, XPG4, FIPS 151-2, POSIX.1

`tcsetattr()`: AES, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

tcsendbreak(), tcdrain(), tcflush(), tcflow() - tty line control functions

SYNOPSIS

```
#include <termios.h>

int tcsendbreak(int fildes, int duration);

int tcdrain(int fildes);

int tcflush(int fildes, int queue_selector);

int tcflow(int fildes, int action);
```

DESCRIPTION

If the terminal is using asynchronous serial data transmission, `tcsendbreak()` causes transmission of a continuous stream of zero-valued bits for at least 0.25 seconds, but not more than 0.5 seconds. For all HP-UX implementations, *duration* is ignored.

`tcdrain()` waits until all output written to *fildes* has been transmitted.

`tcflush()` discards data written to *fildes* but not transmitted, or data received but not read, depending on the value of *queue_selector*:

- If *queue_selector* is `TCIFLUSH`, data received but not read is flushed.
- If *queue_selector* is `TCOFLUSH`, data written but not transmitted is flushed.
- If *queue_selector* is `TCIOFLUSH`, both data received but not read, and data written but not transmitted is flushed.

`tcflow()` suspends transmission of data to *fildes* or reception of data from *fildes* , depending on the value of *action*:

- If *action* is `TCOOFF`, output is suspended.
- If *action* is `TCOON`, suspended output is restarted.
- If *action* is `TCIOFF`, a STOP character is transmitted which is intended to cause the terminal to stop transmitting data to the system.
- If *action* is `TCION`, a START character is transmitted which is intended to cause the terminal to start transmitting data to the system.

RETURN VALUE

Upon successful completion, a value of zero is returned. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

ERRORS

These functions fail if one or more of the following is true:

[EBADF]	*fildes* is not a valid file descriptor.
[EINTR]	A signal was received during `tcdrain()`.
[EINVAL]	The *queue_selector* or the *action* argument is not a proper value.
[ENOTTY]	The file associated with *fildes* is not a terminal.

SEE ALSO

tcattribute(3C), tccontrol(3C), termio(7).

STANDARDS CONFORMANCE

`tcdrain()`: AES, XPG3, XPG4, FIPS 151-2, POSIX.1

`tcflow()`: AES, XPG3, XPG4, FIPS 151-2, POSIX.1

`tcflush()`: AES, XPG3, XPG4, FIPS 151-2, POSIX.1

`tcsendbreak()`: AES, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

tcgetpgrp() - get foreground process group id

SYNOPSIS

```
#include <unistd.h>

pid_t tcgetpgrp(int fildes);
```

DESCRIPTION

`tcgetpgrp()` returns the value of the process group ID of the foreground process group associated with the terminal referenced by *fildes*. `tcgetpgrp()` is allowed from a process that is a member of a background process group (see *termio*(7)); however, the information can be subsequently changed by a process that is a member of a foreground process group.

RETURN VALUE

Upon successful completion, `tcgetpgrp()` returns the value of the process group ID of the foreground process group associated with the terminal referenced by *fildes*. Otherwise, `tcgetpgrp()` returns a value of -1 and sets `errno` to indicate the error.

ERRORS

`tcgetpgrp()` fails if any of the following conditions are encountered:

[EACCES] The file associated with *fildes* is the controlling terminal of the calling process, however, there is no foreground process group defined for the controlling terminal.

[EBADF] *fildes* is not a valid file descriptor.

[ENOTTY] The file associated with *fildes* is not the controlling terminal or the calling process does not have a controlling terminal.

WARNING

The error EACCES, which is returned if the controlling terminal has no foreground process group, might not be returned in future releases, depending on the course taken by the POSIX standard. Portable applications therefore should not rely on this error condition.

SEE ALSO

setpgid(2), setsid(2), tcsetpgrp(3C), termio(7).

STANDARDS CONFORMANCE

`tcgetpgrp()`: AES, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

tcsetpgrp() - set foreground process group id

SYNOPSIS

```
#include <unistd.h>

int tcsetpgrp(int fildes, pid_t pgrp_id);
```

DESCRIPTION

If the calling process has a controlling terminal, `tcsetpgrp()` sets the foreground process group ID associated with the terminal referenced by *fildes* to *pgrp_id*. The file associated with *fildes* must be the controlling terminal of the calling process and the controlling terminal must be currently associated with the session of the calling process. The value of *pgrp_id* must match a process group ID of a process in the same session as the calling process.

RETURN VALUE

Upon successful completion, `tcsetpgrp()` returns zero. Otherwise, `tcsetpgrp()` returns -1 and sets `errno` to indicate the error.

ERRORS

`tcsetpgrp()` fails if any of the following conditions are encountered:

[EBADF]	*fildes* is not a valid file descriptor.
[EINVAL]	The value of the *pgrp_id* argument is not supported.
[ENOTTY]	The calling process does not have a controlling terminal, or the *fildes* is not the controlling terminal, or the controlling terminal is no longer associated with the session of the calling process.
[EPERM]	The value of *pgrp_id* is a supported value but does not match the process group ID of a process in the same session as the calling process.

SEE ALSO

setpgid(2), setsid(2), tcgetpgrp(3C), termio(7).

STANDARDS CONFORMANCE

`tcsetpgrp()`: AES, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

tgetent(), tgetnum(), tgetflag(), tgetstr(), tgoto(), tputs() - emulate /etc/termcap access routines

SYNOPSIS

```
#include <curses.h>

int tgetent(char *bp, const char *name);

int tgetnum(const char *id);

int tgetflag(const char *id);

char *tgetstr(const char *id, char **area);

char *tgoto(char *cm, int destcol, int destline);

int tputs(char *cp, int affcnt, int (*outc)(int));
```

DESCRIPTION

These functions extract and use capabilities from the compiled terminal capability data bases (see *terminfo*(4)). They are emulation routines that are provided as a part of the *curses*(3X) library.

`tgetent()` Extracts the compiled entry for terminal *name* into buffers accessible by the programmer. Unlike previous termcap routines, all capability strings (except cursor addressing and padding information) are already compiled and stored internally upon return from `tgetent()`. The buffer pointer *bp* is redundant in the emulation, and is ignored. It should not be relied upon to point to meaningful information. `tgetent()` returns -1 if it cannot access the *terminfo* directory, 0 if there is no capability file for *name,* and 1 if all goes well. If a `TERMINFO` environment variable is set, `tgetent()` first looks for `TERMINFO/?/`*name* (where `?` is the first character of *name*), and if that file is not accessible, it looks for `/usr/lib/terminfo/?/`*name*.

`tgetnum()` Gets the numeric value of capability *id,* returning -1 if it is not given for the terminal. `tgetnum()` is useful only with capabilities having numeric values.

`tgetflag()` Returns 1 if the specified capability is present in the terminal's entry, and 0 if it is not. `tgetflag()` is useful only with capabilities that are boolean in nature (i.e. either present or missing in *terminfo*(4)).

`tgetstr()` Returns a pointer to the string value of capability *id*. In addition, if *area* is not a NULL pointer, `tgetstr()` places the capability in the buffer at *area* and advances the area pointer. The returned string capability is compiled except for cursor addressing and padding information. `tgetstr()` is useful only with capabilities having string values.

`tgoto()` Returns a cursor addressing string decoded from *cm* to go to column *destcol* in line *destline*. (Programs that call `tgoto()` should be sure to turn off the TAB3 bit or bits, since `tgoto()` can now output a tab. See *termio*(7)). Note that programs using `termcap` should in general turn off TAB3 anyway since some terminals use Ctrl-I for other functions, such as nondestructive space.) If a `%` sequence is given that is not understood, `tgoto()` returns [OOPS].

`tputs()` Decodes the padding information of the string *cp*. *affcnt* gives the number of lines affected by the operation, or 1 if this is not applicable. *outc* is a routine that is called with each character in turn. The `terminfo` variable *pad_char* should contain a pad character to be used (from the *pc* capability) if a null (`^@`) is inappropriate.

FILES

`/usr/lib/libcurses.a` `-lcurses` library

`/usr/lib/terminfo/?/*` data bases

SEE ALSO

ex(1), terminfo(4), termio(7).

NAME

tmpfile() - create a temporary file

SYNOPSIS

```
#include <stdio.h>

FILE *tmpfile(void);
```

DESCRIPTION

`tmpfile()` creates a temporary file by generating a name through `tmpnam()` (see *tmpnam*(3S)), and returns a corresponding `FILE` pointer. If the file cannot be opened a NULL pointer is returned. The file is automatically deleted when the process using it terminates. The file is opened for update (`wb+`).

NOTES

On HP-UX systems, the `wb+` mode is equivalent to the `w+` mode.

SEE ALSO

creat(2), unlink(2), mktemp(3C), fopen(3S), tmpnam(3S).

STANDARDS CONFORMANCE

`tmpfile()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

tmpnam(), tempnam() - create a name for a temporary file

SYNOPSIS

```
#include <stdio.h>

char *tmpnam(char *s);

char *tempnam(const char *dir, const char *pfx);
```

DESCRIPTION

`tmpnam()` and `tempnam()` generate file names that can safely be used for a temporary file.

`tmpnam()` Always generates a file name using the path-prefix defined as `P_tmpdir` in the `<stdio.h>` header file. If *s* is NULL, `tmpnam()` leaves its result in an internal static area and returns a pointer to that area. The next call to `tmpnam()` destroys the contents of the area. If *s* is not NULL, it is assumed to be the address of an array of at least `L_tmpnam` bytes, where `L_tmpnam` is a constant defined in `<stdio.h>`; `tmpnam()` places its result in that array and returns *s*.

`tempnam()` allows the user to control the choice of a directory. The argument *dir* points to the name of the directory in which the file is to be created. If *dir* is NULL or points to a string that is not an appropriate directory name, the path-prefix defined as `P_tmpdir` in the `<stdio.h>` header file is used. If that directory is not accessible, `/tmp` is used as a last resort. This entire sequence can be eliminated by providing an environment variable `TMPDIR` in the user's environment, whose value is the name of the desired temporary-file directory.

Many applications are written such that temporary files have certain initial character sequences in their names. Use the *pfx* argument to define a given prefix. The argument can be NULL or point to a string of up to five characters to be used as the first characters in the temporary-file name.

`tempnam()` uses `malloc()` (see *malloc*(3C)) to get space for the constructed file name, and returns a pointer to this area. Thus, any pointer value returned from `tempnam()` can serve as an argument to `free()` (see *malloc*(3C)). If `tempnam()` cannot return the expected result for any reason; i.e., `malloc()` failed, or none of the above mentioned attempts to find an appropriate directory was successful, a NULL pointer is returned.

NOTES

`tmpnam()` and `tempnam()` generate a different file name each time they are called, but start recycling previously used names if called more than `TMP_MAX` times in a single process.

Files created using these functions and either `fopen()` or `creat()` (see *fopen*(3S) and *creat*(2)) are temporary only in the sense that they reside in a directory intended for temporary use, and their names are unique. It is the user's responsibility to use *unlink*(2) to remove the file when it is no longer needed.

WARNINGS

Between the time a file name is created and the file is opened, it is possible for some other process to create a file with the same name. This can never happen if that other process is using these functions or `mktemp`, and the file names are chosen such that duplication by other means is unlikely.

SEE ALSO

creat(2), unlink(2), malloc(3C), mktemp(3C), fopen(3S), tmpfile(3S).

STANDARDS CONFORMANCE

`tmpnam()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`tempnam()`: AES, SVID2, XPG2, XPG3, XPG4

NAME

sin(), cos(), tan(), asin(), acos(), atan(), atan2(), sinf(), cosf(), tanf(), asinf(), acosf(), atanf(), atan2f() - trigonometric functions

SYNOPSIS

```
#include <math.h>

double sin(double x);

double cos(double x);

double tan(double x);

double asin(double x);

double acos(double x);

double atan(double x);

double atan2(double y, double x);

float sinf(float x);

float cosf(float x);

float tanf(float x);

float asinf(float x);

float acosf(float x);

float atanf(float x);

float atan2f(float y, float x);
```

DESCRIPTION

The following trigonometric functions return the values indicated:

`sin(`*x*`)` sine of *x* (*x* specified in radians)

`cos(`*x*`)` cosine of *x* (*x* specified in radians)

`tan(`*x*`)` tangent of *x* (*x* specified in radians)

`asin(`*x*`)` arcsine of *x* in the range $-\pi/2$ to $\pi/2$.

`acos(`*x*`)` arccosine of *x* in the range 0 to π.

`atan(`*x*`)` arctangent of *x* in the range $-\pi/2$ to $\pi/2$. If *x* is ±INFINITY , `atan()` returns $\pm\pi/2$ respectively.

`atan2(`*y*`,`*x*`)` arctangent of *y*/*x*, in the range $-\pi$ to π, using the signs of both arguments to determine the quadrant of the return value.

Other `atan2()` returns:

- $\pi/4$ when *y* and *x* are +INFINITY.
- $3*\pi/4$ when *y* is +INFINITY and *x* is –INFINITY.
- $-\pi/4$ when *y* is –INFINITY and *x* is +INFINITY.
- $-3*\pi/4$ when *y* and *x* are –INFINITY.
- 0.0 when *y* is 0.0 and *x* is a positive number.
- π when *y* is 0.0 and *x* is a negative number, or $-\pi$ when *y* is -0.0 and *x* is a negative number.
- $\pi/2$ when *y* is a positive number and *x* is 0.0, or $-\pi/2$ when *y* is a negative number and *x* is 0.0.
- $\pm\pi/2$ if *y*/*x* would overflow. The result will be $\pi/2$ if *y* is a positive number and $-\pi/2$ if *y* is a negative number.
- $\pm\pi$ or 0.0 if *y*/*x* would underflow. The result is 0.0 if *x* is a positive number, π if *x* is a negative number and *y* is a positive number, and $-\pi$ if *x* and *y* are both negative numbers.

`sinf()`, `cosf()`, `tanf()`, `asinf()`, `acosf()`, `atanf()`, and `atan2f()` are `float` versions of these functions; they take `float` arguments and return `float` results. Their performance is significantly faster than that of the `double` versions of the functions. Programs must be compiled in ANSI mode (use the `-Aa` option) in order to use these functions; otherwise, the compiler promotes the `float` arguments to `double`, and the functions return incorrect results.

DEPENDENCIES

Series 300/400

`sinf()`, `cosf()`, `tanf()`, `asinf()`, `acosf()`, `atanf()`, and `atan2f()` are not supported on Series 300/400 systems.

Series 700/800

`sinf()`, `cosf()`, `tanf()`, `asinf()`, `acosf()`, `atanf()`, and `atan2f()` are not specified by any standard (they are, however, named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard). These functions are provided in the PA1.1 versions of the math library only. The `+DA1.1` option (the default on Series 700 systems) links in a PA1.1 version automatically. A PA1.1 library can be linked in explicitly. For more information, see the *HP-UX Floating-Point Guide*.

ERRORS

/lib/libm.a

`sin()`, `cos()`, and `tan()` lose accuracy when their argument is far from zero. For arguments sufficiently large, these functions return 0.0 when there would otherwise be a complete loss of significance. In this case a message indicating TLOSS error is printed on the standard error output. For less extreme arguments causing partial loss of significance, a PLOSS error is generated but no message is printed. In both cases, `errno` is set to ERANGE.

If the magnitude of the argument of `asin()` or `acos()` is greater than one, or if both arguments of `atan2()` are 0.0, 0.0 is returned and `errno` is set to EDOM. In addition, a message indicating `DOMAIN` error is printed on the standard error output.

`sin()`, `cos()`, `tan()`, `acos()`, and `asin()` return NaN and set `errno` to EDOM when x is NaN or ±INFINITY. In addition, a message indicating `DOMAIN` error is printed on the standard error output.

`atan()` returns NaN and sets `errno` to EDOM when x is NaN. In addition, a message indicating `DOMAIN` error is printed on the standard error output.

`atan2()` returns NaN and sets `errno` to EDOM when x or y is NaN. In addition, a message indicating `DOMAIN` error is printed on the standard error output.

These error-handling procedures can be changed with the `matherr()` function (see *matherr*(3M)).

/lib/libM.a

No error messages are printed on the standard error output.

`sin()`, `cos()`, and `tan()` lose accuracy when their argument is far from zero. For arguments sufficiently large, these functions return 0.0 when there would otherwise be a complete loss of significance. For less extreme arguments causing partial loss of significance, a `PLOSS` error is generated. In both cases, `errno` is set to ERANGE.

If the magnitude of the argument of `asin()` or `acos()` is greater than one, NaN is returned and `errno` is set to EDOM.

If both arguments of `atan2()` are 0.0, 0.0 is returned and `errno` is set to EDOM.

`sin()`, `cos()`, `tan()`, `acos()`, and `asin()` return NaN and set `errno` to EDOM when x is NaN or ±INFINITY.

`atan()` returns NaN and sets `errno` to EDOM when x is NaN.

`atan2()` returns NaN and sets `errno` to EDOM when x or y is NaN.

These error-handling procedures can be changed with the function `_matherr()` (see *matherr*(3M)). Note that `_matherr()` is provided in order to assist in migrating programs from `libm.a` to `libM.a` and is *not* a part of XPG3, ANSI C, or POSIX.

SEE ALSO

trigd(3M), isinf(3M), isnan(3M), matherr(3M).

STANDARDS CONFORMANCE

`acos()` in libm.a: AES, SVID2, XPG2, XPG3, FIPS 151-2, POSIX.1
`acos()` in libM.a: AES, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`asin()` in libm.a: AES, SVID2, XPG2, XPG3, FIPS 151-2, POSIX.1
`asin()` in libM.a: AES, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`atan()` in libm.a: AES, SVID2, XPG2, XPG3, FIPS 151-2, POSIX.1
`atan()` in libM.a: AES, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`atan2()` in libm.a: AES, SVID2, XPG2, XPG3, FIPS 151-2, POSIX.1
`atan2()` in libM.a: AES, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`cos()` in libm.a: AES, SVID2, XPG2, XPG3, FIPS 151-2, POSIX.1
`cos()` in libM.a: AES, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`sin()` in libm.a: AES, SVID2, XPG2, XPG3, FIPS 151-2, POSIX.1
`sin()` in libM.a: AES, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

`tan()` in libm.a: AES, SVID2, XPG2, XPG3, FIPS 151-2, POSIX.1
`tan()` in libM.a: AES, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

sind(), cosd(), tand(), asind(), acosd(), atand(), atan2d(), sindf(), cosdf(), tandf(), asindf(), acosdf(), atandf(), atan2df() - degree-valued trigonometric functions

SYNOPSIS

```
#include <math.h>

double sind(double x);

double cosd(double x);

double tand(double x);

double asind(double x);

double acosd(double x);

double atand(double x);

double atan2d(double y, double x);

float sindf(float x);

float cosdf(float x);

float tandf(float x);

float asindf(float x);

float acosdf(float x);

float atandf(float x);

float atan2df(float y, float x);
```

DESCRIPTION

`sind()`, `cosd()`, `tand()`, `asind()`, `acosd()`, `atand()`, and `atan2d()` are degree-valued versions of the trigonometric functions. The functions return the values indicated:

`sind()` sine of x (x specified in degrees)

`cos()` cosine of x (x specified in degrees)

`tand()` tangent of x (x specified in degrees)

`asind()` arcsine of x in the range –90 to 90.

`acosd()` arccosine of x in the range 0 to 180.

`atand()` arctangent of x in the range –90 to 90. If x is ±INFINITY , `atand()` returns ±90 respectively.

`atan2d()` arctangent of y/x, in the range –180 to 180, using the signs of both arguments to determine the quadrant of the return value.

Other `atan2d()` returns:

- 45 when y and x are +INFINITY.
- 135 when y is +INFINITY and x is –INFINITY.
- –45 when y is –INFINITY and x is +INFINITY.
- –135 when y and x are –INFINITY.
- 0.0 when y is 0.0 and x is a positive number.
- 180 when y is 0.0 and x is a negative number, or –180 when y is -0.0 and x is a negative number.
- 90 when y is a positive number and x is 0.0, or –90 when y is a negative number and x is 0.0.
- ±90 if y/x would overflow. The result will be 90 if y is a positive number and –90 if y is a negative number.
- ±180 or 0.0 if y/x would underflow. The result will be 0.0 if x is a positive number, 180 if x is a negative number and y is a positive number, and –180 if x and y are both negative numbers.

`sindf()`, `cosdf()`, `tandf()`, `asindf()`, `cosdf()`, `atandf()`, and `atan2df()` are `float` versions of these functions; they take `float` arguments and return `float` results. They are named in accordance with the conventions specified in the "Future Library Directions" section of the ANSI C standard. Their performance is significantly faster than that of the `double` versions of the functions. Compiling must be done in ANSI mode (use the `-Aa` option) in order to use these functions; otherwise, the compiler promotes the `float` arguments to `double`, and the functions return incorrect results.

DEPENDENCIES

Series 300/400

These functions are not supported on the Series 300/400.

Series 700/800

These functions are provided in the PA1.1 versions of the math library only. The `+DA1.1` option (the default on Series 700 systems) links in a PA1.1 version automatically. A PA1.1 library can be linked in explicitly. For more information, see the *HP-UX Floating-Point Guide*.

ERRORS

/lib/libm.a

`sind()`, `cosd()`, and `tand()` lose accuracy when their argument is far from zero. For arguments sufficiently large, these functions return 0 when there would otherwise be a complete loss of significance. In this case a message indicating `TLOSS` error is printed on the standard error output. For less extreme arguments causing partial loss of significance, a `PLOSS` error is generated but no message is printed. In both cases, `errno` is set to ERANGE.

If the magnitude of the argument of `asind()` or `acosd()` is greater than one, or if both arguments of `atan2d()` are 0.0, 0 is returned and `errno` is set to EDOM. In addition, a message indicating `DOMAIN` error is printed on the standard error output.

`sind()`, `cosd()`, `tand()`, `acosd()`, and `asind()` return NaN and set `errno` to EDOM when x is NaN or ±INFINITY. In addition, a message indicating DOMAIN error is printed on the standard error output.

`atand()` returns NaN and sets `errno` to EDOM when x is NaN. In addition, a message indicating DOMAIN error is printed on the standard error output.

`atan2d()` returns NaN and sets `errno` to EDOM when x or y is NaN. In addition, a message indicating `DOMAIN` error is printed on the standard error output.

These error-handling procedures can be changed with the `matherr()` function (see *matherr*(3M)).

/lib/libM.a

No error messages are printed on the standard error output.

`sind()`, `cosd()`, and `tand()` lose accuracy when their argument is far from zero. For arguments sufficiently large, these functions return 0 when there would otherwise be a complete loss of significance. For less extreme arguments causing partial loss of significance, a `PLOSS` error is generated. In both cases, `errno` is set to ERANGE.

If the magnitude of the argument of `asind()` or `acosd()` is greater than one, NaN is returned and `errno` is set to EDOM.

If both arguments of `atan2d()` are 0.0, 0 is returned and `errno` is set to EDOM.

`sind()`, `cosd()`, `tand()`, `acosd()`, and `asind()` return NaN and set `errno` to EDOM when x is NaN or ±INFINITY.

`atand()` returns NaN and sets `errno` to EDOM when x is NaN.

`atan2d()` returns NaN and sets `errno` to EDOM when x or y is NaN.

These error-handling procedures can be changed with the function `_matherr()` (see *matherr*(3M)). Note that `_matherr()` is provided in order to assist in migrating programs from `libm.a` to `libM.a` and is *not* a part of XPG3, ANSI C, or POSIX.

SEE ALSO

trig(3M), isinf(3M), isnan(3M), matherr(3M).

NAME

tsearch(), tfind(), tdelete(), twalk() - manage binary search trees

SYNOPSIS

```
#include <search.h>

void *tsearch(
    const void *key,
    void **rootp,
    int (*compar)(const void *, const void *)
);

void *tfind(
    const void *key,
    void * const *rootp,
    int (*compar)(const void *, const void *)
);

void *tdelete(
    const void *key,
    void **rootp,
    int (*compar)(const void *, const void *)
);

void twalk(
    const void *root,
    void (*action)(const void *, VISIT, int)
);
```

DESCRIPTION

`tsearch()`, `tfind()`, `tdelete()`, and `twalk()` are routines for manipulating binary search trees. They are generalized from Knuth (6.2.2) Algorithms T and D. All comparisons are done with a user-supplied routine, *compar*. This routine is called with two arguments, the pointers to the elements being compared. It returns an integer less than, equal to, or greater than 0, according to whether the first argument is to be considered less than, equal to or greater than the second argument. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

`tsearch()` is used to build and access the tree. *key* is a pointer to an entry to be accessed or stored. If there is an entry in the tree equal to the value pointed to by *key*, a pointer to the previous key associated with this found entry is returned. Otherwise, *key* is inserted, and a pointer to it returned. Note that since the value returned is a pointer to *key* and *key* itself is a pointer, the value returned is a pointer to a pointer. Only pointers are copied, so the calling routine must store the data. *rootp* points to a variable that points to the root of the tree. A NULL value for the variable pointed to by *rootp* denotes an empty tree; in this case, the variable is set to point to the entry which will be at the root of the new tree.

Like `tsearch()`, `tfind()` searches for an entry in the tree, returning a pointer to it if found. However, if it is not found, `tfind()` returns a NULL pointer. The arguments for `tfind()` are the same as for `tsearch()`.

`tdelete()` deletes a node from a binary search tree. Arguments are the same as for `tsearch()`. The variable pointed to by *rootp* is changed if the deleted node was the root of the tree. `tdelete()` returns a pointer to the parent of the deleted node, or a NULL pointer if the node is not found.

`twalk()` traverses a binary search tree. *root* is the root of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.) *action* is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments:

- First argument is the address of the node being visited.
- Second argument is a value from an enumeration data type `typedef enum { preorder, postorder, endorder, leaf } VISIT;` (defined in the <search.h> header file), depending on whether this is the first, second or third time that the node has been visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf.

- Third argument is the level of the node in the tree, with the root being level zero.

EXAMPLE

The following code reads strings, and stores structures containing a pointer to each string and a count of its length. It then walks the tree, printing out the stored strings and their lengths in alphabetical order.

```
.C #include <stdlib.h>
.C #include <search.h>
.C #include <stdio.h>
.C #include <string.h>
.IP
.C  struct element /* pointers to
.C {
.C  char *string;"
.C  int length;"
.C };
.C  char string_space[10000]; /* space to
struct element elements[500];           /* elements to store */
struct element *root = NULL;    /* this points to the root */
void print_node(void *, VISIT, int);
int element_compare(const void *, const void *);

main( )
{
    char *strptr = string_space;
    struct element *element_ptr = elements;
    struct element **ts_retval;

    int i = 0;

    while (gets(strptr) != NULL && i++ < 500)
    {
        /* set element */
        element_ptr->string = strptr;
        element_ptr->length = strlen(strptr);

        /* put element into the tree */
        ts_retval = (struct element **) tsearch((void *) element_ptr,
                    (void **) &root, element_compare);

        if (*ts_retval == element_ptr)
        {
            (void) printf("The element \"%s\" ",
                          (*ts_retval)->string);
            (void) printf("has now been inserted into the tree\n");
        }
        else
        {
            (void) printf("The element \"%s\" ",
                          (*ts_retval)->string);
            (void) printf("already existed in the tree\n");
        }

        /* adjust pointers, so we don't overwrite tree */
        strptr += element_ptr->length + 1;
        element_ptr++;
    }
    twalk((void *) root, print_node);
}
/* This routine compares two elements, based on an
      alphabetical ordering of the string field.  */
int
element_compare(elem1, elem2)
```

```
void *elem1, *elem2;
{
    return strcmp(((struct element *) elem1)->string,
                  ((struct element *) elem2)->string);
}
/* This routine prints out a node, the first time
    twalk encounters it.  */
void
print_node(element, order, level)
void *element;
VISIT order;
int level;
{
    if (order == preorder || order == leaf)
    {
        (void) printf("string = %20s,  length = %d\n",
                      (*(struct element **) element)->string,
                      (*(struct element **) element)->length);
    }
}
```

SEE ALSO

bsearch(3C), hsearch(3C), lsearch(3C).

RETURN VALUE

A NULL pointer is returned by `tsearch()` if there is not enough space available to create a new node.

A NULL pointer is returned by `tsearch()`, `tfind()`, and `tdelete()` if *rootp* is NULL on entry.

If the datum is found, both `tsearch()` and `tfind()` return a pointer to it. If not, `tfind()` returns NULL, and `tsearch()` returns a pointer to the inserted item.

WARNINGS

The *root* argument to `twalk()` is one level of indirection less than the *rootp* arguments to `tsearch()` and `tdelete()`.

Two nomenclatures are used to refer to the order in which tree nodes are visited. `tsearch()` uses preorder, postorder and endorder to respectively refer to visting a node before any of its children, after its left child and before its right and after both its children. The alternate nomenclature uses preorder, inorder, and postorder to refer to the same visits, which could result in some confusion over the meaning of postorder. If the calling function alters the pointer to the root, results are unpredictable.

STANDARDS CONFORMANCE

`tsearch()`: AES, SVID2, XPG2, XPG3, XPG4

`tdelete()`: AES, SVID2, XPG2, XPG3, XPG4

`tfind()`: AES, SVID2, XPG2, XPG3, XPG4

`twalk()`: AES, SVID2, XPG2, XPG3, XPG4

NAME

ttyname(), isatty() - find name of a terminal

SYNOPSIS

```
#include <unistd.h>

char *ttyname(int fildes);

int isatty(int fildes);
```

DESCRIPTION

`ttyname()` returns a pointer to a string containing the null-terminated path name of the terminal device associated with file descriptor *fildes*.

`isatty()` returns 1 if *fildes* is associated with a terminal device, 0 otherwise.

RETURN VALUE

`ttyname()` returns a NULL pointer if *fildes* does not describe a terminal device in directory `/dev`.

ERRORS

`isatty()` and `ttyname()` fail if any of the following conditions are encountered:

[EBADF]	The *fildes* argument is invalid.
[ENOTTY]	An inappropriate I/O control operation has been attempted.

WARNINGS

The return value points to static data whose content is overwritten by each call.

FILES

```
/dev/*
/dev/pty/*
```

STANDARDS CONFORMANCE

`ttyname()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

`isatty()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1

NAME

ttyslot() - find the slot in the utmp file of the current user

SYNOPSIS

```
#include <unistd.h>

int ttyslot(void);
```

DESCRIPTION

`ttyslot()` returns the index of the current user's entry in the `/etc/utmp` file. This is accomplished by scanning `/etc/utmp` for the name of the terminal associated with the standard input, standard output, or standard error (file descriptor 0, 1 or 2).

RETURN VALUE

`ttyslot()` returns -1 if an error was encountered while searching for the terminal name or if none of file descriptors 0, 1, or 2 is associated with a terminal device.

FILES

/etc/utmp

SEE ALSO

getut(3C), ttyname(3C).

STANDARDS CONFORMANCE

`ttyslot()`: XPG2

NAME

ungetc() - push character back into input stream

SYNOPSIS

```
#include <stdio.h>

int ungetc(int c, FILE *stream);
```

DESCRIPTION

`ungetc()` inserts the character *c* (converted to an unsigned char) into the buffer associated with an input *stream*. That character, *c*, is returned by the next call to `getc()` (see *getc*(3S)) on that *stream*. A successful intervening call to a file positioning function with *stream* (`fseek()`, `fsetpos()`, or `rewind()`) erases all memory of the inserted characters.

`ungetc()` affects only the buffer associated with the input *stream*. It does not affect the contents of the file corresponding to *stream*.

One character of pushback is guaranteed.

If *c* equals EOF, `ungetc()` does nothing to the buffer and returns EOF.

RETURN VALUE

If successful, `ungetc()` returns *c* and clears the end-of-file indicator for the stream. `ungetc()` returns EOF if it cannot insert the character.

SEE ALSO

fseek(3S), fsetpos(3S), getc(3S), setbuf(3S).

STANDARDS CONFORMANCE

`ungetc()`: AES, SVID2, XPG2, XPG3, XPG4, FIPS 151-2, POSIX.1, ANSI C

NAME

ungetwc() - push a wide character back into an input stream

SYNOPSIS

```
#include <wchar.h>

wint_t ungetwc(wint_t wc, FILE *stream);
```

Remarks:

This function is compliant with the XPG4 Worldwide Portability Interface wide-character I/O functions. It parallels the 8-bit character I/O function defined in *ungetc(3S)*.

DESCRIPTION

`ungetwc()` pushes the character corresponding to the wide-character code *wc* into the buffer associated with an input *stream*. That wide-character code, *wc*, is returned by the next call to `getwc()` (see *getwc*(3C)) on that *stream*. A successful intervening call to a file positioning function with *stream* (`fseek()`, `fsetpos()`, or `rewind()`) erases all memory of the pushed-back characters.

`ungetwc()` affects only the buffer associated with the input *stream*. It does not affect the contents of the file corresponding to *stream*.

One character of pushback is guaranteed.

If *wc* equals `WEOF`, `ungetwc()` does nothing to the buffer and returns `WEOF`.

The definition for this function, the type `wint_t` and the value `WEOF` are provided in the `<wchar.h>` header.

EXTERNAL INFLUENCES

Locale

The `LC_CTYPE` category determines how wide character conversions are done.

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUE

If successful, `ungetwc()` returns *wc* and clears the end-of-file indicator for the stream. `ungetwc()` returns `WEOF` if it cannot insert the wide character.

SEE ALSO

fseek(3S), fsetpos(3S), getwc(3C), setbuf(3S).

STANDARDS CONFORMANCE

`ungetwc()`: XPG4

NAME

vprintf(), vfprintf(), vsprintf() - print formatted output of a varargs argument list

SYNOPSIS

```
#include <stdio.h>
#include <varargs.h>

int vprintf(const char *format, va_list ap);

int vfprintf(FILE *stream, const char *format, va_list ap);

int vsprintf(char *s, const char *format, va_list ap);
```

DESCRIPTION

vprintf(), **vfprintf()**, and **vsprintf()** are the same as **printf()**, **fprintf()**, and **sprintf()** respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by *varargs*(5).

EXAMPLE

The following demonstrates how **vfprintf()** could be used to write an error routine:

```
#include <stdio.h>
#include <varargs.h>
      .
      .
      .
/*
 *    error should be called using the form
 *         error(function_name, format, arg1, arg2...);
 */
/*VARARGS0*/
void
error(va_alist)
va_dcl
{
      va_list args;
      char *fmt;

      va_start(args);

      /* print out name of function causing error */
      (void)fprintf(stderr, "ERROR in %s: ", va_arg(args, char *));
      fmt = va_arg(args, char *);

      /* print out remainder of message */
      (void)vfprintf(stderr, fmt, args);
      va_end(args);
      (void)abort( );
}
```

SEE ALSO

setlocale(3C), printf(3S), varargs(5).

STANDARDS CONFORMANCE

vprintf(): AES, SVID2, XPG2, XPG3, XPG4, ANSI C

vfprintf(): AES, SVID2, XPG2, XPG3, XPG4, ANSI C

vsprintf(): AES, SVID2, XPG2, XPG3, XPG4, ANSI C

NAME

vscanf(), vfscanf(), vsscanf() - formatted input conversion to a varargs argument list, read from stream file

SYNOPSIS

```
#include <stdio.h>
#include <varargs.h>

int vscanf(const char *format, va_list ap);

int vfscanf(FILE *stream, const char *format, va_list ap);

int vsscanf(char *s, const char *format, va_list ap);
```

DESCRIPTION

vscanf(), **vfscanf()**, and **vsscanf()** are the same as **scanf()**, **fscanf()**, and **sscanf()** respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by *varargs*(5).

SEE ALSO

scanf(3S), setlocale(3C), varargs(5).

NAME

towupper(), towlower() - translate wide characters

SYNOPSIS

```
#include <wchar.h>

wint_t towupper(wint_t wc);

wint_t towlower(wint_t wc);
```

Remarks:

These functions are compliant with the XPG4 Worldwide Portability Interface wide-character conversion functions. They parallel the 8-bit character conversion functions defined in *conv(3C)*.

DESCRIPTION

`towupper()` and `towlower()` have as domain a `wint_t`, the value of which is representable as a `wchar_t` or the value `WEOF`. If the argument has any other value, the behavior is undefined. If the argument of `towupper()` represents a lowercase letter, the result is the corresponding uppercase letter. If the argument of `towlower()` represents an uppercase letter, the result is the corresponding lowercase letter. All other arguments are returned unchanged.

Definitions for these functions, the types `wint_t`, `wchar_t`, and the value `WEOF` are provided in the <wchar.h> header.

EXTERNAL INFLUENCES

Locale

The `LC_CTYPE` category determines the translations to be done.

International Code Set Support

Single-byte character code sets are supported. Japanese HP15 and EUC multi-byte character code sets are supported. `towupper()` and `towlower()` return their argument for values in other multi-byte character code sets outside the ASCII range.

WARNING

`towupper()` and `towlower()` are supplied both as library functions and as macros defined in the <wchar.h> header. Normally, the macro versions are used. To obtain the library function, either use a `#undef` to remove the macro definition or, if compiling in ANSI C mode, enclose the function name in parenthesis or take its address. The following examples use the library function for `towlower()`:

```
#include <wchar.h>
#undef towlower
    ...
main()
{
        ...
    c1 = towlower(c);
        ...
}
```

or

```
#include <wchar.h>
    ...
main()
{
   wint_t (*conv_func)();
            ...
   c1 = (towlower)(c);
            ...
   conv_func = towlower;
            ...
}
```

AUTHOR

`wconv()` was developed by AT&T and HP.

SEE ALSO

conv(3C), multibyte(3C), wctype(3C), setlocale(3C), lang(5).

STANDARDS CONFORMANCE

`towlower()`: XPG4

`towupper()`: XPG4

NAME

wcsftime() - convert date and time to wide-character string

SYNOPSIS

```
#include <wchar.h>

size_t wcsftime(
    wchar_t *ws,
    size_t maxsize,
    const char *format,
    const struct tm *timeptr
);
```

Remarks:

This function is compliant with the XPG4 Worldwide Portability Interface wide-character formatting functions. It parallels the 8-bit character formatting function defined in *strftime*(3C).

DESCRIPTION

`wcsftime()` converts the contents of a `tm` structure (see *ctime*(3C)) to a formatted date and time wide-character string.

`wcsftime()` places wide characters into the array pointed to by *ws* as controlled by the string pointed to by *format*. The *format* string consists of zero or more directives and ordinary characters. A directive consists of a `%` character, an optional field width and precision specification, and a terminating character that determines the directive's behavior. All ordinary characters (including the terminating null character) are converted into corresponding wide characters and are copied into the array. No more than *maxsize* wide characters are placed into the array. Each directive is replaced by the appropriate wide characters as described in the following list. The appropriate wide characters are determined by the program's locale, by the values contained in the structure pointed to by *timeptr*, and by the `TZ` environment variable (see External Influences below).

The definition for this function and the type `wchar_t` are provided in the <`wchar.h`> header.

Directives

The following directives, shown without the optional field width and precision specification, are replaced by the corresponding wide characters as indicated:

`%a`	locale's abbreviated weekday name
`%A`	locale's full weekday name
`%b`	locale's abbreviated month name
`%B`	locale's full month name
`%c`	locale's appropriate date and time representation
`%d`	day of the month as a decimal number [01,31]
`%E`	locale's combined Emperor/Era name and year
`%H`	hour (24-hour clock) as a decimal number [00,23]
`%I`	hour (12-hour clock) as a decimal number [01,12]
`%j`	day of the year as a decimal number [001,366]
`%m`	month as a decimal number [01,12]
`%M`	minute as a decimal number [00,59]
`%n`	new-line wide character
`%N`	locale's Emperor/Era name
`%o`	locale's Emperor/Era year
`%p`	locale's equivalent of either AM or PM
`%S`	second as a decimal number [00,61]
`%t`	tab wide character
`%U`	week number of the year (the first Sunday as the first day of week 1) as a decimal number [00,53]
`%w`	weekday as a decimal number [0(Sunday),6]
`%W`	week number of the year (the first Monday as the first day of week 1) as a decimal number [00,53]

%x	locale's appropriate date representation
%X	locale's appropriate time representation
%y	year without century as a decimal number [00,99]
%Y	year with century as a decimal number
%Z	time zone name (or by no characters if no time zone exists)
%%	Percent character (%)

The following directives are provided for backward compatibility with the directives supported by the `date` command and the `ctime()` functions (see *date*(1) and *ctime*(3C)). It is recommended that the directives above be used in preference to those below.

%D	date in usual U.S. format (%m/%d/%y) (use **%x** instead)
%F	locale's full month name (use **%B** instead)
%h	locale's abbreviated month name (use **%b** instead)
%r	time in 12-hour U.S. format (%I:%M:%S [AM | PM]) (use **%X** instead)
%T	time in 24-hour U.S. format (%H:%M:%S) (use **%X** instead)
%z	time zone name (or by no characters if no time zone exists) (use **%Z** instead)

If a directive is not one of the above, the behavior is undefined.

Field Width and Precision

An optional field width and precision specification can immediately follow the initial % of a directive in the following order:

[- | 0]*w*	the decimal digit string *w* specifies a minimum field width in which the result of the conversion is right- or left-justified. It is right-justified (with space padding) by default. If the optional - character is specified, it is left-justified with space padding on the right. If the optional 0 character is specified, it is right-justified and padded with zeros on the left.
.*p*	the decimal digit string *p* specifies the minimum number of digits to appear for the **d, H, I, j, m, M, o, S, U, w, W, y** and **Y** directives, and the maximum number of corresponding wide characters to be used from the **a, A, b, B, c, D, E, F, h, n, N, p, r, t, T, x, X, z, Z,** and **%** directives. In the first case, if a directive supplies fewer digits than specified by the precision, it is expanded with leading zeros. In the second case, if a directive supplies more characters than specified by the precision, excess characters are truncated on the right.

If no field width or precision is specified for a **d, H, I, m, M, S, U, W, y,** or **j** directive, a default of **.2** is used for all but **j** for which **.3** is used.

EXTERNAL INFLUENCES

Locale

The **LC_TIME** category determines the characters to be substituted for those directives described above as being from the locale.

The **LC_CTYPE** category determines the interpretation of the bytes within *format* as single and/or multi-byte characters as well as how wide-character conversions are done.

The **LC_NUMERIC** category determines the characters used to form numbers for those directives that produce numbers in the output. If **ALT_DIGITS** (see *langinfo*(5)) is defined for the locale, the characters so specified are used in place of the default ASCII characters.

Environment Variables

TZ determines the time zone name substituted for the **%Z** and **%z** directives. The time zone name is determined by calling the function `tzset()` which sets the external variable `tzname` (see *ctime*(3C)).

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUE

If the total number of resulting wide characters including the terminating null wide character is not more than `maxsize`, `wcsftime()` returns the number of wide characters placed into the array pointed to by *ws*, not including the terminating null wide character. Otherwise, zero is returned and the contents of the array are indeterminate.

EXAMPLES

If the *timeptr* argument contains the following values:

```
timeptr->tm_sec = 4;
timeptr->tm_min = 9;
timeptr->tm_hour = 15;
timeptr->tm_mday = 4;
timeptr->tm_mon = 6;
timeptr->tm_year = 88;
timeptr->tm_wday = 1;
timeptr->tm_yday = 185;
timeptr->tm_isdst = 1;
```

the following combinations of the `LC_TIME` category and format strings produce the indicated output:

LC_TIME	Format String	Output
american	%x	Mon, Jul 4, 1988
german	%x	Mo., 4. Juli 1988
american	%X	03:09:04 PM
french	%X	15h09 04
any†	%H:%M:%S	15:09:04
any†	%.1H:%.1M:%.1S	15:9:4
any†	%2.1H:%-3M:%03.1S	15:9 :004

† The directives used in these examples are not affected by the `LC_TIME` category of the locale.

WARNINGS

The function `tzset()` is called upon every invocation of `wcsftime()` (whether or not the time zone name is copied to the output array).

The range of values for `%S` ([0,61]) extends to 61 to allow for the occasional one or two leap seconds. However, the system does not accumulate leap seconds and the `tm` structure generated by the functions `localtime()` and `gmtime()` (see *ctime*(3C)) never reflects any leap seconds.

Results are undefined if values contained in the structure pointed to by *timeptr* exceed the ranges defined for the `tm` structure (see *ctime*(3C)) or are not consistent (such as if the `tm_yday` element is set to 0, indicating the first day of January, while the `tm_mon` element is set to 11, indicating a day in December).

AUTHOR

`wcsftime()` was developed by HP.

SEE ALSO

date(1), ctime(3C), setlocale(3C), environ(5), langinfo(5), hpnls(5).

STANDARDS CONFORMANCE

`wcsftime()`: XPG4

NAME

wcstod() - convert wide character string to double-precision number

SYNOPSIS

```
#include <wchar.h>

double wcstod(const wchar_t *nptr, wchar_t **endptr);
```

Remarks:

This function is compliant with the XPG4 Worldwide Portability Interface wide-character formatting functions. It parallels the 8-bit character formatting function defined in *strtod*(3C).

DESCRIPTION

`wcstod()` returns, as a double-precision floating-point number, the value represented by the wide character string pointed to by *nptr*. The wide character string is scanned (leading white-space characters as defined by `iswspace()` in *wctype*(3C) are ignored) up to the first unrecognized character. If no conversion can take place, zero is returned.

`wcstod()` recognizes wide characters in the following sequence:

1. An optional string of "white-space" wide characters which are ignored,
2. An optional sign,
3. A string of digits optionally containing a radix character,
4. An optional `e` or `E` followed by an optional sign or space, followed by an integer.

The radix character is determined by the current NLS environment (see *setlocale*(3C)). If `setlocale()` has not been called successfully, the default NLS environment, "C", is used (see *lang*(5)). The default environment specifies a period (.) as the radix character.

If the value of *endptr* is not `(wchar_t **)NULL`, the variable to which it points is set to point at the wide character after the last number, if any, that was recognized. If no number can be formed, ***endptr* is set to *nptr*, and zero is returned.

The definition for this function and the type `wchar_t` are provided in the <`wchar.h`> header.

EXTERNAL INFLUENCES

Locale

The `LC_NUMERIC` category determines the value of the radix character within the currently loaded NLS environment.

The `LC_CTYPE` category determines how wide character codes are interpreted.

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUE

If the correct value would cause overflow, `+HUGE_VAL` or `-HUGE_VAL` is returned (according to the sign of the value), and `errno` is set to ERANGE.

If the correct value would cause underflow, zero is returned and `errno` is set to ERANGE.

AUTHOR

`wcstod()` was developed by AT&T and HP.

SEE ALSO

wctype(3C), setlocale(3C), scanf(3S), wcstol(3C), hpnls(5), lang(5).

STANDARDS CONFORMANCE

`wcstod()`: XPG4

NAME

wcstol(), wcstoul() - convert wide character string to long integer

SYNOPSIS

```
#include <wchar.h>

long int wcstol(const wchar_t *nptr, wchar_t **endptr, int base);

unsigned long int wcstoul(const wchar_t *nptr, wchar_t **endptr, int
base);
```

Remarks:

These functions are compliant with the XPG4 Worldwide Portability Interface wide-character formatting functions. They parallel the 8-bit character formatting functions defined in *strtol*(3C).

DESCRIPTION

`wcstol()` (`wcstoul()`) converts the wide character string pointed to by *nptr* to `long int` (`unsigned long int`) representation. The wide character string is scanned up to the first wide character inconsistent with the base. Leading "white-space" wide characters (as defined by `iswspace()` in *wctype*(3C)) are ignored. If no conversion can take place, zero is returned.

If *base* is greater than or equal to 2 and less than or equal to 36, it is used as the base for conversion. After an optional leading sign, leading zeros are ignored, and `0x` or `0X` is ignored if *base* is 16.

If *base* is zero, the wide character string itself determines the base as follows: After an optional leading sign, a leading zero indicates octal conversion; a leading `0x` or `0X` hexadecimal conversion. Otherwise, decimal conversion is used.

If the value of *endptr* is not `(wchar_t **)NULL`, a pointer to the wide character terminating the scan is returned in the location pointed to by *endptr*. If no integer can be formed, the location pointed to by *endptr* is set to *nptr*, and zero is returned.

Definitions for these functions and the type `wchar_t` are provided in the <`wchar.h`> header.

EXTERNAL INFLUENCES

Locale

The `LC_CTYPE` category determines how wide character codes are interpreted.

International Code Set Support

Single- and multi-byte character code sets are supported.

RETURN VALUE

Upon successful completion, both functions return the converted value, if any. If the correct value would cause overflow, `wcstol()` returns `LONG_MAX` or `LONG_MIN` (according to the sign of the value), and sets `errno` to ERANGE; `wcstoul()` returns `ULONG_MAX` and sets `errno` to ERANGE.

For all other errors, zero is returned and `errno` is set to indicate the error.

ERRORS

`wcstol()` and `wcstoul()` fail and `errno` is set if any of the following conditions are encountered:

[EINVAL]	The value of *base* is not supported.
[ERANGE]	The value to be returned would have caused overflow.

SEE ALSO

wctype(3C), wcstod(3C), scanf(3S).

STANDARDS CONFORMANCE

`wcstol()`: XPG4

`wcstoul()`: XPG4

NAME

wcscat(), wcsncat(), wcscmp(), wcsncmp(), wcscpy(), wcsncpy(), wcslen(), wcschr(), wcsrchr(), wcspbrk(), wcsspn(), wcscspn(), wcswcs(), wcstok(), wcscoll(), wcwidth(), wcswidth() - wide character string operations

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcscat(wchar_t *ws1, const wchar_t *ws2);

wchar_t *wcsncat(wchar_t *ws1, const wchar_t *ws2, size_t n);

int wcscmp(const wchar_t *ws1, const wchar_t *ws2);

int wcsncmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);

wchar_t *wcscpy(wchar_t *ws1, const wchar_t *ws2);

wchar_t *wcsncpy(wchar_t *ws1, const wchar_t *ws2, size_t n);

size_t wcslen(const wchar_t *ws);

wchar_t *wcschr(const wchar_t *ws, wchar_t wc);

wchar_t *wcsrchr(const wchar_t *ws, wchar_t wc);

wchar_t *wcspbrk(const wchar_t *ws1, const wchar_t *ws2);

size_t wcsspn(const wchar_t *ws1, const wchar_t *ws2);

size_t wcscspn(const wchar_t *ws1, const wchar_t *ws2);

wchar_t *wcswcs(const wchar_t *ws1, const wchar_t *ws2);

wchar_t *wcstok(wchar_t *ws1, const wchar_t *ws2);

int wcscoll(const wchar_t *ws1, const wchar_t *ws2);

int wcwidth(const wchar_t wc);

int wcswidth(const wchar_t *ws, size_t n);
```

Remarks:

These functions are compliant with the XPG4 Worldwide Portability Interface wide-character string handling functions. They parallel the 8 bit string functions defined in *string(3C)*.

DESCRIPTION

The arguments *ws1*, *ws2*, and *ws* point to wide character strings (arrays of type `wchar_t` terminated by a null value).

`wcscat()` appends a copy of wide string *ws2* to the end of wide string *ws1*. `wcsncat()` appends a maximum of *n* characters; fewer if *ws2* is shorter than *n* characters. Each returns a pointer to the null-terminated result (the value of *ws1*).

`wcscmp()` compares its arguments and returns an integer less than, equal to, or greater than zero, depending on whether *ws1* is lexicographically less than, equal to, or greater than *ws2*. The comparison of corresponding wide characters is done by comparing numeric values of the wide character codes. Null pointer values for *ws1* and *ws2* are treated the same as pointers to empty wide strings. `wcsncmp()` makes the same comparison but examines a maximum of *n* characters (*n* less than or equal to zero yields equality).

`wcscpy()` copies wide string *ws2* to *ws1*, stopping after the null value has been copied. `wcsncpy()` copies up to *n* characters from *ws2*, adding null values to *ws1* if necessary, until a total of *n* have been copied. The result is not null-terminated if the length of *ws2* is *n* or more. Each function returns *ws1*. Note that `wcsncpy()` should not be used to copy an arbitrary structure. If that structure contains sizeof(wchar_t) consecutive null bytes, `wcsncpy()` may not copy the entire structure. Use the `memcpy()` function (see *memory*(3C)) to copy arbitrary binary data.

`wcslen()` returns the number of wide characters in *ws*, not including the terminating null wide character.

`wcschr()` (`wcsrchr()`) returns a pointer to the first (last) occurrence of wide character *wc* in wide string *ws*, or a null pointer if *wc* does not occur in the wide string. The null wide character terminating a wide

string is considered to be part of the wide string.

`wcspbrk()` returns a pointer to the first occurrence in wide string *ws1* of any wide character from wide string *ws2*, or a null pointer if no wide character from *ws2* exists in *ws1*.

`wcsspn()` (`wcscspn()`) returns the length of the maximum initial segment of wide string *ws1*, which consists entirely of wide characters from (not from) wide string *ws2*.

`wcswcs()` returns a pointer to the first occurrence of wide string *ws2* in wide string *ws1*, or a null pointer if *ws2* does not occur in the wide string. If *ws2* points to a wide string of zero length, `wcswcs()` returns *ws1*.

`wcstok()` considers the wide string *ws1* to consist of a sequence of zero or more text tokens separated by spans of one or more wide characters from the separator wide string *ws2*. The first call (with a non-null pointer *ws1* specified) returns a pointer to the first wide character of the first token, and writes a null wide character into *ws1* immediately following the returned token. The function keeps track of its position in the wide string *ws1* between separate calls, so that subsequent calls made with the first argument a null pointer work through the wide string immediately following that token. In this way subsequent calls work through the wide string *ws1* until no tokens remain. The separator wide string *ws2* can be different from call to call. When no token remains in *ws1*, a null pointer is returned.

`wcscoll()` returns an integer greater than, equal to, or less than zero, according to whether the wide string pointed to by *ws1* is greater than, equal to, or less than the wide string pointed to by *ws2*. The comparison is based on wide strings interpreted as appropriate to the program's locale (see Locale below). In the "C" locale `wcscoll()` works like `wcscmp()`.

`wcwidth()` returns the number of column positions required for the wide character *wc*, or 0 if *wc* is a null wide character.

`wcswidth()` returns the number of column positions required for *n* wide characters (or fewer than *n* wide characters if a null wide character is encountered before *n* wide characters are exhausted) in the wide string pointed to by *ws*. `wcswidth()` returns or 0 if *ws* points to a null wide character.

Definitions for these functions and the type `wchar_t` are provided in header file <`wchar.h`>.

EXTERNAL INFLUENCES

Locale

The `LC_COLLATE` category determines the collation ordering used by the `wcscoll()` function. See *nlsinfo*(1) to determine the collation used for a particular locale.

The `LC_CTYPE` category determines how widths are calculated by the `wcwidth()` and `wcswidth()` functions.

WARNINGS

The functions `wcscat()`, `wcsncat()`, `wcscpy()`, `wcsncpy()`, and `wcstok()` alter the contents of the array to which *ws1* points. They do not check for overflow of the array.

Null pointers for destination wide strings cause undefined behavior.

Wide character movement is performed differently in different implementations, so copying that involves overlapping source and destination wide strings may yield unexpected results.

For the `wcscoll()` function, the results are undefined if the languages specified by the `LC_COLLATE` and `LC_CTYPE` categories use different code sets.

AUTHOR

`wcstring` functions were developed by HP.

SEE ALSO

nlsinfo(1), wconv(3C), memory(3C), multibyte(3C), setlocale(3C), string(3C), hpnls(5).

STANDARDS CONFORMANCE

`wcscat()`: XPG4
`wcschr()`: XPG4
`wcscmp()`: XPG4
`wcscoll()`: XPG4

wcscpy():XPG4
wcscspn():XPG4

wcslen():XPG4
wcsncat():XPG4

wcsncmp():XPG4
wcsncpy():XPG4

wcspbrk():XPG4
wcsrchr():XPG4

wcsspn():XPG4
wcstok():XPG4

wcswcs():XPG4
wcswidth():XPG4

wcwidth():XPG4

NAME

iswalpha(), iswupper(), iswlower(), iswdigit(), iswxdigit(), iswalnum(), iswspace(), iswpunct(), iswprint(), iswgraph(), iswcntrl(), wctype(), iswctype() - classify wide characters

SYNOPSIS

```
#include <wchar.h>

wctype_t wctype(const char *charclass);
int iswctype(wint_t wc, wctype_t prop);
int iswalnum(wint_t wc);
int iswalpha(wint_t wc);
int iswcntrl(wint_t wc);
int iswdigit(wint_t wc);
int iswgraph(wint_t wc);
int iswlower(wint_t wc);
int iswprint(wint_t wc);
int iswpunct(wint_t wc);
int iswspace(wint_t wc);
int iswupper(wint_t wc);
int iswxdigit(wint_t wc);
```

Remarks:

These functions are compliant with the XPG4 Worldwide Portability Interface wide-character classification functions. They parallel the 8-bit character classification functions defined in *ctype(3C)*.

DESCRIPTION

These functions classify wide character values according to the rules of the coded character set identified by the last successful call to **setlocale()** (see *setlocale*(3C)).

If **setlocale()** has not been called successfully, characters are classified according to the rules of the default ASCII 7-bit coded character set (see *setlocale*(3C)).

Each of the classification functions is a predicate that returns non-zero for true, zero for false.

wctype() is defined for valid character class names as defined in the current locale. *charclass* is a string identifying a generic character class for which codeset-specific type information is required. The following class names are defined in all locales: **alnum**, **alpha**, **blank**, **cntrl**, **digit**, **graph**, **lower**, **print**, **punct**, **space**, **upper**, and **xdigit**. **wctype()** returns a value of type **wctype_t** that can be used in a subsequent call to **iswctype()**, or **(wctype_t)-1** if *charclass* is not valid in the current locale.

The classification functions return non-zero under the following circumstances, and zero otherwise:

iswctype(*wc*,*prop*) — *wc* has the property defined by *prop*.

iswalpha(*wc*) — *wc* is a letter.

iswupper(*wc*) — *wc* is an uppercase letter.

iswlower(*wc*) — *wc* is a lowercase letter.

iswdigit(*wc*) — *wc* is a decimal digit (in ASCII: characters [0-9]).

iswxdigit(*wc*) — *wc* is a hexadecimal digit (in ASCII: characters [0-9], [A-F] or [a-f]).

iswalnum(*wc*) — *wc* is an alphanumeric (letters or digits).

iswspace(*wc*) — *wc* is a character that creates "white space" in displayed text (in ASCII: space, tab, carriage return, new-line, vertical tab, and form-feed).

iswpunct(*wc*) — *wc* is a punctuation character (in ASCII: any printing character except the space character (040), digits, letters).

iswprint(*wc*) — *wc* is a printing character.

iswgraph(*wc*) — *wc* is a visible character (in ASCII: printing characters, excluding the space character (040)).

`iswcntrl(wc)` *wc* is a control character (in ASCII: character codes less than 040 and the delete character (0177)).

If the argument to any of these functions is outside the domain of the function, the result is 0 (false).

Definitions for these functions and the types `wint_t`, `wchar_t`, and `wctype_t` are provided in the <wchar.h> header.

EXTERNAL INFLUENCES

Locale

The `LC_CTYPE` category determines the classification of character type.

International Code Set Support

Single-byte character code sets are supported. Japanese HP15 and EUC multi-byte character code sets are supported. The classification functions return zero for values in other multi-byte character code sets outside the ASCII range.

WARNINGS

These functions are supplied both as library functions and as macros defined in the <wchar.h> header. Normally, the macro versions are used. To obtain the library function, either use a `#undef` to remove the macro definition or, if compiling in ANSI-C mode, enclose the function name in parenthesis or take its address. The following example uses the library functions for `iswalpha()`, `iswdigit()`, and `iswspace()`:

```
#include <wchar.h>
#undef iswalpha
        ...
main()
{
    int (*ctype_func)();
            ..
    if ( iswalpha(c) )
            ..
    if ( (iswdigit)(c) )
            ..
    ctype_func = iswspace;
            ..
}
```

AUTHOR

`wctype()` was developed by AT&T and HP.

SEE ALSO

ctype(3C), multibyte(3C), setlocale(3C), ascii(5).

STANDARDS CONFORMANCE

`wctype()`: XPG4
`iswctype()`: XPG4
`iswalnum()`: XPG4
`iswalpha()`: XPG4
`iswcntrl()`: XPG4
`iswdigit()`: XPG4
`iswgraph()`: XPG4
`iswlower()`: XPG4
`iswprint()`: XPG4
`iswpunct()`: XPG4
`iswspace()`: XPG4
`iswupper()`: XPG4
`iswxdigit()`: XPG4

NAME

wordexp, wordfree - perform word expansions

SYNOPSIS

```
#include <wordexp.h>

int wordexp(const char *words, wordexp_t *pwordexp, int flags);

void wordfree(wordexp_t *pwordexp);
```

DESCRIPTION

`wordexp()` performs word expansions and places the list of expanded words into the structure pointed to by *pwordexp*.

The *words* argument is a pointer to a string containing one or more words to be expanded. The expansions are the same as would be performed by the shell (see *sh-posix*(1), if words were the part of a command line representing the arguments to a utility. Therefore, *words* must not contain an unquoted new-line character or any of the unquoted shell special characters `|`, `&`, `;`, `<` or `>`, except in the context of shell command substitution. If *words* contains an unquoted comment character, `#`, it is treated as the beginnning of a token which `wordexp()` interprets as a comment indicator, causing the remainder of *words* to be ignored.

The structure type `wordexp_t` is defined in the header `<wordexp.h>`, and includes the following members:

`we_wordc`	A `size_t` used to keep count of words matched by *words*.
`we_wordv`	A `char**` used as a pointer to a list of expanded words.
`we_offs`	Also a `size_t` used to indicate number of slots to reserve at the the beginning of pwordexp->we_wordv.

`wordexp()` stores the number of generated words into `pwordexp->we_wordv`. Each individual field created during field splitting or pathname expansion is a separated word in the `pwordexp->we_wordv` list. The words are in order as described in shell word expansions. The first pointer after the last word pointer is a null pointer.

It is the caller's responsibility to allocate the storage pointed to by *pwordexp*. `wordexp()` allocates other space as needed, including memory pointed to by `pwordexp->we_wordv`.

wordfree() frees any memory associated with *pwordexp* from a previous call to `wordexp()`.

The *flags* argument is used to control the behavior of `wordexp()`. The value of flags is the bitwise inclusive OR of zero or more of the following constants, which are defined in `<wordexp.h>`:

`WRDE_APPEND`	Append words generated to the ones from a previous call to `wordexp()`.
`WRDE_DOOFFS`	Make use of `pwordexp->we_offs`. If this flag is set, `pwordexp->we_offs` is used to specify how many null pointers to add to the beginning of `pwordexp->we_wordv`. In other words, `pwordexp->we_wordv` points to `pwordexp->we_offs` null pointers, followed by `pwordexp->we_wordc` word pointers, followed a null pointer.
`WRDE_NOCMD`	Fail if command substitution is requested.
`WRDE_REUSE`	The *pwordexp* argument was passed to a previous successful call to `wordexp()`, and has not been passed to `wordfree()`. The result is the same as if the application had called `wordfree()` and then called `wordexp()` without `WRDE_REUSE`.
`WRDE_SHOWERR`	Do not redirect *stderr* to `/dev/null`.
`WRDE_UNDEF`	Report error on an attempt to expand an undefined shell variable.

The `WRDE_APPEND` flag can be used to append a new sets of words to those generated by a previous call to `wordexp()`. The following rules apply when two or more calls to `wordexp()` are made with the same value of *pwordexp* and without intervening calls to `wordfree()`:

- The first such call must not set `WRDE_APPEND`. All subsequent calls must set it.

- All of the calls must set WRDE_DOOFFS, or all must not set it.
- After the second and each subsequent call, pwordexp->we_wordv points to a list containing the following:
  - Zero or more null pointers, as specified by WRED_DOOFFS and pwordexp->we_offs.
  - Pointers to the words that were in the pwordexp->we_wordv list before the call, in the same order as before.
  - Pointers to the new words generated by the latest call, in the specified order.
- The count returned in *pwordexp->we_wordc* is the total number of words from all of the calls.
- The application can change any of the fields after a call to wordexp(), but if it does so, if must reset them to the original value before a subsequent call, using the same *pwordexp* value, to wordfree() or wordexp() with the WRDE_APPEND or WRDE_REUSE flag.

If *words* contains an unquoted newline, |, &, ;, <, >, parenthesis, or curly barcket in an inappropriate context, wordexp() fails, and the number of expanded words is zero.

Unless WRDE_SHOWERR is set in *flags*, wordexp() redirects *stderr* to /dev/null for any utilities executed as a result of command subsitution while expanding *words*. If WRDE_SHOWERR is set, wordexp() writes messages to *stderr* if syntax errors are detected while expanding *words*.

If WRDE_DOOFFS is set, pwordexp->we_offs has the same value for each wordexp() call and the wordfree() call using a given *wordexp*.

RETURN VALUE

Upon successful completion, wordexp() returns zero; otherwise, it returns a nonzero value defined in <wordexp.h> to indicate the error:

WRDE_BADCHAR	One of the unquoted characters |, &, ;, <, >, parentheses, or braces appears in *words* in an inappropriate context
WRDE_BADVAL	Reference to undefined shell variable when WRDE_UNDEF is set in *flags*.
WRDE_CMDSUB	Command substitution requested when WRDE_NOCMD was set in *flags*.
WRDE_NOSPACE	Attempt to allocate memory failed.
WRDE_SYNTAX	Shell syntax error such as unbalanced parentheses or unterminated string.
WRDE_INTERNAL	Internal error.

If wordexp() returns the error value WRDE_NOSPACE, the pwordexp->we_wordc and pwordexp->we_wordv are updated to reflect any words that were successfully expanded. In other cases, they are not modified.

SEE ALSO

sh-posix(1), fnmatch(3C), glob(3C), regexp(5).

STANDARDS CONFORMANCE

wordexp(): XPG4, POSIX.2

wordfree(): XPG4, POSIX.2

NAME

xdr*() - library routines for external data representation

DESCRIPTION

These routines allow C programmers to describe arbitrary data structures in a machine-independent fashion. Data for remote procedure calls are transmitted using these routines.

Functions

`xdr_array()`	Translate arrays to/from external representation.
`xdr_bool()`	Translate Booleans to/from external representation.
`xdr_bytes()`	Translate counted byte strings to/from external representation.
`xdr_char()`	Translate characters to/from external representation.
`xdr_destroy()`	Destroy XDR stream and free associated memory.
`xdr_double()`	Translate double precision to/from external representation.
`xdr_enum()`	Translate enumerations to/from external representation.
`xdr_float()`	Translate floating point to/from external representation.
`xdr_getpos()`	Return current position in XDR stream.
`xdr_free()`	Free the memory allocated to create XDR data structures.
`xdr_inline()`	Invoke the in-line routines associated with XDR stream.
`xdr_int()`	Translate integers to/from external representation.
`xdr_long()`	Translate long integers to/from external representation.
`xdr_opaque()`	Translate fixed-size opaque data to/from external representation.
`xdr_pointer()`	Similar to `xdr_reference()` except it is able to follow recursive data structures such as a binary tree.
`xdr_reference()`	Chase pointers within structures.
`xdr_setpos()`	Change current position in XDR stream.
`xdr_short()`	Translate short integers to/from external representation.
`xdr_string()`	Translate null-terminated strings to/from external representation.
`xdr_u_char()`	Translate unsigned characters to/from external representation.
`xdr_u_int()`	Translate unsigned integers to/from external representation.
`xdr_u_long()`	Translate unsigned long integers to/from external representation.
`xdr_u_short()`	Translate unsigned short integers to/from external representation.
`xdr_union()`	Translate descriminated unions to/from external representation.
`xdr_vector()`	Translate fixed-length arrays to/from external representation.
`xdr_void()`	Always return one (1).
`xdr_wrapstring()`	Package RPC routine for XDR routine, or vice-versa.
`xdrmem_create()`	Initialize an XDR stream.
`xdrrec_create()`	Initialize an XDR stream with record boundaries.
`xdrrec_endofrecord()`	Mark XDR record stream with an end-of-record.
`xdrrec_eof()`	Mark XDR record stream with an end-of-file.
`xdrrec_skiprecord()`	Skip remaining record in XDR record stream.

xdrstdio_create() Initialize an XDR stream as standard I/O FILE stream.

AUTHOR

xdr*() was developed by Sun Microsystems, Inc.

SEE ALSO

Programming and Protocols for NFS Services.

NAME

ypclnt(), yp_all(), yp_bind(), yp_first(), yp_get_default_domain(), yp_master(), yp_match(), yp_next(), yp_order(), yp_unbind(), yperr_string(), ypprot_err() - Network Information Service client interface

SYNOPSIS

```
#include <rpcsvc/ypclnt.h>

#include <sys/types.h>
#include <rpc/rpc.h>
#include <rpcsvc/yp_prot.h>

int yp_all(
      char *indomain,
      char *inmap,
      struct ypall_callback incallback
);

int yp_bind(char *indomain);

int yp_first(
      char *indomain,
      char *inmap,
      char **outkey,
      int *outkeylen,
      char **outval,
      int *outvallen
);

int yp_get_default_domain(char **outdomain);

int yp_master(
      char *indomain,
      char *inmap,
      char **outmaster
);

int yp_match(
      char *indomain,
      char *inmap,
      char *inkey,
      int inkeylen,
      char **outval,
      int *outvallen
);

int yp_next(
      char *indomain,
      char *inmap,
      char *inkey,
      int inkeylen,
      char **outkey,
      int *outkeylen,
      char **outval,
      int *outvallen
);

int yp_order(
      char *indomain,
      char *inmap,
      unsigned long *outorder
);

void yp_unbind(char *indomain);
```

```
char *yperr_string(int incode);
int ypprot_err(unsigned int incode);
```

DESCRIPTION

These functions provide an interface to the Network Information Service (NIS) network-lookup service. The package can be loaded from the library `/lib/libc.a`. Refer to *ypfiles*(4) and *ypserv*(1M) for an overview of the NIS, including the definitions of *map* and NIS *domain*, and a description of the various servers, databases, and commands comprising the NIS.

Input parameter names begin with `in`; output parameter names begin with `out`. Output parameters of type `char **` should be the addresses of uninitialized character pointers. Memory is allocated by the NIS client package using `malloc()` and can be freed if the user code has no continuing need for it (see *malloc*(3C)). For each *outkey* and *outval*, two extra bytes of memory are allocated at the end that contain new-line and null (in that order), but these two bytes are not reflected in *outkeylen* and *outvallen*. The *indomain* and *inmap* strings must be non-null and null-terminated. String parameters that are accompanied by a length parameter cannot be null, but can point to null strings with a length parameter of zero. Counted strings need not be null-terminated.

The NIS lookup calls require a map (database) name and a NIS domain name. The client process should know the name of the map of interest. Client processes should obtain the host's NIS domain by calling `yp_get_default_domain()` and use the returned *outdomain* as the *indomain* parameter to subsequent NIS calls.

To use the NIS services, the client process must be "bound" to an NIS server that serves the appropriate NIS domain using `yp_bind()`. Binding does not have to occur explicitly by user code. Rather, it occurs automatically whenever a NIS lookup function is called. `yp_bind()` can be called directly for processes that use a backup strategy (such as a local file) when NIS services are not available.

Each binding allocates (uses up) one client process socket descriptor. Each bound NIS domain costs one socket descriptor. However, multiple requests to the same NIS domain use that same descriptor. `yp_unbind()` is available at the client interface for processes that explicitly manage their socket descriptors while accessing multiple NIS domains. The call to `yp_unbind()` makes the NIS domain **unbound** and frees all per-process and per-node resources used to bind it.

If an RPC failure results when using a binding, that NIS domain is unbound automatically. The ypclnt layer then continues retrying until the operation succeeds, provided `ypbind` is running (see *ypbind*(1M)) and either:

a. the client process cannot bind a server for the proper NIS domain, or

b. RPC requests to the server fail.

If an error is not RPC-related, if `ypbind` is not running, or if a bound `ypserv` process returns any answer (success or failure), the ypclnt layer returns control to the user code with either an error code or with a success code and any results (see *ypbind*(1M) and *ypserv*(1M)).

Operational Behavior

`yp_match()` Returns the value associated with a passed key. This key must be exact; no pattern matching is available.

`yp_first()` Returns the first key-value pair from the named map in the named NIS domain.

`yp_next()` Returns the next key-value pair in a named map. To obtain the second key-value pair, the *inkey* parameter should be the *outkey* returned from an initial call to `yp_first()`. To obtain the (*n* + *1*)th*key-value*pair,*the inkey* value should be the *outkey* value from the *n*th call to `yp_next()`.

The concepts of first and next are particular to the structure of the NIS map being processed. No relation in retrieval order exists to either the lexical order within any original ASCII file or to any obvious numerical sorting order on the keys, values, or key-value pairs. The only ordering guarantee is that if the `yp_first()` function is called on a particular map and the `yp_next()` function is called repeatedly on the same map at the same server until the call fails with an error of YPERR_NOMORE, every entry in the database is retrieved exactly once. If the same sequence of operations is performed on the same map at the same server, the entries are retrieved in

the same order.

Under conditions of heavy server load or server failure, the NIS domain may become unbound and bind again (perhaps to a different server) while a client is running. This process can cause a break in one of the enumeration (retrieval) rules: specific entries may be seen twice by the client or not at all. This approach protects the client from error messages that would otherwise be returned in the midst of the enumeration.

`yp_all()` describes a better solution to enumerating all entries in a map.

`yp_all()` Provides a way to transfer an entire map from server to client in a single request using TCP (rather than UDP as with other functions in this package). The entire transaction occurs as a single RPC request and response. You can use `yp_all()` like any other NIS procedure by identifying the map in the normal manner and supplying the name of a function called to process each key-value pair within the map. A return from the call to `yp_all()` occurs only when the transaction is completed (either successfully or unsuccessfully) or the *foreach* function decides it does not want any more key-value pairs.

The third parameter to `yp_all()` is:

```
struct ypall_callback *incallback {
    int (*foreach)();
    char *data;
};
```

The function `foreach()` is called as follows:

```
foreach(
    int instatus;
    char *inkey;
    int inkeylen;
    char *inval;
    int invallen;
    char *indata;
);
```

Where:

instatus Holds one of the return status values defined in `<rpcsvc/yp_prot.h>`: either `YP_TRUE` or an error code (see `ypprot_err()` below, for a function that converts a NIS protocol error code to a ypclnt layer error code, as defined in `<rpcsvc/ypclnt.h>`).

inkey
inval The key and value parameters are somewhat different than defined in the SYNOPSIS section above. First, the memory pointed to by *inkey* and *inval* is private to `yp_all()`, and is overwritten with the arrival of each new key-value pair. Therefore, `foreach()` should do something useful with the contents of that memory, but it does not own the memory. Key and value objects presented to the `foreach()` look exactly as they do in the server's map. Therefore, if they were not newline-terminated or null-terminated in the map, they will not be terminated with newline or null characters here, either.

indata Is the contents of the *incallback->data* element passed to `yp_all()` The *data* element of the callback structure can share state information between `foreach()` and the mainline code. Its use is optional, and no part of the NIS client package inspects its contents. Cast it to something useful or ignore it as appropriate.

The `foreach()` function is Boolean. It should return zero to indicate it needs to be called again for further received key-value pairs, or non-zero to stop the flow of key-value pairs. If `foreach()` returns a non-zero value, it is not called again and the functional value of `yp_all()` is then 0.

`yp_order()`
Returns the order number for a map.

`yp_master()`
Returns the host name of the master NIS server for a map.

`yperr_string()`
Returns a pointer to an error message string that is null-terminated, but contains no period or newline.

`ypprot_err()`
Takes an NIS protocol error code as input and returns a ypclnt layer error code that can be used as input to `yperr_string()`

RETURN VALUE

All functions in this package of type `int` return 0 if the requested operation is successful or one of the following errors if the operation fails.

[YPERR_BADARGS]	args to function are bad
[YPERR_BADDB]	NIS map is defective
[YPERR_DOMAIN]	cannot bind to server on this NIS domain
[YPERR_KEY]	no such key in map
[YPERR_MAP]	no such map in server's NIS domain
[YPERR_NODOM]	local NIS domain name not set
[YPERR_NOMORE]	no more records in map
[YPERR_PMAP]	cannot communicate with portmap
[YPERR_RESRC]	resource allocation failure
[YPERR_RPC]	RPC failure – NIS domain has been unbound
[YPERR_VERS]	NIS client/server version mismatch: the NIS server bound to uses Version 1 protocol, so it does not provide `yp_all()` functionality
[YPERR_YPBIND]	cannot communicate with ypbind
[YPERR_YPERR]	internal NIS server or client error
[YPERR_YPSERV]	cannot communicate with ypserv

AUTHOR

`ypclnt()` was developed by Sun Microsystems, Inc.

SEE ALSO

domainname(1), rpcinfo(1M), ypserv(1M), ypfiles(4).

NAME

yppasswd() - update user password in Network Information Service

SYNOPSIS

```
#include <pwd.h>
#include <rpcsvc/yppasswd.h>

int yppasswd(char *oldpass, struct passwd *newpw);
```

DESCRIPTION

If *oldpass* is the old, unencrypted user password, this routine replaces the password entry with the encrypted *newpw*.

RPC Info

program number:
YPPASSWDPROG

xdr routines:

```
xdr_yppasswd(xdrs, yp)
  XDR *xdrs;
  struct yppasswd *yp;
xdr_passwd(xdrs, pw)
  XDR *xdrs;
  struct passwd *pw;
```

procs:

YPPASSWDPROC_UPDATE
Takes *struct yppasswd* as an argument; returns an integer.
Behaves the same as the *yppasswd()* function.
Uses UNIX authentication.

versions:
YPPASSWDVERS

structures:

```
struct yppasswd {
  char *oldpass;          /* old (unencrypted) password */
  struct passwd newpw;          /* new pw structure */
};
```

RETURN VALUE

yppasswd() returns 0 if successful and –1 if an error occurs.

AUTHOR

`yppasswd()` was developed by Sun Microsystems, Inc.

SEE ALSO

yppasswd(1), yppasswdd(1M).

Index to Volume 2

Index
Volume 2

Description	Entry Name(Section)
a64l() – convert base-64 value to long integer ASCII string	**a64l(3C)**
AAudioString() – get name of audio controller (string) passed to AOpenAudio()	**AAudioString(3X)**
ABestAudioAttributes() – get best audio attributes for controller	**ABestAudioAttributes(3X)**
abort a per-process timer	**rmtimer(3C)**
abort() – generate an IOT fault	**abort(3C)**
abs(), **labs()** – return integer absolute value	**abs(3C)**
absolute value, floor, ceiling, remainder, round-to-nearest functions	**floor(3M)**
absolute value function, complex	**hypot(3M)**
absolute value, return integer	**abs(3C)**
ACalculateLength() – return the size in bytes of converted data	**ACalculateLength(3X)**
accelerator, math, check for presence of	**is_hw_present(3C)**
accept() – accept connection on a socket	**accept(2)**
access and modification times, set or update file	**utime(2)**
access control list (ACL), change owner and/or group in	**chownacl(3C)**
access control list (ACL), copy to another file	**cpacl(3C)**
access control list (ACL) information, get	**getacl(2)**
access control list (ACL) information, set	**setacl(2)**
access control list (ACL) structure, convert string form to	**strtoacl(3C)**
access control list (ACL) structure, convert to string form	**acltostr(3C)**
access control list; add, modify, or delete entry	**setaclentry(3C)**
access() – determine accessibility of a file	**access(2)**
access exported file system information	**exportent(3N)**
accessibility of a file, determine	**access(2)**
access list, get group	**getgroups(2)**
access list, initialize group	**initgroups(3C)**
access list, set group	**setgroups(2)**
access long integer data in a machine-independent fashion	**sputl(3X)**
access mode (permissions) of file, change	**chmod(2)**
access, open, or close a directory and associated *directory stream*	**directory(3C)**
access or build a binary search tree	**tsearch(3C)**
access protections, modify memory mapping	**mprotect(2)**
access rights to a file, get a user's effective	**getaccess(2)**
access **utmp()** or **wtmp()** file	**getut(3C)**
accounting: enable or disable process accounting	**acct(2)**
acct() – enable or disable process accounting	**acct(2)**
ACheckEvent() – get first event found in audio event queue	**ACheckEvent(3X)**
ACheckMaskEvent() – get first event in audio event queue that matches mask	**ACheckMaskEvent(3X)**
AChooseAFileAttributes() – select attributes for creating new file	**AChooseAFileAttributes(3X)**
AChoosePlayAttributes() – select attributes for playing existing file	**AChoosePlayAttributes(3X)**
AChooseSourceAttributes() – select attributes for existing file or stream	**AChooseSourceAttributes(3X)**
aclentrystart() – convert pattern string form to access control list (ACL) structure	**strtoacl(3C)**
ACloseAudio() – close connection to specific audio server	**ACloseAudio(3X)**
acltostr() – convert access control list (ACL) structure to string form	**acltostr(3C)**
AConnectionNumber() – get audio server connection number	**AConnectionNumber(3X)**
AConnectRecordStream() – connect socket to TCP socket address	**AConnectRecordStream(3X)**
AConvertAFile() – convert audio file data format	**AConvertAFile(3X)**
AConvertBuffer() – convert a buffer of data	**AConvertBuffer(3X)**
acosdf() – trigonometric arcosine function (float, degrees)	**trigd(3M)**
acosd() – trigonometric arcosine function (degrees)	**trigd(3M)**
acosf() – trigonometric arcosine function (float)	**trig(3M)**
acos() – trigonometric arcosine function	**trig(3M)**
acquire exclusive use of audio server	**AGrabServer(3X)**
ACreateSBucket() – create empty sound bucket and return pointer to it	**ACreateSBucket(3X)**
active controllers on HP-IB, change	**hpib_pass_ctl(3I)**
activity on specified HP-IB bus, stop	**hpib_abort(3I)**
ADataFormats() – get list of data formats supported by audio controller	**ADataFormats(3X)**
add argument and data to NetIPC option buffer	**addopt(3N)**
add a swap device for interleaved paging/swapping	**swapon(2)**
add audio event handler for this connection	**AtInitialize(3X)**

Index
Volume 2

Description	Entry Name(Section)
add callback procedure for audio toolkit	**AtAddCallback(3X)**
addexportent ( ) – access exported file system information	**exportent(3N)**
addmntent ( ) – add entry to open file system description file	**getmntent(3X)**
add, modify, or delete access control list entry	**setaclentry(3C)**
addopt ( ) – add argument and data to NetIPC option buffer	**addopt(3N)**
addresses – first locations beyond allocated program regions	**end(3C)**
address, get socket	**getsockname(2)**
address manipulation routines, Internet	**inet(3N)**
address of connected peer, get	**getpeername(2)**
address string conversion routines, network station	**net_aton(3C)**
address to a socket, bind	**bind(2)**
add value to environment	**putenv(3C)**
ADestroySBucket ( ) – destroy specified sound bucket	**ADestroySBucket(3X)**
ADVANCE ( ) – advance pointer to next 8- or 16-bit character	**nl_tools_16(3C)**
advance ( ) – regular expression substring comparison routines	**regexp(3X)**
advise system of process' expected paging behavior	**madvise(2)**
AEndConversion ( ) – finish stream data conversion	**AEndConversion(3X)**
AEventsQueued ( ) – get number of events in queue for server connection	**AEventsQueued(3X)**
AGetAFileAttributes ( ) – get file attributes of file	**AGetAFileAttributes(3X)**
AGetChannelGain – get transaction channel gain	**AGetChannelGain(3X)**
AGetDataFormats ( ) – get data formats for specified file format	**AGetDataFormats(3X)**
AGetErrorText ( ) – copy error description into specified buffer	**AGetErrorText(3X)**
AGetGain ( ) – get play volume or record gain of specified transaction	**AGetGain(3X)**
AGetSBucketData ( ) – copy data in sound bucket to buffer	**AGetSBucketData(3X)**
AGetSilenceValue ( ) – get a silence value	**AGetSilenceValue(3X)**
AGetSystemChannelGain ( ) – get system or monitor channel gain	**AGetSystemChannelGain(3X)**
AGetTransStatus ( ) – get status of specified transaction	**AGetTransStatus(3X)**
AGMGainRestricted ( ) – find out if audio controller restricts gain entries	**AGMGainRestricted(3X)**
AGrabServer ( ) – acquire exclusive use of audio server	**AGrabServer(3X)**
AInputChannels ( ) – get list of A/D input channels on current hardware	**AInputChannels(3X)**
AInputSources ( ) – get types of input sources existing on current hardware	**AInputSources(3X)**
alarm clock, set a process's	**alarm(2)**
alarm ( ) – set a process's alarm clock	**alarm(2)**
allocate a per-process timer	**mktimer(3C)**
allocate data and stack space then lock process into memory	**datalock(3C)**
allocated program regions, first locations beyond	**end(3C)**
allocation, change data segment space	**brk(2)**
allocator for main memory	**malloc(3C)**
allow interface to enable SRQ line on HP-IB	**hpib_rqst_srvce(3I)**
almanac ( ) – return numeric date information in MPE format	**almanac(3X)**
ALoadAFile ( ) – copy audio file into new sound bucket with data conversion	**ALoadAFile(3X)**
alphasort ( ) – sort a directory pointer array	**scandir(3C)**
AMaskEvent ( ) – get first matching event in audio event queue	**AMaskEvent(3X)**
AMaxInputGain ( ) – get maximum input gain supported by audio controller	**AMaxInputGain(3X)**
AMaxOutputGain ( ) – get maximum output gain supported by audio controller	**AMaxOutputGain(3X)**
AMinInputGain ( ) – get minimum input gain supported by audio controller	**AMinInputGain(3X)**
AMinOutputGain ( ) – get minimum output gain supported by audio controller	**AMinOutputGain(3X)**
ANextEvent ( ) – dequeue and return first event in audio event queue	**ANextEvent(3X)**
anonymous memory region, initialize semaphore in mapped file or	**msem_init(2)**
anonymous region, remove semaphore in mapped file or	**msem_remove(2)**
another process, request connection to	**ipcconnect(2)**
ANumDataFormats ( ) – get number of data formats supported by controller	**ANumDataFormats(3X)**
ANumSamplingRates ( ) – get number of sampling rates supported by controller	**ANumSamplingRates(3X)**
AOpenAudio ( ) – open connection to specified audio server	**AOpenAudio(3X)**
AOutputChannels ( ) – get D/A output channels existing on current hardware	**AOutputChannels(3X)**
AOutputDestinations ( ) – get types of output destinations on hardware	**AOutputDestinations(3X)**
APauseAudio ( ) – pause the specified audio transaction	**APauseAudio(3X)**
APeekEvent ( ) – return but do not dequeue first event in audio event queue	**APeekEvent(3X)**

Description	Entry Name(Section)
`APlaySBucket()` – play specified sound bucket and return transaction ID	**APlaySBucket(3X)**
`APlaySStream()` – initiate transaction and return transaction ID and SStream structure	**APlaySStream(3X)**
`AProtocolRevision()` – get revision number of protocol on audio server	**AProtocolRevision(3X)**
`AProtocolVersion()` – get version number of protocol on audio server	**AProtocolVersion(3X)**
`APutBackEvent()` – push event onto head of audio event queue	**APutBackEvent(3X)**
`APutSBucketData()` – copy audio data from buffer to sound bucket	**APutSBucketData(3X)**
`AQLength()` – return number of events on audio event queue	**AQLength(3X)**
`AQueryAFile()` – get file format of specified file	**AQueryAFile(3X)**
arcsine, arccosine, arctangent trigonometric functions	**trig(3M)**
arcsine, arccosine, arctangent trigonometric functions (degrees)	**trigd(3M)**
`ARecordAData()` – read audio data into sound bucket	**ARecordAData(3X)**
`ARecordSStream()` – initiate transaction	**ARecordSStream(3X)**
`AResumeAudio()` – resume specified audio transaction	**AResumeAudio(3X)**
argument and data to NetIPC option buffer, add	**addopt(3N)**
argument list, print formatted output of a varargs	**vprintf(3S)**
argument, varargs, formatted input conversion to a	**vscanf(3S)**
argument vector, get option letter from	**getopt(3C)**
arm a per-process timer, relatively	**reltimer(3C)**
array element, convert floating-point number to string or string	**ecvt(3C)**
array, sort a directory pointer	**scandir(3C)**
`ASamplingRates()` – return list of sampling rates supported by audio controller	**ASamplingRates(3X)**
`ASaveSBucket()` – write sound bucket data into file with data conversion	**ASaveSBucket(3X)**
ASCII, 7-bit, translate characters to	**conv(3C)**
ASCII string, convert between long integer and base-64	**a64l(3C)**
ASCII string, convert long integer to	**ltostr(3C)**
`asctime()`, `nl_ascxtime()` – convert **tm** structure date and time to string	**ctime(3C)**
`ASelectInput()` – request report of specified audio events	**ASelectInput(3X)**
`AServerVendor()` – get vendor name of audio server for this connection	**AServerVendor(3X)**
`ASetChannelGain()` – set transaction channel gain	**ASetChannelGain(3X)**
`ASetCloseDownMode()` – set close-down mode on connection	**ASetCloseDownMode(3X)**
`ASetErrorHandler()` – replace audio error handler	**ASetErrorHandler(3X)**
`ASetGain()` – set play volume or record gain of specified transaction	**ASetGain(3X)**
`ASetIOErrorHandler()` – replace audio I/O error handler	**ASetIOErrorHandler(3X)**
`ASetSystemChannelGain()` – set system or monitor channel gain	**ASetSystemChannelGain(3X)**
`ASetSystemPlayGain()` – set system play volume	**ASetSystemPlayGain(3X)**
`ASetSystemRecordGain()` – set system record gain	**ASetSystemRecordGain(3X)**
`ASetupConversion()` – perform setup required for stream data conversion	**ASetupConversion(3X)**
`ASimplePlayer()` – return gain matrix of basic play device	**ASimplePlayer(3X)**
`ASimpleRecorder()` – return gain matrix of basic recording device	**ASimpleRecorder(3X)**
`asindf()` – trigonometric arcsine function (float, degrees)	**trigd(3M)**
`asind()` – trigonometric arcsine function (degrees)	**trigd(3M)**
`asinf()` – trigonometric arcsine function (float)	**trig(3M)**
`asinh()` – inverse hyperbolic sine function	**asinh(3M)**
`asin()` – trigonometric arcsine function	**trig(3M)**
`ASoundBitOrder()` – get bit order used for one-bit-per-sample data	**ASoundBitOrder(3X)**
`ASoundByteOrder()` – get audio data byte order for this connection	**ASoundByteOrder(3X)**
assertion, verify program	**assert(3X)**
`assert()` – verify program assertion	**assert(3X)**
assign buffering to a stream file	**setbuf(3S)**
associate name with call socket or destination call socket	**ipcname(2)**
`AStopAudio()` – stop specified audio transaction	**AStopAudio(3X)**
`async_daemon`: NFS daemon	**nfssvc(2)**
asynchronous faults, enable	**pfm_$enable(3)**
asynchronous faults, enable	**pfm_$enable_faults(3)**
asynchronous faults, inhibit but allow time-sliced task switching	**pfm_$inhibit_faults(3)**
asynchronous faults, inhibit	**pfm_$inhibit(3)**
`AtAddCallback()` – add callback procedure for audio toolkit	**AtAddCallback(3X)**
`atan2df()` – trigonometric arctangent-and-quadrant function (float, degrees)	**trigd(3M)**

Index
Volume 2

Description	Entry Name(Section)
`atan2d()` – trigonometric arctangent-and-quadrant function (degrees)	**trigd(3M)**
`atan2f()` – trigonometric arctangent-and-quadrant function (float)	**trig(3M)**
`atan2()` – trigonometric arctangent-and-quadrant function	**trig(3M)**
`atandf()` – trigonometric arctangent function (float, degrees)	**trigd(3M)**
`atand()` – trigonometric arctangent function (degrees)	**trigd(3M)**
`atanf()` – trigonometric arctangent function (float)	**trig(3M)**
`atan()` – trigonometric arctangent function	**trig(3M)**
`atexit()` – register a function to be called at program termination	**atexit(2)**
`AtInitialize()` – add audio event handler for this connection	**AtInitialize(3X)**
ATN commands, enable/disable odd parity on	**hpib_parity_ctl(3I)**
`atof()` – convert string to double-precision number	**strtod(3C)**
`atoi()` – convert string to long integer	**strtol(3C)**
`atol()` – convert string to long integer	**strtol(3C)**
atomically release blocked signals and wait for interrupt	**sigpause(2)**
`AtRemoveCallback()` – set callback to NULL	**AtRemoveCallback(3X)**
attach shared memory to data segment	**shmop(2)**
Attention line on HP-IB, control	**hpib_atn_ctl(3I)**
attributes of specified file, get file	**AGetAFileAttributes(3X)**
attributes to use when creating a new file, select	**AChoosePlayAttributes(3X)**
`AuCreatePlay()` – create an audio play widget	**AuCreatePlay(3X)**
`AuCreateRecord()` – create an audio record widget	**AuCreateRecord(3X)**
`audctl()` – start or halt auditing system; set or get audit files	**audctl(2)**
audiochannel gain, get system or monitor	**AGetSystemChannelGain(3X)**
audio channel gain, set system or monitor	**ASetSystemChannelGain(3X)**
audio event handler for this connection, add	**AtInitialize(3X)**
audio file data format, convert	**AConvertAFile(3X)**
audio play widget	**AuPlayWidget(3X)**
audio play widget, create an	**AuCreatePlay(3X)**
audio record widget	**AuRecordWidget(3X)**
audio record widget, create an	**AuCreateRecord(3X)**
audio toolkit, add callback procedure for	**AtAddCallback(3X)**
audio widget play operation, initiate an	**AuInvokePlay(3X)**
audio widget record operation, initiate an	**AuInvokeRecord(3X)**
audit: get events and system calls currently being audited	**getevent(2)**
audit: set current events and system calls to be audited	**setevent(2)**
audit: set or clear auditing on calling process	**setaudproc(2)**
audit: set or get audit files	**audctl(2)**
audit: start or halt auditing system	**audctl(2)**
audit files, set or get	**audctl(2)**
audit ID (`aid()`) for current process, get	**getaudid(2)**
audit ID (`aid()`), set for current process	**setaudid(2)**
auditing, set or clear on calling process	**setaudproc(2)**
auditing, suspend or resume on current process	**audswitch(2)**
auditing system, start or halt	**audctl(2)**
audit process flag for calling process, get	**getaudproc(2)**
audit record, write for self-auditing process	**audwrite(2)**
`audswitch()` – suspend or resume auditing on current process	**audswitch(2)**
`audwrite()` – write audit record for self-auditing process	**audwrite(2)**
`AuInvokePlay()` – initiate an audio widget play operation	**AuInvokePlay(3X)**
`AuInvokeRecord()` – initiate an audio widget record operation	**AuInvokeRecord(3X)**
`AUngrabServer()` – release server from exclusive use by this connection	**AUngrabServer(3X)**
`AUpdateDataLength()` – update a file's header	**AUpdateDataLength(3X)**
`AuPlayWidget()` – audio play widget	**AuPlayWidget(3X)**
`AuRecordWidget()` – audio record widget	**AuRecordWidget(3X)**
`AuSaveFile()` – save sound bucket data created by record widget	**AuSaveFile(3X)**
`auth_destroy()` – destroy authentication information handle	**rpc(3C)**
`authnone_create()` – get RPC authentication handle with no checking	**rpc(3C)**
`authunix_create_default()` – get default UNIX authentication handle	**rpc(3C)**

Description	Entry Name(Section)
`authunix_create()` – get RPC authentication handle with UNIX permissions	**rpc(3C)**
`AVendorRelease()` – get vendor release number of audio server for this connection	**AVendorRelease(3X)**
`AWriteAHeader()` – write a header for an audio file	**AWriteAHeader(3X)**
back into input stream, push character	**ungetc(3S)**
back into input stream, push wide character	**ungetwc(3C)**
base-64 ASCII string, convert long integer to	**a64l(3C)**
baud rate, tty, set or get	**cfspeed(3C)**
`bcmp()` – BSD memory compare	**memory(3C)**
`bcopy()` – BSD memory copy	**memory(3C)**
behavior, advise system of process' expected paging	**madvise(2)**
Bessel functions	**bessel(3M)**
binary input/output to a stream file, buffered	**fread(3S)**
binary search routine for sorted tables	**bsearch(3C)**
binary search tree, manage a	**tsearch(3C)**
bind a socket to a privileged IP port	**bindresvport(3N)**
`bind()` – bind address to a socket	**bind(2)**
`bindresvport()` – bind a socket to a privileged IP port	**bindresvport(3N)**
`blclose()` – terminal block-mode library interface	**blmode(3C)**
`blget()` – terminal block-mode library interface	**blmode(3C)**
`blmode()` – terminal block-mode library interface	**blmode(3C)**
blocked signals, examine and change	**sigprocmask(2)**
blocked signals, release and atomically wait for interrupt	**sigpause(2)**
block-mode terminal I/O library interface	**blmode(3C)**
block signals	**sigblock(2)**
`blopen()` – terminal block-mode library interface	**blmode(3C)**
`blread()` – terminal block-mode library interface	**blmode(3C)**
`blset()` – terminal block-mode library interface	**blmode(3C)**
boot the system	**reboot(2)**
break value and file size limits, get or set	**ulimit(2)**
`brk()`, `sbrk()` – change data segment space allocation	**brk(2)**
BSD-4.2-compatible `kill()`, `sigvec()`, and `signal()` system calls	**bsdproc(2)**
`bsearch()` – binary search routine for sorted tables	**bsearch(3C)**
buffer, add argument and data to NetIPC option	**addopt(3N)**
buffered binary input/output to a stream file	**fread(3S)**
buffered input/output standard stream file package	**stdio(3S)**
buffer, flush with or without closing stream	**fclose(3S)**
buffering, assign to a stream file	**setbuf(3S)**
buffer, initialize NetIPC option	**initopt(3N)**
buffer, obtain option code and data from NetIPC option	**readopt(3N)**
buffers, flush to disk	**sync(2)**
buffers, use to perform I/O with an HP-IB channel	**hpib_io(3I)**
build or access a binary search tree	**tsearch(3C)**
bus address for an interface, set HP-IB	**hpib_address_ctl(3I)**
bus	see HP-IB
bus, stop activity on specified HP-IB	**hpib_abort(3I)**
byte order, network and host, convert values between	**byteorder(3N)**
bytes needed by a NetIPC option, return number of	**optoverhead(3N)**
bytes over HP-IB, send command	**hpib_send_cmnd(3I)**
bytes, swap	**swab(3C)**
`byte_status()`, `BYTE_STATUS()` – test for valid 1- or 2-byte character	**nl_tools_16(3C)**
`bzero()` – BSD memory clear	**memory(3C)**
`cabs()` – complex absolute value function	**hypot(3M)**
`cachectl()` – flush and/or purge the cache	**cachectl(3C)**
cache, flush and/or purge the	**cachectl(3C)**
`calendar()` – return MPE calendar date	**calendar(3X)**
callback procedure for audio toolkit, add	**AtAddCallback(3X)**
callback to NULL, set	**AtRemoveCallback(3X)**
calling process, get audit process flag for	**getaudproc(2)**

Index
Volume 2

Description	Entry Name(Section)
calling process, set or clear auditing on	**setaudproc(2)**
calling process, signal the	**pfm_$signal(3)**
`calloc()` – allocate memory for array – main memory allocator	**malloc(3C)**
`callrpc()` – call remote procedure	**rpc(3C)**
call socket or destination call socket, associate name with	**ipcname(2)**
call socket or destination call socket, delete name associated with a	**ipcnamerase(2)**
call socket or VC socket, determine status of	**ipcselect(2)**
call socket, receive connection request on a	**ipcrecvcn(2)**
calls, remote procedure, library routines for	**rpc(3C)**
calls, system, BSD-4.2-compatible `kill()`, `sigvec()`, and `signal()`	**bsdproc(2)**
cancel a per-process timer	**rmtimer(3C)**
C and Pascal execution startup routines	**crt0(3)**
capabilities, check for presence of hardware	**is_hw_present(3C)**
catalog for reading, close or open NLS message	**catopen(3C)**
catalog support, RTE/MPE-style message	**catread(3C)**
catalogue, get message from an NLS message	**catgetmsg(3C)**
`catclose()` – close NLS message catalog for reading	catopen(3C)
`catgetmsg()` – get message from an NLS message catalogue	**catgetmsg(3C)**
`catgets()` – get an NLS program message	**catgets(3C)**
`catopen()` – open NLS message catalog for reading	**catopen(3C)**
`catread()` – MPE/RTE-style message catalog support	**catread(3C)**
`cbrtf()`, `cbrt()`, `sqrt()`, `sqrtf()` – cube root, square root functions	**exp(3M)**
`cbrt()`, `sqrt()`, `sqrtf()`, `cbrtf()` – cube root, square root functions	**exp(3M)**
`c_colwidth()`, `c_COLWIDTH()` – test for valid first byte in 16-bit character	**nl_tools_16(3C)**
`ceil()`, `floor()`, `fmod()`, `fmodf()`, `fabs()`, `fabsf()`, `rint()` – ceiling, floor, remainder, absolute value, round-to-nearest functions	**floor(3M)**
ceiling, floor, remainder, absolute value, round-to-nearest functions	**floor(3M)**
`cfgetispeed()` – get tty input baud rate	**cfspeed(3C)**
`cfgetospeed()` – get tty output baud rate	**cfspeed(3C)**
`cfsetispeed()` – set tty input baud rate	**cfspeed(3C)**
`cfsetospeed()` – set tty output baud rate	**cfspeed(3C)**
change access mode (permissions) of file	**chmod(2)**
change active controllers on HP-IB	**hpib_pass_ctl(3I)**
change data segment space allocation	**brk(2)**
change or add value to environment	**putenv(3C)**
change or examine blocked signals	**sigprocmask(2)**
change or examine signal action	**sigaction(2)**
change or read real-time priority	**rtprio(2)**
change owner and group of a file	**chown(2)**
change owner and/or group in access control list (ACL)	**chownacl(3C)**
change priority of a process	**nice(2)**
change root directory	**chroot(2)**
change the name of a file	**rename(2)**
change working directory	**chdir(2)**
channel, create an interprocess	**pipe(2)**
channel from buffers, perform I/O with an HP-IB	**hpib_io(3I)**
channel gain, get system or monitor audio	**AGetSystemChannelGain(3X)**
channel gain, get transaction	**AGetChannelGain(3X)**
channel gain, set system or monitor audio	**ASetSystemChannelGain(3X)**
channel gain, set transaction	**ASetChannelGain(3X)**
channel, perform low-overhead I/O on an HP-IB/GPIO/parallel	**io_burst(3I)**
character back into input stream, push	**ungetc(3S)**
character code set, convert to another	**iconv(3C)**
character, compare memory contents with specified	**memory(3C)**
character device special file, control	**ioctl(2)**
character, find location of in memory	**memory(3C)**
character or data word from a stream file, get	**getc(3S)**
character or word, put on a stream	**putc(3S)**

Description	Entry Name(Section)
characters and strings conversions, multibyte	multibyte(3C)
characters, classify according to type	ctype(3C)
characters, classify according to type	wctype(3C)
characters, classify for use with NLS	nl_ctype(3C)
character, set contents of memory area to specified	memory(3C)
characters, tools to process 16-bit	nl_tools_16(3C)
characters, translate for use with NLS (obsolete – use conv(3C))	nl_conv(3C)
characters, translate to uppercase, lowercase, or 7-bit ASCII	conv(3C)
character-string login name of the user, get	cuserid(3S)
character string operations	string(3C)
character string or stream file, read from with formatted input conversion	scanf(3S)
CHARADV() – get character and advance pointer to next character	nl_tools_16(3C)
CHARAT() – get value of 8- or 16-bit character	nl_tools_16(3C)
chdir() – change working directory	chdir(2)
check for presence of hardware capabilities	is_hw_present(3C)
check the network, scatter data to	spray(3N)
child or traced process to stop or terminate, wait for	wait(2)
child process and process times, get	times(2)
chmod(), fchmod() – change access mode (permissions) of file	chmod(2)
chownacl() – change owner and/or group in access control list (ACL)	chownacl(3C)
chown(), fchown() – change owner and group of a file	chown(2)
chroot() – change root directory	chroot(2)
circuit connection, establish or receive data on NetIPC virtual	ipcrecv(2)
circuit connection, send data on a virtual	ipcsend(2)
classify characters according to type	ctype(3C)
classify characters according to type	wctype(3C)
classify characters for use with NLS	nl_ctype(3C)
cleanup handler, establish a	pfm_$cleanup(3)
cleanup handler, release a	pfm_$rls_cleanup(3)
cleanup handler, reset a	pfm_$reset_cleanup(3)
cleanup handlers, exiting	pfm_$signal(3)
cleanup handlers	pfm_$intro(3)
clearenv – clear the process environment	clearenv(3C)
clearerr() – clear I/O error on stream	ferror(3S)
clear or set auditing on calling process	setaudproc(2)
clear the process environment	clearenv(3C)
client interface, Network Information Service	ypclnt(3C)
clnt_broadcast() – broadcast remote procedure call everywhere	rpc(3C)
clnt_call() – call remote procedure associated with client handle	rpc(3C)
clnt_control() – change or retrieve information associated with client handle	rpc(3C)
clnt_create() – create RPC client using caller-specified transport	rpc(3C)
clnt_destroy() – destroy client's RPC handle	rpc(3C)
clnt_freeres() – free data allocated by RPC/XDR when decoding results	rpc(3C)
clnt_geterr() – copy error info from client handle to error structure	rpc(3C)
clnt_pcreateerror() – print reason why client handle creation failed	rpc(3C)
clnt_perrno() – print message corresponding to condition given	rpc(3C)
clnt_perror() – print message explaining why RPC call failed	rpc(3C)
clntraw_create() – create toy RPC client for simulation	rpc(3C)
clnt_spcreateerror() – get pointer to why client handle creation failed	rpc(3C)
clnt_sperrno() – get pointer to message corresponding to error value	rpc(3C)
clnt_sperror() – get pointer to why an RPC call failed	rpc(3C)
clnttcp_create() – create RPC client using TCP transport	rpc(3C)
clntudp_create() – create RPC client using UDP transport	rpc(3C)
clock date and time, get or set system	gettimeofday(2)
clock, get current value of system-wide	getclock(3C)
clock() – report CPU time used	clock(3C)
clock() – return the MPE clock value	clock(3X)
clock, set value of system-wide	setclock(3C)

Index
Volume 2

Description	Entry Name(Section)
clock value, MPE, return the	**clock(3X)**
close, access, or open a directory and associated *directory stream*	**directory(3C)**
close a stream	**fclose(3S)**
`close()` – close a file descriptor	**close(2)**
close connection to specific audio server	**ACloseAudio(3X)**
`closedir()` – close a currently open directory	**directory(3C)**
close legal user shells file	**getusershell(3C)**
`closelog()` – close system log file	**syslog(3C)**
close or open NLS message catalog for reading	**catopen(3C)**
close or open pipe I/O to or from a process	**popen(3S)**
cluster configuration file, get entry from	**getccent(3C)**
cluster nodes, get a list of active diskless	**cnodes(2)**
clusters, diskless	see diskless clusters
`cnodeid()` – get diskless cnode ID of local machine	**cnodeid(2)**
cnode ID of local machine, get diskless	**cnodeid(2)**
`cnodes()` – get a list of active nodes in cluster	**cnodes(2)**
code set conversion, character	**iconv(3C)**
collation, non-ASCII string	**nl_string(3C)**
command bytes over HP-IB, send	**hpib_send_cmnd(3I)**
command, remote, return a stream to	**rcmd(3N)**
command, return stream to a remote	**rexec(3N)**
command, shell, issue a	**system(3S)**
communication, create an endpoint for	**socket(2)**
communication package, standard interprocess	**stdipc(3C)**
compare contents of memory with character	**memory(3C)**
compare two non-ASCII strings	**nl_string(3C)**
compare two strings	**string(3C)**
compare two wide strings	**wcstring(3C)**
comparison routines for regular expressions	**regexp(3X)**
compile and match routines for regular expressions	**regexp(3X)**
compile a regular expression	**regcmp(3X)**
`compile()` – regular expression compile routine	**regexp(3X)**
compiling routines, regular expression	**regcomp(3C)**
complementary error function and error function	**erf(3M)**
completion status code, return an error message for a	**error_$c_text(3)**
complex absolute value function	**hypot(3M)**
concatenate two strings	**string(3C)**
concatenate two wide strings	**wcstring(3C)**
condition becomes true, wait until the requested status	**hpib_status_wait(3I)**
conditions, define for I/O device interrupt	**io_on_interrupt(3I)**
conduct a serial poll on HP-IB	**hpib_spoll(3I)**
conduct parallel poll on HP-IB	**hpib_ppoll(3I)**
configurable pathname variables, get	**pathconf(2)**
configurable system variables, get	**sysconf(2)**
configuration file, cluster, get entry from	**getccent(3C)**
configuration values, get string-valued	**confstr(3C)**
`confstr()` – get string-valued configuration values	**confstr(3C)**
connected peer, get address of	**getpeername(2)**
connected sockets, create a pair of	**socketpair(2)**
`connect()` – initiate connection on a socket	**connect(2)**
connection, add audio event handler for this	**AtInitialize(3X)**
connection, establish an out-bound terminal line	**dial(3C)**
connection, establish or receive data on NetIPC virtual circuit	**ipcrecv(2)**
connection on a socket, accept	**accept(2)**
connection on a socket, initiate	**connect(2)**
connection request on a call socket, receive	**ipcrecvcn(2)**
connection, send data on a virtual circuit	**ipcsend(2)**
connections on a socket, listen for	**listen(2)**

Description	Entry Name(Section)
connection to another process, request	ipcconnect(2)
connect socket to TCP socket address; return transaction ID	AConnectRecordStream(3X)
consumption limit, get or set system resource	getrlimit(2)
context-dependent file path names, manipulate	getcdf(3C)
context-dependent file search, return process context for	getcontext(2)
context, signal stack, set and/or get	sigstack(2)
control Attention line on HP-IB	hpib_atn_ctl(3I)
control character device special file	ioctl(2)
control DMA allocation for an interface	io_dma_ctl(3I)
control EOI mode for HP-IB file	hpib_eoi_ctl(3I)
control, file system	fsctl(2)
control functions, tty line	tccontrol(3C)
controllers on HP-IB, change active	hpib_pass_ctl(3I)
control lines on GPIO card, set	gpio_set_ctl(3I)
controlling terminal, generate file name of	ctermid(3S)
control operations, message	mstctl(2)
control operations, semaphore	semctl(2)
control operations, shared memory	shmctl(2)
control register defaults (floating-point), set	fpgetround(3M)
control register (floating-point), examine and set	fpgetround(3M)
control response to parallel poll on HP-IB	hpib_card_ppoll_resp(3I)
control routines for open-files	fcntl(2)
control system log	syslog(3C)
control system resource consumption limit	getrlimit(2)
control terminal device (Version 6 compatibility only)	stty(2)
control the HP-IB interface Remote Enable line	hpib_ren_ctl(3I)
control tty device	tcattribute(3C)
conventions, numeric formatting, of current locale, query	localeconv(3C)
conversion, formatted input, to a varargs argument	vscanf(3S)
conversion routines, network station address string	net_aton(3C)
conversions, multibyte characters and strings	multibyte(3C)
convert a buffer of data	AConvertBuffer(3X)
convert access control list (ACL) structure to string form	acltostr(3C)
convert audio file data format	AConvertAFile(3X)
convert between 3-byte integers and long integers	l3tol(3C)
convert between long integer and base-64 ASCII string	a64l(3C)
convert character code set to another	iconv(3C)
convert date and time to string	ctime(3C)
convert date and time to string	strftime(3C)
convert date and time to wide-character string	wcsftime(3C)
convert file to stream	fopen(3S)
convert floating-point number to string or string array element	ecvt(3C)
convert long double floating-point number to string	ldcvt(3C)
convert long integer to string	ltostr(3C)
convert string data order	strord(3C)
convert string form to access control list (ACL) structure	strtoacl(3C)
convert string to double-precision number	strtod(3C)
convert string to floating-point number	cvtnum(3C)
convert string to long double-precision number	strtold(3C)
convert user format date and time	getdate(3C)
convert values between host and network byte order	byteorder(3N)
convert wide character string to double-precision number	wcstod(3C)
coprocessor, math, check for presence of	is_hw_present(3C)
copy access control list (ACL) to another file	cpacl(3C)
copy audio data from buffer to sound bucket	APutSBucketData(3X)
copy audio data in sound bucket to buffer; return number of bytes	AGetSBucketData(3X)
copy audio file into new sound bucket with data conversion	ALoadAFile(3X)
copy error description into specified buffer	AGetErrorText(3X)

Index
Volume 2

Description	Entry Name(Section)
copy memory to another area	memory(3C)
`copysign()`, `copysignf()` – copysign functions	ieee(3M)
`copysignf()`, `copysign()` – copysign functions	ieee(3M)
copysign functions	ieee(3M)
`cosdf()` – trigonometric cosine function (float, degrees)	trigd(3M)
`cosd()` – trigonometric cosine function (degrees)	trigd(3M)
`cosf()` – trigonometric cosine function (float)	trig(3M)
`cosh()`, `coshf()` – hyperbolic cosine functions	sinh(3M)
`coshf()`, `cosh()` – hyperbolic cosine functions	sinh(3M)
`cosh()` – inverse hyperbolic cosine function	asinh(3M)
cosine trigonometric function (degrees)	trigd(3M)
cosine trigonometric function	trig(3M)
`cos()` – trigonometric cosine function	trig(3M)
`cpacl()` – copy access control list (ACL) to another file	cpacl(3C)
cpu, set name of host	sethostname(2)
CPU, set NetIPC node name of host	ipcsetnodename(2)
CPU time used, report	clock(3C)
`creat()` – create a new file or rewrite an existing one	creat(2)
create a call socket	ipccreate(2)
create a destination descriptor	ipcdest(2)
create a directory file	mkdir(2)
create a directory, or a special or ordinary file	mknod(2)
create a name for a temporary file	tmpnam(3S)
create an audio play widget	AuCreatePlay(3X)
create an audio record widget	AuCreateRecord(3X)
create an endpoint for communication	socket(2)
create a new file	creat(2)
create a new file or rewrite an existing one	creat(2)
create a new process	fork(2)
create an interprocess channel	pipe(2)
create a pair of connected sockets	socketpair(2)
create a socket	socket(2)
create a temporary file	tmpfile(3S)
create a unique (usually temporary) file name	mktemp(3C)
created by record widget, save sound bucket data	AuSaveFile(3X)
create empty sound bucket and return pointer to it	ACreateSBucket(3X)
create file names	glob(3C)
create session and set process group ID	setsid(2)
creating a new file, select play attributes to use when	AChoosePlayAttributes(3X)
`crt0.o`, `gcrt0.o`, `mcrt0.o`, `frt0.o`, `mfrt0.o` – execution startup routines	crt0(3)
`crt0.o`, `mcrt0.o` – C and Pascal execution startup routines	crt0(3)
CRT optimization and screen handling package	curses(3X)
CRT screen handling and optimization package	curses(3X)
`crypt()`, `setkey()`, `encrypt()` – generate hashing encryption	crypt(3C)
`ctermid()` – generate file name for terminal	ctermid(3S)
`ctime()`, `nl_cxtime()` – convert `clock()` date and time to string	ctime(3C)
cube root, square root, power, logarithm, exponential functions	exp(3M)
current events and system calls to be audited	setevent(2)
current host, get name of	gethostname(2)
current HP-UX system, get name and version of	uname(2)
current locale, query numeric formatting conventions of	localeconv(3C)
current process, get audit ID (`aid()`) for	getaudid(2)
current process, set audit ID (`aid()`) for	setaudid(2)
current process, suspend or resume auditing on	audswitch(2)
current user, find the slot in the `utmp()` file of the	ttyslot(3C)
current value of system-wide clock, get	getclock(3C)
current working directory, get path-name of	getcwd(3C)
`currlangid()` – get current NLS language ID number	langinfo(3C)

Description	Entry Name(Section)
`curses()` – CRT screen handling and optimization package	**curses(3X)**
cursor control, CRT optimization, and screen handling package	**curses(3X)**
`cuserid()` – get character-string login name of the user	**cuserid(3S)**
`cvtnum()` – convert string to floating-point number	**cvtnum(3C)**
daemons, NFS	**nfssvc(2)**
data and stack space, allocate then lock process into memory	**datalock(3C)**
database operations, error text	**error_$intro(3)**
database subroutines (new multiple database version)	**ndbm(3X)**
database subroutines (old version – see also **ndbm(3X)**)	**dbm(3X)**
data created by record widget, save sound bucket	**AuSaveFile(3X)**
data format, convert audio file	**AConvertAFile(3X)**
data from a file, read	**read(2)**
data from NetIPC option buffer, obtain option code and	**readopt(3N)**
data, get character or word from a stream file	**getc(3S)**
data, get wide character from a stream file	**getwc(3C)**
`datalock()` – lock process into memory after allocating data and stack space	**datalock(3C)**
data order, convert string	**strord(3C)**
data path width (in bits), set	**io_width_ctl(3I)**
data pointer for binary search tree, get	**tsearch(3C)**
data representation, library routines for external	**xdr(3C)**
data segment and shared memory, attach or detach	**shmop(2)**
data segment space allocation, change	**brk(2)**
data, send on a virtual circuit connection	**ipcsend(2)**
data, text, or process, lock in memory	**plock(2)**
data to a file, write	**write(2)**
data to check the network, scatter	**spray(3N)**
data to NetIPC option buffer, add argument and	**addopt(3N)**
data transfer rate, inform system of required minimum I/O	**io_speed_ctl(3I)**
date and time, convert to string	**ctime(3C)**
date and time, convert to string	**strftime(3C)**
date and time, convert to wide-character string	**wcsftime(3C)**
date and time, convert user format	**getdate(3C)**
date and time, get more precisely (Version 7 compatibility only)	**ftime(2)**
date and time, get or set system clock	**gettimeofday(2)**
date and time, set	**stime(2)**
`daylight()` – Daylight Savings Time flag	**ctime(3C)**
`dbm_clearerr()` – reset error condition on named database	**ndbm(3X)**
`dbm_close()` – close an open database	**ndbm(3X)**
`dbmclose()` – close currently open database (old single-data-base version)	**dbm(3X)**
`dbm_delete()` – delete a database key and associated contents	**ndbm(3X)**
`dbm_error()` – error in reading or writing in a database	**ndbm(3X)**
`dbm_fetch()` – access a database entry under a key	**ndbm(3X)**
`dbm_firstkey()` – get first key in a database	**ndbm(3X)**
`dbminit()` – open a single database (old single-data-base version)	**dbm(3X)**
`dbm_nextkey()` – get next key in a database	**ndbm(3X)**
`dbm_open()` – open a database for access	**ndbm(3X)**
`dbm_store()` – store an entry under a key in a database	**ndbm(3X)**
decimal ASCII string, convert long integer to	**ltostr(3C)**
decimal library, packed, HP3000-mode	**hppac(3X)**
define additional signal stack space	**sigspace(2)**
define interface parallel poll response	**hpib_ppoll_resp_ctl(3I)**
define I/O device interrupt (fault) conditions	**io_on_interrupt(3I)**
define what to do upon receipt of a signal	**signal(2)**
degree-valued trigonometric functions	**trigd(3M)**
delete, add, or modify delete access control list entry	**setaclentry(3C)**
delete allocated signal stack space	**sigspace(2)**
delete a node from a binary search tree	**tsearch(3C)**
`delete()` – delete key and data under it (old single-data-base version)	**dbm(3X)**

Index
Volume 2

Description	Entry Name(Section)
delete file or directory name; remove directory entry	**unlink(2)**
delete name associated with a call socket or destination call socket	**ipcnamerase(2)**
dequeue and return first event in audio event queue	**ANextEvent(3X)**
descend a directory hierarchy recursively	**ftw(3C)**
description of disk by its name, get	**getdiskbyname(3C)**
descriptor, close a file	**close(2)**
descriptor, create a destination	**ipcdest(2)**
descriptor file entry, get file system (BSD 4.2 compatibility only)	**getfsent(3X)**
descriptor, map stream pointer to file	**fileno(3S)**
descriptor, obtain a destination	**ipclookup(2)**
descriptor, release a	**ipcshutdown(2)**
destination call socket, associate name with call socket or	**ipcname(2)**
destination call socket, delete name associated with a call socket or	**ipcnamerase(2)**
destination descriptor, create a	**ipcdest(2)**
destination descriptor, obtain a	**ipclookup(2)**
destroy specified sound bucket	**ADestroySBucket(3X)**
detach shared memory from data segment	**shmop(2)**
determine accessibility of a file	**access(2)**
determine current signal stack space	**sigspace(2)**
determine how last I/O read terminated	**io_get_term_reason(3I)**
determine status of call socket or VC socket	**ipcselect(2)**
device file, FIFO, make a	**mkfifo(3C)**
device for interleaved paging/swapping, add a swap	**swapon(2)**
device ID to file path, map	**devnm(3)**
device I/O interrupt (fault) control	**io_on_interrupt(3I)**
device special file, control character	**ioctl(2)**
devnm() – map device ID to file path	**devnm(3)**
dial(), **undial()** – establish an out-bound terminal line connection	**dial(3C)**
difftime() – difference between two calendar time values	**ctime(3C)**
directory: access, open, or close a directory and associated *directory stream*	**directory(3C)**
directory: change root directory	**chroot(2)**
directory: change working directory	**chdir(2)**
directory: delete file or directory name; remove directory entry	**unlink(2)**
directory: get entries in a filesystem-independent format	**getdirentries(2)**
directory: get path-name of current working directory	**getcwd(3C)**
directory: make a directory file	**mkdir(2)**
directory: make a directory, or a special or ordinary file	**mknod(2)**
directory: remove a directory file	**rmdir(2)**
directory: scan a directory	**scandir(3C)**
directory entry, remove; delete file or directory name	**unlink(2)**
directory file, remove a	**rmdir(2)**
directory hierarchy, recursively descend a	**ftw(3C)**
directory pointer array, sort a	**scandir(3C)**
directory, scan a	**scandir(3C)**
directory stream, directory and associated, open for access	**directory(3C)**
disable/enable odd parity on ATN commands	**hpib_parity_ctl(3I)**
disable or enable I/O interrupts for the associated **eid()**	**io_interrupt_ctl(3I)**
disable or enable process accounting	**acct(2)**
disk description by its name, get	**getdiskbyname(3C)**
disk, flush buffers to	**sync(2)**
diskless cluster nodes, get a list of active	**cnodes(2)**
diskless cnode ID of local machine, get	**cnodeid(2)**
disk quotas, manipulate	**quotactl(2)**
disk storage, preallocate fast	**prealloc(2)**
disk, synchronize a file's in-core state with its state on	**fsync(2)**
distance function, Euclidean (hypotenuse)	**hypot(3M)**
division and remainder, integer	**div(3C)**
div(), **ldiv()** – integer division and remainder	**div(3C)**

Description	Entry Name(Section)
DMA allocation for an interface, control	io_dma_ctl(3I)
`dn_comp()` – resolver routines	resolver(3N)
`dn_expand()` – resolver routines	resolver(3N)
domain, get or set name of current NIS	getdomainname(2)
double-precision number, convert string to	strtod(3C)
double-precision number, convert string to long	strtold(3C)
double-precision number, convert wide character string to	wcstod(3C)
`drand48()`, `erand48()` – generate double-precision pseudo-random numbers	drand48(3C)
`drem()` – remainder manipulations	ieee(3M)
`dup2()` – duplicate an open file descriptor to a specific slot	dup2(2)
`dup()` – duplicate an open file descriptor	dup(2)
duplicate an open file descriptor	dup(2)
duplicate an open file descriptor to a specific slot	dup2(2)
duplicate entries in a table, eliminate	lsearch(3C)
dynamic file system swapping	swapon(2)
echo, suppress while reading password from terminal	getpass(3C)
`ecvt()`, `fcvt()` – convert floating-point number to string	ecvt(3C)
`edata` – first address beyond initialized program data region	end(3C)
effective access rights to a file, get a user's	getaccess(2)
effective or real user or group ID, get	getuid(2)
effective, real, and/or saved user or group IDs, set	setresuid(2)
element, convert floating-point number to string or string array	ecvt(3C)
eliminate duplicate entries in a table	lsearch(3C)
emulate `/etc/termcap` access routines	termcap(3X)
enable asynchronous faults	pfm_$enable(3)
enable asynchronous faults	pfm_$enable_faults(3)
enable/disable odd parity on ATN commands	hpib_parity_ctl(3I)
enable or disable I/O interrupts for the associated `eid()`	io_interrupt_ctl(3I)
enable or disable process accounting	acct(2)
enable SRQ line on HP-IB, allow interface to	hpib_rqst_srvce(3I)
`encrypt()` – generate hashing encryption	crypt(3C)
encryption, hashing, generate	crypt(3C)
encryption, password	crypt(3C)
`endccent()` – close cluster configuration file	getccent(3C)
`endexportent()` – access exported file system information	exportent(3N)
`end` – first address beyond uninitialized program data region	end(3C)
`endfsent()` – close file system descriptor file	getfsent(3X)
`endgrent()` – close currently open `group()` file	getgrent(3C)
`endhostent()` – get network host entry	gethostent(3N)
end locations of allocated regions in program	end(3C)
`endmntent()` – close file system description file	getmntent(3X)
`endnetent()`: get network entry	getnetent(3N)
`endnetgrent()` – get network group entry	getnetgrent(3C)
endpoint for communication, create an	socket(2)
`endprotoent()` – get protocol entry	getprotoent(3N)
`endpwent()` – close currently open password file	getpwent(3C)
`endservent()`: get service entry	getservent(3N)
`endspwent()` – close currently open secure password file	getspwent(3C)
`endusershell()` – close legal user shells file	getusershell(3C)
`endutent()` – close currently open `utmp()` file	getut(3C)
entries from a directory, get in a filesystem-independent format	getdirentries(2)
entries from name list, get	nlist(3C)
entries in a table, eliminate duplicate	lsearch(3C)
entry from cluster configuration file, get	getccent(3C)
entry from `group()` file, get	getgrent(3C)
entry from password file, get	getpwent(3C)
entry from secure password file, get	getspwent(3C)
entry, get file system description file	getmntent(3X)

Index
Volume 2

Description	Entry Name(Section)
entry, get file system descriptor file (BSD 4.2 compatibility only)	**getfsent(3X)**
entry, get or set protocol	**getprotoent(3N)**
entry, get RPC	**getrpcent(3C)**
entry, network group, get or set	**getnetgrent(3C)**
entry, service, get or set	**getservent(3N)**
entry, write password file	**putpwent(3C)**
environment, change or add value to	**putenv(3C)**
environment, clear the process	**clearenv(3C)**
environment list, search for value of specified variable name	**getenv(3C)**
environment of a program, initialize the NLS	**nl_init(3C)**
environment, save/restore stack for non-local goto	**setjmp(3C)**
environment variable, search environment list for value of	**getenv(3C)**
EOI mode for HP-IB file, control	**hpib_eoi_ctl(3I)**
erf(), **erfc()** – error function and complementary error function	**erf(3M)**
errno() – error indicator for system calls	**errno(2)**
errno – system error messages	**perror(3C)**
error_$c_get_text() – return subsystem, module, and error texts for a status code	**error_$c_get_text(3)**
error_$c_text() – return an error message for a status code	**error_$c_text(3)**
error function and complementary error function	**erf(3M)**
error-handling function, math library	**matherr(3M)**
error indicator for system calls	**errno(2)**
error_$intro – error text database operations	**error_$intro(3)**
error message for a status code, return an	**error_$c_text(3)**
error messages, system	**perror(3C)**
error number, provide text describing NetIPC	**ipcerrmsg(3N)**
error text database operations	**error_$intro(3)**
error texts for a status code, return subsystem, module, and	**error_$c_get_text(3)**
establish a cleanup handler	**pfm_$cleanup(3)**
establish an out-bound terminal line connection	**dial(3C)**
establish NetIPC virtual circuit connection	**ipcrecv(2)**
establish time limit for I/O operations	**io_timeout_ctl(3I)**
/etc/termcap access routines, emulate	**termcap(3X)**
etext – first address beyond program text region	**end(3C)**
Euclidean distance (hypotenuse) function	**hypot(3M)**
event handler for this connection, add audio	**AtInitialize(3X)**
events and system calls currently being audited, get	**getevent(2)**
events and system calls to be audited	**setevent(2)**
examine and change blocked signals	**sigprocmask(2)**
examine and change signal action	**sigaction(2)**
examine pending signals	**sigpending(2)**
exception flags (floating-point), examine and set	**fpgetround(3M)**
exceptions, managing signal	**pfm_$intro(3)**
exception trap enable bits (floating-point), examine and set	**fpgetround(3M)**
execl(), **execle()**, **execlp()**, **execv()**, **execve()**, **execvp()** – execute an object-code file	**exec(2)**
execute an object-code file	**exec(2)**
execute a regular expression against a string	**regcmp(3X)**
execution profile, prepare	**monitor(3C)**
execution startup routines, C, Pascal, and FORTRAN	**crt0(3)**
execution, suspend for interval	**sleep(3C)**
execution time profile	**profil(2)**
existing file, truncate to zero for rewriting	**creat(2)**
exit a program	**pgm_$exit**
exit(), **_exit()** – terminate process	**exit(2)**
exiting cleanup handlers	**pfm_$signal(3)**
exit, register a function to be called at	**atexit(2)**
expansions, perform word	**wordexp(3C)**
exp(), **expf()** – exponential functions	**exp(3M)**
expf(), **exp()** – exponential functions	**exp(3M)**

Description	Entry Name(Section)
explicit load of shared libraries	**shl_load(3X)**
exponent and mantissa, split floating-point into	**frexp(3C)**
exponential, logarithm, power, square root, cube root functions	**exp(3M)**
exponent manipulations	**ieee(3M)**
exported file system information, access	**exportent(3N)**
exportent() – access exported file system information	**exportent(3N)**
expression matching routines, regular	**regcomp(3C)**
expression, regular, compile and match routines	**regexp(3X)**
expression, regular, compile or execute against a string	**regcmp(3X)**
external data representation, library routines for	**xdr(3C)**
fabs(), **fabsf()**, **floor()**, **ceil()**, **fmod()**, **fmodf()**, **rint()** – absolute value, floor, ceiling, remainder, round-to-nearest functions	**floor(3M)**
fabsf(), **fabs()**, **floor()**, **ceil()**, **fmod()**, **fmodf()**, **rint()** – absolute value, floor, ceiling, remainder, round-to-nearest functions	**floor(3M)**
facilities, software signal	**sigvector(2)**
fast disk storage, preallocate	**prealloc(2)**
fault, generate an IOT	**abort(3C)**
fault (interrupt) conditions, define for I/O device	**io_on_interrupt(3I)**
fault management	**pfm_$intro**
faults, enable asynchronous	**pfm_$enable(3)**
faults, enable asynchronous	**pfm_$enable_faults(3)**
faults, inhibit asynchronous but allow time-sliced task switching	**pfm_$inhibit_faults(3)**
faults, inhibit asynchronous	**pfm_$inhibit(3)**
fchdir() – change working directory	**chdir(2)**
fchmod() – change access mode (permissions) of file	**chmod(2)**
fchown() – change owner and group of a file	**chown(2)**
fclose() – flush buffer then close stream	**fclose(3S)**
fcntl() – open-file control	**fcntl(2)**
fcpacl() – copy access control list (ACL) to another file	**cpacl(3C)**
fcvt(), **ecvt()** – convert floating-point number to string	**ecvt(3C)**
fdopen() – associate a stream with an open file descriptor	**fopen(3S)**
feof() – check for end-of-file error on stream	**ferror(3S)**
ferror() – check for I/O error on stream	**ferror(3S)**
fetch() – access data under a key (old single-data-base version)	**dbm(3X)**
fflush() – flush buffer without closing stream	**fclose(3S)**
ffs() – BSD find first set bit	**memory(3C)**
fgetacl() – get access control list (ACL) information	**getacl(2)**
fgetccent() – get pointer to cluster configuration entry in a stream	**getccent(3C)**
fgetc(), **getc()** – get character from a stream file	**getc(3S)**
fgetgrent() – get next entry in **group()**-file-formatted input stream	**getgrent(3C)**
fgetpos() – save file position indicator for a stream	**fgetpos(3S)**
fgetpwent() – get next entry in password-file-formatted input stream	**getpwent(3C)**
fgets(), **gets()** – get a string from a *standard input* stream	**gets(3S)**
fgetspwent() – get next entry in secure password-file-formatted input stream	**getspwent(3C)**
fgetwc(), **getwc()** – get wide character from a stream file	**getwc(3C)**
fgetws(), **getws()** – get a wide string from a *standard input* stream	**getws(3C)**
FIFO special file, make a	**mkfifo(3C)**
file: access **wtmp()** or **utmp()** file	**getut(3C)**
file: assign buffering to a stream file	**setbuf(3S)**
file: change access mode (permissions) of file	**chmod(2)**
file: change owner and group of a file	**chown(2)**
file: change the name of a file	**rename(2)**
file: close a file descriptor	**close(2)**
file: copy access control list (ACL) to another file	**cpacl(3C)**
file: create a name for a temporary file	**tmpnam(3S)**
file: create a new file or rewrite an existing one	**creat(2)**
file: create a temporary file	**tmpfile(3S)**
file: delete file or directory name; remove directory entry	**unlink(2)**

Index
Volume 2

Description	Entry Name(Section)
file: determine accessibility of a file	**access(2)**
file: execute an object-code file	**exec(2)**
file: get file status	**stat(2)**
file: link additional name to an existing file	**link(2)**
file: make a directory file or a special or ordinary file	**mknod(2)**
file: make a symbolic link to a file	**symlink(2)**
file: make a unique (usually temporary) file name	**mktemp(3C)**
file: open a file for reading or writing	**open(2)**
file: open-file control routines	**fcntl(2)**
file: print formatted output with numbered arguments to a file or string	**printmsg(3C)**
file: read data from a file	**read(2)**
file: read from file, stream, or character string with formatted input conversion	**scanf(3S)**
file: remove a directory file	**rmdir(2)**
file: remove a file	**remove(3C)**
file: rewrite an existing file	**creat(2)**
file: truncate a file to a specified length	**truncate(2)**
file: truncate an existing file to zero for rewriting	**creat(2)**
file: write data to a file	**write(2)**
file access and modification times, set or update	**utime(2)**
file attributes of specified file, get	**AGetAFileAttributes(3X)**
file, CDF: return process context for context-dependent file search, return	**getcontext(2)**
file, cluster configuration: get entry from cluster configuration file	**getccent(3C)**
file creation (permissions) mask, set and get	**umask(2)**
file data format, convert audio	**AConvertAFile(3X)**
file descriptor: duplicate an open file descriptor	**dup(2)**
file descriptor: duplicate an open file descriptor to a specific slot	**dup2(2)**
file descriptor, map stream pointer to	**fileno(3S)**
file entry, get file system description	**getmntent(3X)**
file entry, get file system descriptor (BSD 4.2 compatibility only)	**getfsent(3X)**
file, get a user's effective access rights to a	**getaccess(2)**
file, get file attributes of specified	**AGetAFileAttributes(3X)**
file, group: get entry from `group()` file	**getgrent(3C)**
file handle for file on remote node, get	**getfh(2)**
file locking: provide semaphores and record locking on files	**lockf(2)**
file name generation function	**glob(3C)**
file name of controlling terminal, generate	**ctermid(3S)**
filename patterns, match	**fnmatch(3C)**
`fileno()` – get integer file descriptor for stream	**ferror(3S)**
`fileno()` – map stream pointer to file descriptor	**fileno(3S)**
file on remote node, get file handle for	**getfh(2)**
file or anonymous memory region, initialize semaphore in mapped	**msem_init(2)**
file or anonymous region, remove semaphore in mapped	**msem_remove(2)**
file, password: get entry from password file	**getpwent(3C)**
file, password: get entry from secure password file	**getspwent(3C)**
file path, map device ID to	**devnm(3)**
file path names, manipulate context-dependent	**getcdf(3C)**
file pointer: move read/write file pointer	**lseek(2)**
file position indicator for a stream, save or restore	**fgetpos(3S)**
files, audit, set or get	**audctl(2)**
file search: return process context for context-dependent file search	**getcontext(2)**
file's in-core state with its state on disk, synchronize a	**fsync(2)**
file size limits and break value, get or set	**ulimit(2)**
file, stream: buffered binary input/output to a stream file	**fread(3S)**
file, stream: convert file to stream; open or re-open a stream file	**fopen(3S)**
file, stream: get character or data word from a stream file	**getc(3S)**
file, stream: get wide character from a stream file	**getwc(3C)**
file, stream: open or re-open a stream file; convert file to stream	**fopen(3S)**
file, stream: reposition or get pointer for I/O operations on a stream file	**fseek(3S)**

Description	Entry Name(Section)
file, synchronize a mapped	**msync(2)**
file system: get file system description file entry	**getmntent(3X)**
file system: get file system descriptor file entry (BSD 4.2 compatibility only)	**getfsent(3X)**
file system: get file system statistics	**statfs(2)**
file system: get mounted file system statistics	**ustat(2)**
file system: mount a file system	**vfsmount(2)**
file system: mount a removable file system	**mount(2)**
file system: unmount a file system	**umount(2)**
file system control	**fsctl(2)**
file system information, access exported	**exportent(3N)**
file systems, keep track of remotely mounted	**mount(3N)**
file system statistics, get	**statfsdev(3C)**
file system swapping	**swapon(2)**
file tree: walk a file tree	**ftw(3C)**
file, `utmp()`, of the current user, find the slot in the	**ttyslot(3C)**
find name of a terminal	**ttyname(3C)**
find out if audio controller restricts gain entries	**AGMGainRestricted(3X)**
find the slot in the `utmp()` file of the current user	**ttyslot(3C)**
finish stream data conversion	**AEndConversion(3X)**
`finitef()`, `finite()` – floating-point classification functions	**ieee(3M)**
`finite()`, `finitef()` – floating-point classification functions	**ieee(3M)**
`firstkey()` – get first key in database (old single-data-base version)	**dbm(3X)**
first locations beyond allocated program regions	**end(3C)**
`firstof2()`, `FIRSTof2()` – test for valid first byte in 16-bit character	**nl_tools_16(3C)**
flag for calling process, get audit process	**getaudproc(2)**
floating-point: convert floating-point number to string or string array element	**ecvt(3C)**
floating-point: convert string to floating-point number	**cvtnum(3C)**
floating-point: split floating-point into mantissa and exponent	**frexp(3C)**
floating-point classification functions	**fpclassify(3M)**
floating-point classification functions	**ieee(3M)**
floating-point classification functions	**isinf(3M)**
floating-point classification functions	**isnan(3M)**
floating-point mode control functions	**fpgetround(3M)**
floating-point number to string, convert long double	**ldcvt(3C)**
`floor()`, `ceil()`, `fmod()`, `fmodf()`, `fabs()`, `fabsf()`, `rint()` – floor, ceiling, remainder, absolute value, round-to-nearest functions	**floor(3M)**
floor, ceiling, remainder, absolute value, round-to-nearest functions	**floor(3M)**
flush and/or purge the cache	**cachectl(3C)**
flush buffers to disk	**sync(2)**
flush buffer with or without closing stream	**fclose(3S)**
`fmodf()`, `fmod()`, `ceil()`, `floor()`, `fabs()`, `fabsf()`, `rint()` – remainder, ceiling, floor, absolute value, round-to-nearest functions	**floor(3M)**
`fmod()`, `fmodf()`, `ceil()`, `floor()`, `fabs()`, `fabsf()`, `rint()` – remainder, ceiling, floor, absolute value, round-to-nearest functions	**floor(3M)**
`fnmatch()` – match filename patterns	**fnmatch(3C)**
`fopen()` – open a named file and associate with a stream	**fopen(3S)**
foreground process group ID, get	**tcgetpgrp(3C)**
foreground process group ID, set	**tcsetpgrp(3C)**
`fork()` – create a new process	**fork(2)**
format, convert audio file data	**AConvertAFile(3X)**
format date and time, convert user	**getdate(3C)**
formatted input conversion, read from stream file or character string	**scanf(3S)**
formatted input conversion to a varargs argument	**vscanf(3S)**
formatted output of a varargs argument list, print	**vprintf(3S)**
formatted output, print to *standard output*, file, or string	**printf(3S)**
formatted output with numbered arguments, print to a file or string	**printmsg(3C)**
formatted read and conversion from stream file or character string	**scanf(3S)**
formatting conventions, numeric, of current locale, query	**localeconv(3C)**

Index
Volume 2

Description	Entry Name(Section)
FORTRAN execution startup routines	crt0(3)
`fpathconf()` – get configurable pathname variables	pathconf(2)
`fpclassifyf()`, `fpclassify()` – floating-point classification functions	fpclassify(3M)
`fpclassify()`, `fpclassifyf()` – floating-point classification functions	fpclassify(3M)
`fpgetcontrol()`, `fpsetcontrol()` – examine and set floating-point control register	fpgetround(3M)
`fpgetfastmode()`, `fpsetfastmode()` – examine and set floating-point underflow mode	fpgetround(3M)
`fpgetmask()`, `fpsetmask()` – examine and set floating-point exception trap enables	fpgetround(3M)
`fpgetround()`, `fpsetround()` – examine and set floating-point rounding mode	fpgetround(3M)
`fpgetsticky()`, `fpsetsticky()` – examine and set floating-point exception flags	fpgetround(3M)
`fprintf()`, `nl_fprintf()` – print formatted output to a file	printf(3S)
`fprintmsg()` – print formatted output with numbered arguments to a file	printmsg(3C)
`fpsetcontrol()`, `fpgetcontrol()` – examine and set floating-point control register	fpgetround(3M)
`fpsetdefaults()` – set floating-point control register defaults	fpgetround(3M)
`fpsetfastmode()`, `fpgetfastmode()` – examine and set floating-point underflow mode	fpgetround(3M)
`fpsetmask()`, `fpgetmask()` – examine and set floating-point exception trap enables	fpgetround(3M)
`fpsetround()`, `fpgetround()` – examine and set floating-point rounding mode	fpgetround(3M)
`fpsetsticky()`, `fpgetsticky()` – examine and set floating-point exception flags	fpgetround(3M)
`fputc()`, `putc()` – put character on a stream	putc(3S)
`fputs()` – write null-terminated string to a named stream file	puts(3S)
`fputwc()`, `putwc()` – put wide character on a stream	putwc(3C)
`fputws()` – write null-terminated wide string to a named stream file	fputws(3C)
`fread()`, `fwrite()` – buffered binary input/output to a stream file	fread(3S)
free a per-process timer	rmtimer(3C)
`free()` – release allocated block of main memory	malloc(3C)
`freopen()` – substitute a named file in place of an already open stream	fopen(3S)
`frexp()`, `ldexp()`, `modf()` – split floating-point into mantissa and exponent	frexp(3C)
`frt0.o`, `mfrt0.o` – FORTRAN execution startup routines	crt0(3)
`fscanf()`, `nl_fscanf()` – formatted read from named input stream file	scanf(3S)
`fsctl()` – file system control	fsctl(2)
`fseek()`, `rewind()`, `ftell()` – reposition a file pointer in a stream	fseek(3S)
`fseek()` – set position of next I/O operation on stream file	fseek(3S)
`fsetaclentry()` – add, modify, or delete access control list entry	setaclentry(3C)
`fsetacl()` – set access control list (ACL) information	setacl(2)
`fsetpos()` – restore file position indicator for a stream	fgetpos(3S)
`fstatfsdev()`, `statfsdev()` – get file system statistics	statfsdev(3C)
`fstatfs()`, `statfs()` – get file system statistics	statfs(2)
`fstat()`, (`stat()`, `lstat()`) – get open file status	stat(2)
`fsync()` – synchronize a file's in-core state with its state on disk	fsync(2)
`ftell()` – get offset from beginning-of-file of current byte in stream file	fseek(3S)
`ftime()` – get date and time more precisely (Version 7 compatibility only)	ftime(2)
`ftok()` – standard interprocess communication package	stdipc(3C)
`ftruncate()`, `truncate()` – truncate a file to a specified length	truncate(2)
`ftw()`, `ftwh()` `nftw()` – walk a file tree	ftw(3C)
function: Bessel functions	bessel(3M)
function: complex absolute value	hypot(3M)
function: Euclidean distance (hypotenuse)	hypot(3M)
function: hyperbolic trigonometric functions	sinh(3M)
function: inverse hyperbolic trigonometric functions	asinh(3M)
function: log gamma	gamma(3M)
function: trigonometric functions (degrees)	trigd(3M)
function: trigonometric functions	trig(3M)
function to be called at program termination, register a	atexit(2)
`fwrite()`, `fread()` – buffered binary input/output to a stream file	fread(3S)
gain, get system or monitor audio channel	AGetSystemChannelGain(3X)
gain, get transaction channel	AGetChannelGain(3X)
gain, set system or monitor audio channel	ASetSystemChannelGain(3X)
gain, set transaction channel	ASetChannelGain(3X)
gamma function, log	gamma(3M)

Description	Entry Name(Section)
`gamma()`, `lgamma()`, `signgam()` – log gamma function	**gamma(3M)**
`gcrt0.o`, `gfrt0.o` – C and Pascal execution startup routines	**crt0(3)**
`gcvt()`, `nl_gcvt()` – convert floating-point number to string array element	**ecvt(3C)**
generate an IOT fault	**abort(3C)**
generate file name of controlling terminal	**ctermid(3S)**
generate file names	**glob(3C)**
generate hashing encryption	**crypt(3C)**
generate uniformly distributed pseudo-random numbers	**drand48(3C)**
generator, simple random-number	**rand(3C)**
get: character or data word from a stream file	**getc(3S)**
get: data pointer for binary search tree	**tsearch(3C)**
get: date and time more precisely (Version 7 compatibility only)	**ftime(2)**
get: diskless cnode ID of local machine	**cnodeid(2)**
get: entries from a directory in a filesystem-independent format	**getdirentries(2)**
get: entries from name list	**nlist(3C)**
get: entry from `group()` file	**getgrent(3C)**
get: file size limits and break value, get or set	**ulimit(2)**
get: file status	**stat(2)**
get: file system description file entry	**getmntent(3X)**
get: file system descriptor file entry (BSD 4.2 compatibility only)	**getfsent(3X)**
get: file system statistics	**statfs(2)**
get: list of active nodes in diskless cluster	**cnodes(2)**
get: message from an NLS message catalogue	**catgetmsg(3C)**
get: message queue	**msgget(2)**
get: mounted file system statistics	**ustat(2)**
get: name and version of current HP-UX system	**uname(2)**
get: name of current host	**gethostname(2)**
get: NLS program message	**catgets(3C)**
get: option letter from argument vector	**getopt(3C)**
get: path-name of current working directory	**getcwd(3C)**
get: pointer for I/O operations on a stream file, get or reposition	**fseek(3S)**
get: pointer to login name in `utmp()`	**getlogin(3C)**
get: process and child process times	**times(2)**
get: process context for context-dependent file search	**getcontext(2)**
get: process priority	**getpriority(2)**
get: process, process group, or parent process ID	**getpid(2)**
get: real or effective user or group ID	**getuid(2)**
get: set of semaphores	**semget(2)**
get: shared memory segment	**shmget(2)**
get: system clock date and time	**gettimeofday(2)**
get: time	**time(2)**
get: value of process interval timer	**getitimer(2)**
get: wide character from a stream file	**getwc(3C)**
get access control list (ACL) information	**getacl(2)**
`getaccess()` – get a user's effective access rights to a file	**getaccess(2)**
`getacl()`, `fgetacl()` – get access control list (ACL) information	**getacl(2)**
get address of connected peer	**getpeername(2)**
get and/or set signal stack context	**sigstack(2)**
get a silence value	**AGetSilenceValue(3X)**
`getaudid()` – get audit ID (`aid()`) for current process	**getaudid(2)**
get audit ID (`aid()`) for current process	**getaudid(2)**
get audit process flag for calling process	**getaudproc(2)**
`getaudproc()` – get audit process flag for calling process	**getaudproc(2)**
get a user's effective access rights to a file	**getaccess(2)**
get best audio attribute setting for specified controller	**ABestAudioAttributes(3X)**
get bit order used for one-bit-per-sample data	**ASoundBitOrder(3X)**
get byte order of audio data accepted by audio controller for this connection	**ASoundByteOrder(3X)**
`getcccid()` – get cluster configuration file entry matching specified `id()`	**getccent(3C)**

Index
Volume 2

Description	Entry Name(Section)
`getccent()` – get entry in cluster configuration file	**getccent(3C)**
`getccnam()` – get cluster configuration file entry matching specified `name()`	**getccent(3C)**
`getcdf()` – manipulate context-dependent file path names	**getcdf(3C)**
`getc()`, `fgetc()` – get character from a stream file	**getc(3S)**
`getchar()` – get character from *standard input* file	**getc(3S)**
`getclock` – get current value of system-wide clock	**getclock(3C)**
get configurable pathname variables	**pathconf(2)**
get configurable system variables	**sysconf(2)**
get connection number for specified audio server connection	**AConnectionNumber(3X)**
`getcontext()` – return the process context for context-dependent file search	**getcontext(2)**
get current value of system-wide clock	**getclock(3C)**
`getcwd()` – get path-name of current working directory	**getcwd(3C)**
get D/A output channels existing on current hardware	**AOutputChannels(3X)**
get data formats for a specifed file format	**AGetDataFormats(3X)**
`getdate()` – convert user format date and time	**getdate(3C)**
`getdirentries()` – get entries from a directory in a filesystem-independent format	**getdirentries(2)**
`getdiskbyname()` – get disk description by its name	**getdiskbyname(3C)**
get disk description by its name	**getdiskbyname(3C)**
`getdomainname()` – get name of current NIS domain	**getdomainname(2)**
`getegid()` – get effective group ID	**getuid(2)**
`getenv()` – return value for environment name	**getenv(3C)**
`geteuid()` – get effective user ID	**getuid(2)**
`getevent()` – get events and system calls currently being audited	**getevent(2)**
get events and system calls currently being audited	**getevent(2)**
`getexportent()` – access exported file system information	**exportent(3N)**
`getexportopt()` – access exported file system information	**exportent(3N)**
`getfh()` – file handle for file on remote node	**getfh(2)**
get file attributes of specified file	**AGetAFileAttributes(3X)**
get file format of specified file	**AQueryAFile(3X)**
get file handle for file on remote node	**getfh(2)**
get file system statistics	**statfsdev(3C)**
get first event found in audio event queue	**ACheckEvent(3X)**
get first event in audio event queue that matches mask	**ACheckMaskEvent(3X)**
get first matching event in audio event queue	**AMaskEvent(3X)**
get foreground process group ID	**tcgetpgrp(3C)**
`getfsent()` – get next line in file system descriptor file	**getfsent(3X)**
`getfsfile()` – search descriptor file for ordinary file entry	**getfsent(3X)**
`getfsspec()` – search descriptor file for special (device) file entry	**getfsent(3X)**
`getfstype()` – search descriptor file for specified file type entry	**getfsent(3X)**
`getgid()` – get real group ID	**getuid(2)**
`getgrent()` – get next entry in `group()` file	**getgrent(3C)**
`getgrgid()` – get entry from `group()` file that matches `gid()`	**getgrent(3C)**
`getgrnam()` – get entry from `group()` file that matches group name `name()`	**getgrent(3C)**
`getgroups()` – get group access list	**getgroups(2)**
`gethcwd()` – get path-name of current working directory including diskless hidden directories	**getcwd(3C)**
`gethostbyaddr()` – get network host entry	**gethostent(3N)**
`gethostbyname()` – get network host entry	**gethostent(3N)**
`gethostent()` – get network host entry	**gethostent(3N)**
`gethostname()` – get name of current host	**gethostname(2)**
get information about shared library	**shl_load(3X)**
`getitimer()` – get value of process interval timer	**getitimer(2)**
get legal user shells	**getusershell(3C)**
get list of A/D input channels on current hardware	**AInputChannels(3X)**
get list of data formats supported by audio controller	**ADataFormats(3X)**
`getlocale()` – get the locale of a program	**setlocale(3C)**
`getlogin()` – get pointer to login name in `utmp()`	**getlogin(3C)**
get major version number of protocol used by audio server	**AProtocolVersion(3X)**
get maximum input gain supported by audio controller	**AMaxInputGain(3X)**

Description	Entry Name(Section)
get maximum output gain supported by audio controller	**AMaxOutputGain(3X)**
get minimum input gain supported by audio controller	**AMinInputGain(3X)**
get minimum output gain supported by audio controller	**AMinOutputGain(3X)**
get minor revision number of protocol used by audio server	**AProtocolRevision(3X)**
`getmntent()` – get a file system description file entry	**getmntent(3X)**
`get_myaddress()` – get machine's IP address	**rpc(3C)**
get name of audio controller (string) passed to AOpenAudio()	**AAudioString(3X)**
get name of current NIS domain	**getdomainname(2)**
`getnetbyaddr()`: get network entry	**getnetent(3N)**
`getnetbyname()`: get network entry	**getnetent(3N)**
`getnetent()`: get network entry	**getnetent(3N)**
`getnetgrent()` – get network group entry	**getnetgrent(3C)**
get network entry	**getnetent(3N)**
get network group entry	**getnetgrent(3C)**
get network host entry	**gethostent(3N)**
get number of events in queue for specified server connection	**AEventsQueued(3X)**
`getopt()`, `optarg`, `optind`, `opterr` – get option letter from argument vector	**getopt(3C)**
get or set audit files	**audctl(2)**
get or set tty baud rate	**cfspeed(3C)**
`getpass()` – read a password from terminal while suppressing echo	**getpass(3C)**
`getpeername()` – get address of connected peer	**getpeername(2)**
`getpgrp2()` – get process group ID of specified process	**getpid(2)**
`getpgrp()` – get process group ID	**getpid(2)**
`getpid()` – get process ID	**getpid(2)**
get play volume or record gain of specified transaction	**AGetGain(3X)**
`getppid()` – get parent process ID	**getpid(2)**
`getpriority` – get process priority	**getpriority(2)**
`getprotobyname()` – get protocol entry	**getprotoent(3N)**
`getprotobynumber()` – get protocol entry	**getprotoent(3N)**
get protocol entry	**getprotoent(3N)**
`getprotoent()` – get protocol entry	**getprotoent(3N)**
`getpwent()` – get next password file entry	**getpwent(3C)**
`getpw()` – get name from UID (obsolete)	**getpw(3C)**
`getpwnam()` – get password file entry matching login name `name()`	**getpwent(3C)**
`getpwuid()` – get password file entry matching `uid()`	**getpwent(3C)**
`getrlimit()` – set system resource consumption limit	**getrlimit(2)**
`getrpcbyname()`: get RPC entry	**getrpcent(3C)**
`getrpcbynumber()`: get RPC entry	**getrpcent(3C)**
`getrpcent()`: get RPC entry	**getrpcent(3C)**
get RPC entry	**getrpcent(3C)**
`getrpcport()` – get RPC port number	**getrpcport(3N)**
get RPC port number	**getrpcport(3N)**
`getservbyname()`: get service entry	**getservent(3N)**
`getservbyport()`: get service entry	**getservent(3N)**
`getservent()`: get service entry	**getservent(3N)**
get service entry	**getservent(3N)**
`gets()`, `fgets()` – get a string from a *standard input* stream	**gets(3S)**
get socket address	**getsockname(2)**
`getsockname()` – get socket address	**getsockname(2)**
`getsockopt()` – get options on sockets	**getsockopt(2)**
`getspwaid()` – get next secure password file audit ID	**getspwent(3C)**
`getspwent()` – get next secure password file entry	**getspwent(3C)**
`getspwnam()` – get secure password file entry matching login name `name()`	**getspwent(3C)**
`getspwuid()` – get secure password file entry matching `uid()`	**getspwent(3C)**
get status of specified transaction	**AGetTransStatus(3X)**
`getsubopt()` – parse suboptions from a string	**getsubopt(3C)**
get system or monitor audio channel gain	**AGetSystemChannelGain(3X)**
get system resource consumption limit	**getrlimit(2)**

Index
Volume 2

Description	Entry Name(Section)
get the locale of a program	**setlocale(3C)**
get the name of a slave pty	**ptsname(3C)**
`gettimeofday()` – get system clock date and time	**gettimeofday(2)**
`gettimer` – get value of a per-process timer	**gettimer(3C)**
get transaction channel gain	**AGetChannelGain(3X)**
`gettransient()` – get a program number in the transient range	**rpc(3C)**
get tty device operating parameters	**tcattribute(3C)**
get types of input sources existing on current hardware	**AInputSources(3X)**
get types of output destinations existing on current hardware	**AOutputDestinations(3X)**
`getuid()` – get real user ID	**getuid(2)**
`getusershell()` – get legal user shells	**getusershell(3C)**
`getutent()` – get pointer to next entry in a `utmp()` file	**getut(3C)**
`getutid()` – get pointer to entry matching `id()` in a `utmp()` file	**getut(3C)**
`getutline()` – get pointer to entry matching `line()` in a `utmp()` file	**getut(3C)**
get value of a per-process timer	**gettimer(3C)**
get vendor name of audio server for this connection	**AServerVendor(3X)**
get vendor release number of audio server for this connection	**AVendorRelease(3X)**
`getwc()`, `fgetwc()` – get wide character from a stream file	**getwc(3C)**
`getwchar()` – get wide character from *standard input* file	**getwc(3C)**
`getw()` – get data word (integer) from a stream file	**getc(3S)**
`getws()`, `fgetws()` – get a wide string from a *standard input* stream	**getws(3C)**
`gfrt0.o`, `gcrt0.o` – C and Pascal execution startup routines	**crt0(3)**
`glob()`: – file name generation function	**glob(3C)**
`globfree()`: – file name generation function	**glob(3C)**
`gmtime()` – convert date and time to Greenwich Mean Time	**ctime(3C)**
goto, save/restore stack environment for non-local	**setjmp(3C)**
GPIO: return status lines of GPIO card	**gpio_get_status(3I)**
GPIO: set control lines on GPIO card	**gpio_set_ctl(3I)**
`gpio_get_status()` – return status lines of GPIO card	**gpio_get_status(3I)**
`gpio_set_ctl()` – set control lines on GPIO card	**gpio_set_ctl(3I)**
group access list: get group access list	**getgroups(2)**
group access list: initialize group access list	**initgroups(3C)**
group access list: set group access list	**setgroups(2)**
group and/or owner, change in access control list (ACL)	**chownacl(3C)**
group and owner of a file, change	**chown(2)**
group entry, network, get or set	**getnetgrent(3C)**
`group()` file, get entry from	**getgrent(3C)**
group ID: get real or effective group ID	**getuid(2)**
group ID: set group ID	**setuid(2)**
group ID: set real, effective, and/or saved group or user IDs	**setresuid(2)**
group ID, create session and set process	**setsid(2)**
group ID, foreground process, get	**tcgetpgrp(3C)**
group ID, foreground process, set	**tcsetpgrp(3C)**
group ID for job control, set process	**setpgid(2)**
group of processes, send a signal to a process or a	**kill(2)**
`gsignal()` – raise a software signal	**ssignal(3C)**
`gtty()`, `stty()` – control terminal device (Version 6 compatibility only)	**stty(2)**
halt or start auditing system	**audctl(2)**
handler, establish a cleanup	**pfm_$cleanup(3)**
handler for this connection, add audio event	**AtInitialize(3X)**
handler, release a cleanup	**pfm_$rls_cleanup(3)**
handler, reset a cleanup	**pfm_$reset_cleanup(3)**
hardware capabilities, check for presence of	**is_hw_present(3C)**
hashing encryption, generate	**crypt(3C)**
hash search tables, manage	**hsearch(3C)**
`hasmntopt()` – search mount option field in file system description file	**getmntent(3X)**
`havedisk()` – get performance data from remote kernel	**rstat(3N)**
`hcreate()` – allocate space for new hash search table	**hsearch(3C)**

Description	Entry Name(Section)
`hdestroy()` – destroy existing hash search table	**hsearch(3C)**
`herror()` – resolver routines	**resolver(3N)**
hierarchy, directory, recursively descend a	**ftw(3C)**
hold signal upon receipt	**sigset(2V)**
host and network byte order, convert values between	**byteorder(3N)**
host cpu, set name of	**sethostname(2)**
host CPU, set NetIPC node name of	**ipcsetnodename(2)**
host, current, get name of	**gethostname(2)**
host, obtain NetIPC node name of current	**ipcgetnodename(2)**
HP3000-mode packed decimal library	**hppac(3X)**
HP-IB: allow interface to enable SRQ line on HP-IB	**hpib_rqst_srvce(3I)**
HP-IB: change active controllers on HP-IB	**hpib_pass_ctl(3I)**
HP-IB: conduct a serial poll on HP-IB	**hpib_spoll(3I)**
HP-IB: conduct parallel poll on HP-IB	**hpib_ppoll(3I)**
HP-IB: control Attention line on HP-IB	**hpib_atn_ctl(3I)**
HP-IB: control EOI mode for HP-IB file	**hpib_eoi_ctl(3I)**
HP-IB: control response to parallel poll on HP-IB	**hpib_card_ppoll_resp(3I)**
HP-IB: control the Remote Enable line on HP-IB	**hpib_ren_ctl(3I)**
HP-IB: define interface parallel poll response	**hpib_ppoll_resp_ctl(3I)**
HP-IB: enable/disable odd parity on ATN commands	**hpib_parity_ctl(3I)**
HP-IB: perform I/O with an HP-IB channel from buffers	**hpib_io(3I)**
HP-IB: return status of HP-IB interface	**hpib_bus_status(3I)**
HP-IB: send command bytes over HP-IB	**hpib_send_cmnd(3I)**
HP-IB: set HP-IB bus address for an interface	**hpib_address_ctl(3I)**
HP-IB: stop activity on specified HP-IB	**hpib_abort(3I)**
HP-IB: wait until a particular parallel poll value occurs	**hpib_wait_on_ppoll(3I)**
HP-IB: wait until the requested status condition becomes true	**hpib_status_wait(3I)**
`hpib_abort()` – stop activity on specified HP-IB	**hpib_abort(3I)**
`hpib_address_ctl()` – set HP-IB bus address for an interface	**hpib_address_ctl(3I)**
`hpib_atn_ctl()` – control Attention line on HP-IB	**hpib_atn_ctl(3I)**
HP-IB bus address for an interface, set	**hpib_address_ctl(3I)**
`hpib_bus_status()` – return status of HP-IB interface	**hpib_bus_status(3I)**
`hpib_card_ppoll_resp()` – control response to parallel poll on HP-IB	**hpib_card_ppoll_resp(3I)**
`hpib_eoi_ctl()` – control EOI mode for HP-IB file	**hpib_eoi_ctl(3I)**
HP-IB/GPIO/parallel channel, perform low-overhead I/O on an	**io_burst(3I)**
`hpib_io()` – perform I/O with an HP-IB channel from buffers	**hpib_io(3I)**
`hpib_parity_ctl()` – enable/disable odd parity on ATN commands	**hpib_parity_ctl(3I)**
`hpib_pass_ctl()` – change active controllers on HP-IB	**hpib_pass_ctl(3I)**
`hpib_ppoll()` – conduct parallel poll on HP-IB	**hpib_ppoll(3I)**
`hpib_ppoll_resp_ctl()` – define interface parallel poll response	**hpib_ppoll_resp_ctl(3I)**
`hpib_ren_ctl()` – control the Remote Enable line on HP-IB	**hpib_ren_ctl(3I)**
`hpib_rqst_srvce()` – allow interface to enable SRQ line on HP-IB	**hpib_rqst_srvce(3I)**
`hpib_send_cmnd()` – send command bytes over HP-IB	**hpib_send_cmnd(3I)**
`hpib_spoll()` – conduct a serial poll on HP-IB	**hpib_spoll(3I)**
`hpib_status_wait()` – wait until the requested status condition becomes true	**hpib_status_wait(3I)**
`hpib_wait_on_ppoll()` – wait until a particular parallel poll value occurs	**hpib_wait_on_ppoll(3I)**
HPPAC*: HP3000-mode packed decimal library	**hppac(3X)**
`hsearch()` – hash table search routine	**hsearch(3C)**
`htonl()` – convert values between host and network byte order	**byteorder(3N)**
`htons()` – convert values between host and network byte order	**byteorder(3N)**
hyperbolic trigonometric functions, inverse	**asinh(3M)**
hyperbolic trigonometric functions	**sinh(3M)**
hypotenuse of a right triangle	**hypot(3M)**
`hypot()` – Euclidean distance function	**hypot(3M)**
`ICONV`, `ICONV1`, `ICONV2` – code set conversion routines	**iconv(3C)**
`iconvsize()`, `iconvopen()`, `iconvclose()`, `iconvlock()` – code set conversion routines	**iconv(3C)**
ID, create session and set process group	**setsid(2)**
ID, foreground process group, get	**tcgetpgrp(3C)**

Index
Volume 2

Description	Entry Name(Section)
ID, foreground process group, set	**tcsetpgrp(3C)**
ID for job control, set process group	**setpgid(2)**
ID, get real or effective user or group	**getuid(2)**
ID of local machine, get diskless cnode	**cnodeid(2)**
ID, set user or group	**setuid(2)**
ID to file path, map device	**devnm(3)**
`idtolang()` – convert NLS language ID number to language name	**langinfo(3C)**
ignorable signals mask, set current	**sigsetmask(2)**
ignore signal	**sigset(2V)**
ignore signals	**sigblock(2)**
in-core state with its state on disk, synchronize a file's	**fsync(2)**
increase data segment space allocation	**brk(2)**
`index()` – BSD portability string routine	**string(3C)**
`inet_addr()` – Internet address manipulation routines	**inet(3N)**
`inet_lnaof()` – Internet address manipulation routines	**inet(3N)**
`inet_makeaddr()` – Internet address manipulation routines	**inet(3N)**
`inet_netof()` – Internet address manipulation routines	**inet(3N)**
`inet_network()` – Internet address manipulation routines	**inet(3N)**
`inet_ntoa()` – Internet address manipulation routines	**inet(3N)**
INFINITY, test for	**isinf(3M)**
information about users on remote machines, return	**rnusers(3N)**
information, access exported file system	**exportent(3N)**
information, NLS, about native languages	**langinfo(3C)**
information, NLS, about native languages	**nl_langinfo(3C)**
inhibit asynchronous faults; allow time-sliced task switching	**pfm_$inhibit_faults(3)**
inhibit asynchronous faults	**pfm_$inhibit(3)**
`initgroups()` – initialize group access list	**initgroups(3C)**
initialize group access list	**initgroups(3C)**
initialize, manipulate, and test signal sets	**sigsetops(3C)**
initialize NetIPC option buffer	**initopt(3N)**
initialize semaphore in mapped file or anonymous memory region	**msem_init(2)**
initialize the NLS environment of a program	**nl_init(3C)**
initialize the process fault manager package	**pfm_$init(3)**
initiate an audio widget play operation	**AuInvokePlay(3X)**
initiate an audio widget record operation	**AuInvokeRecord(3X)**
initiate connection on a socket	**connect(2)**
initiate transaction and return transaction ID and SStream structure	**APlaySStream(3X)**
initiate transaction; return transaction ID and SStreams structure	**ARecordSStream(3X)**
`initopt()` – initialize NetIPC option buffer	**initopt(3N)**
`innetgr()` – get network group entry	**getnetgrent(3C)**
input conversion, formatted read from stream file or character string	**scanf(3S)**
input conversion, formatted, to a varargs argument	**vscanf(3S)**
input/output, buffered, standard stream file package	**stdio(3S)**
input/output to a stream file, buffered binary	**fread(3S)**
input stream, push character back into	**ungetc(3S)**
input stream, push wide character back into	**ungetwc(3C)**
input string from a *standard input* stream	**gets(3S)**
input wide string from a *standard input* stream	**getws(3C)**
integer absolute value, return	**abs(3C)**
integer, convert string to long	**strtol(3C)**
integer, convert wide character string to long	**wcstol(3C)**
integer data in a machine-independent fashion, access long	**sputl(3X)**
integer division and remainder	**div(3C)**
integer, long, convert to string,	**ltostr(3C)**
integers, convert between 3-byte integers and long integers	**l3tol(3C)**
integer to base-64 ASCII string, convert long	**a64l(3C)**
interface: define HP-IB interface parallel poll response	**hpib_ppoll_resp_ctl(3I)**
interface, control DMA allocation for an	**io_dma_ctl(3I)**

Description	Entry Name(Section)
interface, GPIO: return status lines of GPIO card	**gpio_get_status(3I)**
interface, GPIO: set control lines on GPIO card	**gpio_set_ctl(3I)**
interface, HP-IB: allow interface to enable SRQ line on HP-IB	**hpib_rqst_srvce(3I)**
interface, HP-IB: change active controllers on HP-IB	**hpib_pass_ctl(3I)**
interface, HP-IB: conduct a serial poll on HP-IB	**hpib_spoll(3I)**
interface, HP-IB: conduct parallel poll on HP-IB	**hpib_ppoll(3I)**
interface, HP-IB: control EOI mode for HP-IB file	**hpib_eoi_ctl(3I)**
interface, HP-IB: control response to parallel poll on HP-IB	**hpib_card_ppoll_resp(3I)**
interface, HP-IB: control the HP-IB interface Remote Enable line	**hpib_ren_ctl(3I)**
interface, HP-IB: perform I/O with an HP-IB channel from buffers	**hpib_io(3I)**
interface, HP-IB: return status of HP-IB interface	**hpib_bus_status(3I)**
interface, HP-IB: send command bytes over HP-IB	**hpib_send_cmnd(3I)**
interface, HP-IB: stop activity on specified HP-IB	**hpib_abort(3I)**
interface, HP-IB: wait until a particular parallel poll value occurs	**hpib_wait_on_ppoll(3I)**
interface, HP-IB: wait until the requested status condition becomes true	**hpib_status_wait(3I)**
interface, Network Information Service client	**ypclnt(3C)**
interface parallel poll response, define	**hpib_ppoll_resp_ctl(3I)**
interface, reset an I/O	**io_reset(3I)**
interface, set HP-IB bus address for an	**hpib_address_ctl(3I)**
interface, unlock or lock an I/O	**io_lock(3I)**
interleaved paging/swapping, add a swap device for	**swapon(2)**
Internet address manipulation routines	**inet(3N)**
interprocess channel, create an	**pipe(2)**
interprocess communication package, standard	**stdipc(3C)**
interrupt, atomically release blocked signals and wait for	**sigpause(2)**
interrupt (fault) conditions, define for I/O device	**io_on_interrupt(3I)**
interrupts for the associated `eid()`, disable or enable I/O	**io_interrupt_ctl(3I)**
interval, suspend execution for	**sleep(3C)**
interval timer, set or get value of process	**getitimer(2)**
introduction to subroutines and libraries	**intro(3)**
introduction to system calls	**intro(2)**
`intro()` – introduction to subroutines and libraries	**intro(3)**
inverse hyperbolic trigonometric functions	**asinh(3M)**
I/O: GPIO card, return status lines of	**gpio_get_status(3I)**
I/O: GPIO card, set control lines on	**gpio_set_ctl(3I)**
`io_burst()` – perform low-overhead I/O on an HP-IB/GPIO/parallel channel	**io_burst(3I)**
I/O, control character device special file	**ioctl(2)**
`ioctl()` – control character device special file	**ioctl(2)**
I/O data path width (in bits), set	**io_width_ctl(3I)**
I/O device interrupt (fault) control	**io_on_interrupt(3I)**
`io_dma_ctl()` – control DMA allocation for an interface	**io_dma_ctl(3I)**
`io_eol_ctl()` – set up I/O read termination character on special file	**io_eol_ctl(3I)**
`io_get_term_reason()` – determine how last read terminated	**io_get_term_reason(3I)**
I/O interface, reset an	**io_reset(3I)**
I/O interface, unlock or lock an	**io_lock(3I)**
`io_interrupt_ctl()` – enable/disable interrupts for the associated `eid()`	**io_interrupt_ctl(3I)**
I/O interrupts for the associated `eid()`, disable or enable	**io_interrupt_ctl(3I)**
`io_lock()`, `io_unlock()` – lock and unlock an I/O interface	**io_lock(3I)**
I/O multiplexing, synchronous	**select(2)**
I/O on an HP-IB/GPIO/parallel channel, perform low-overhead	**io_burst(3I)**
`io_on_interrupt()` – device I/O interrupt (fault) control	**io_on_interrupt(3I)**
I/O operations on a stream file, get or reposition pointer for	**fseek(3S)**
I/O operations, set time limit for	**io_timeout_ctl(3I)**
I/O pipe to or from a process, open or close	**popen(3S)**
I/O read, determine how last terminated	**io_get_term_reason(3I)**
I/O read termination character on special file, set up	**io_eol_ctl(3I)**
`io_reset()` – reset an I/O interface	**io_reset(3I)**
`io_speed_ctl()` – inform system of required transfer speed	**io_speed_ctl(3I)**

Index
Volume 2

Description	Entry Name(Section)
IOT fault, generate an	**abort(3C)**
`io_timeout_ctl()` – establish a time limit for I/O operations	**io_timeout_ctl(3I)**
I/O to a stream file, buffered binary	**fread(3S)**
`io_unlock()` – unlock an I/O interface	**io_lock(3I)**
`io_width_ctl()` – set width (in bits) of data path	**io_width_ctl(3I)**
I/O with an HP-IB channel from buffers, perform	**hpib_io(3I)**
`ipcconnect()()` – request connection to another process	**ipcconnect(2)**
`ipccontrol()()` – perform special operations on NetIPC sockets	**ipccontrol(2)**
`ipccreate()()` – create a call socket	**ipccreate(2)**
`ipcdest()()` – create a destination descriptor	**ipcdest(2)**
`ipcerrmsg()` – provide text describing NetIPC error number	**ipcerrmsg(3N)**
`ipcgetnodename()` – obtain NetIPC node name of current host	**ipcgetnodename(2)**
`ipclookup()()` – obtain a destination descriptor	**ipclookup(2)**
`ipcname()()` – associate name with call socket or destination call socket	**ipcname(2)**
`ipcnamerase()()` – delete name associated with a call socket or destination call socket	**ipcnamerase(2)**
`ipcrecvcn()()` – receive connection request on a call socket	**ipcrecvcn(2)**
`ipcrecv()()` – establish or receive data on NetIPC virtual circuit connection	**ipcrecv(2)**
`ipcselect()()` – determine status of call socket or VC socket	**ipcselect(2)**
`ipcsend()()` – send data on a virtual circuit connection	**ipcsend(2)**
`ipcsetnodename()` – set NetIPC node name of host CPU	**ipcsetnodename(2)**
`ipcshutdown()()` – release a descriptor	**ipcshutdown(2)**
IP port, bind socket to a privileged	**bindresvport(3N)**
`is_68010_present()` – check for MC68010 system microprocessor	**is_hw_present(3C)**
`is_68881_present()` – check for MC68881 math coprocessor	**is_hw_present(3C)**
`is_98248A_present()` – check for floating-point accelerator card	**is_hw_present(3C)**
`is_98635A_present()` – check for floating-point math card	**is_hw_present(3C)**
`isalnum()` – character is alphanumeric	**ctype(3C)**
`isalpha()` – character is alpha	**ctype(3C)**
`isascii()` – character is 7-bit ASCII code	**ctype(3C)**
`ISASCII` – character is 7-bit ASCII code	**wctype(3C)**
`isatty()` – find name of a terminal	**ttyname(3C)**
`iscntrl()` – character is a control character	**ctype(3C)**
`isdigit()` – character is a digit	**ctype(3C)**
`isgraph()` – character is a visible character	**ctype(3C)**
`isinff()`, `isinf` – test for INFINITY	**isinf(3M)**
`isinf()`, `isinff` – test for INFINITY	**isinf(3M)**
`islower()` – character is lowercase	**ctype(3C)**
`isnanf()`, `isnan()` – test for NaN	**isnan(3M)**
`isnan()`, `isnanf()` – test for NaN	**isnan(3M)**
`isprint()` – character is a printing character	**ctype(3C)**
`ispunct()` – character is punctuation	**ctype(3C)**
`isspace()` – character is whitespace	**ctype(3C)**
issue a shell command	**system(3S)**
`isupper()` – character is uppercase	**ctype(3C)**
`iswalnum` – character is alphanumeric	**wctype(3C)**
`iswalpha` – character is alpha	**wctype(3C)**
`iswcntrl` – character is a control character	**wctype(3C)**
`iswdigit` – character is a digit	**wctype(3C)**
`iswgraph` – character is a visible character	**wctype(3C)**
`iswlower` – character is lowercase	**wctype(3C)**
`iswprint` – character is a printing character	**wctype(3C)**
`iswpunct` – character is punctuation	**wctype(3C)**
`iswspace` – character is whitespace	**wctype(3C)**
`iswupper` – character is uppercase	**wctype(3C)**
`iswxdigit` – character is a hexadecimal digit	**wctype(3C)**
`isxdigit()` – character is a hexadecimal digit	**ctype(3C)**
`j0()`, `j1()`, `jn()`, `y0()`, `y1()`, `yn()` – Bessel functions	**bessel(3M)**
`j1()` – Bessel function	**bessel(3M)**

Description	Entry Name(Section)
`jn()` – Bessel function	**bessel(3M)**
job control, set process group ID for	**setpgid(2)**
keep track of remotely mounted file systems	**mount(3N)**
kernel, remote, get performance data from	**rstat(3N)**
`killpg()` – 4.2 BSD-compatible `kill()` system call	**bsdproc(2)**
`kill()`, `raise()` – send signal to process or group of processes	**kill(2)**
`kill()` system call, 4.2 BSD-compatible	**bsdproc(2)**
`l3tol()` – convert 3-byte integer to long integer	**l3tol(3C)**
`l64a()` – convert long integer to base-64 value ASCII string	**a64l(3C)**
`labs()` – return long integer absolute value	**abs(3C)**
`langinfo()` – obtain NLS string form of local language variable	**langinfo(3C)**
`langtoid()` – convert NLS language name to language ID number	**langinfo(3C)**
languages, NLS information about native (local)	**langinfo(3C)**
languages, NLS information about native (local)	**nl_langinfo(3C)**
last I/O read, determine how terminated	**io_get_term_reason(3I)**
last locations of allocated regions in program	**end(3C)**
`ldecvt()`, (`_ldecvt()`) – convert long double to string	**ldcvt(3C)**
`_ldecvt()`, `_ldfcvt()`, `_ldgcvt()` – convert long double to string	**ldcvt(3C)**
`ldexp()`, `frexp()`, `modf()` – split floating-point into mantissa and exponent	**frexp(3C)**
`ldfcvt()`, (`_ldfcvt()`) – convert long double to string	**ldcvt(3C)**
`ldgcvt()`, (`_ldgcvt()`) – convert long double to string	**ldcvt(3C)**
`ldiv()` – long integer division and remainder	**div(3C)**
legal user shells, get	**getusershell(3C)**
length of string, find	**string(3C)**
length of wide string, find	**wcstring(3C)**
`lgamma()`, `gamma()`, `signgam()` – log gamma function	**gamma(3M)**
libraries and subroutines, introduction to	**intro(3)**
library, packed decimal, HP3000-mode	**hppac(3X)**
library routines for external data representation	**xdr(3C)**
library routines for remote procedure calls	**rpc(3C)**
limit for I/O operations, set time	**io_timeout_ctl(3I)**
limit, get or set system resource consumption	**getrlimit(2)**
linear table search with optional update	**lsearch(3C)**
line connection, establish an out-bound terminal	**dial(3C)**
line control functions, tty	**tccontrol(3C)**
line on HP-IB, control the Remote Enable	**hpib_ren_ctl(3I)**
lines of GPIO card, return status	**gpio_get_status(3I)**
lines on GPIO card, set control	**gpio_set_ctl(3I)**
line, SRQ, on HP-IB, allow interface to enable	**hpib_rqst_srvce(3I)**
`link()` – link additional name to an existing file	**link(2)**
link, symbolic, read value of	**readlink(2)**
link to a file, make a symbolic	**symlink(2)**
listen for connections on a socket	**listen(2)**
`listen()` – listen for connections on a socket	**listen(2)**
list, get group access	**getgroups(2)**
list, initialize group access	**initgroups(3C)**
list, name, get entries from	**nlist(3C)**
list, print formatted output of a varargs argument	**vprintf(3S)**
list, set group access	**setgroups(2)**
load shared library	**shl_load(3X)**
`localeconv()` – query numeric formatting conventions of current locale	**localeconv(3C)**
locale, current, query numeric formatting conventions of	**localeconv(3C)**
locale of a program, get or set the	**setlocale(3C)**
local machine, get diskless cnode ID of	**cnodeid(2)**
local (native) languages, NLS information about	**langinfo(3C)**
local (native) languages, NLS information about	**nl_langinfo(3C)**
`localtime()` – convert date and time to local timezone	**ctime(3C)**
location of character in memory, find	**memory(3C)**

Index
Volume 2

Description	Entry Name(Section)
locations beyond allocated program regions, first	**end(3C)**
lock a semaphore	**msem_lock(2)**
`lockf()` – provide semaphores and record locking on files	**lockf(2)**
locking on files, provide semaphores and record	**lockf(2)**
lock or unlock an I/O interface	**io_lock(3I)**
lock process into memory after allocating data and stack space	**datalock(3C)**
lock process, text, or data in memory	**plock(2)**
`log10f()`, `logf()`, `log2f()`, `log()`, `log10()`, `log2()` – logarithm functions	**exp(3M)**
`log10()`, `log()`, `log2()`, `logf()`, `log10f()`, `log2f()` – logarithm functions	**exp(3M)**
`log2f()`, `logf()`, `log10f()`, `log()`, `log10()`, `log2()` – logarithm functions	**exp(3M)**
`log2()`, `log()`, `log10()`, `logf()`, `log10f()`, `log2f()` – logarithm functions	**exp(3M)**
logarithm, exponential, power, square root, cube root functions	**exp(3M)**
`logb()`, `scalb()` – exponent manipulations	**ieee(3M)**
`logf()`, `log10f()`, `log2f()`, `log()`, `log10()`, `log2()` – logarithm functions	**exp(3M)**
log gamma function	**gamma(3M)**
login name in `utmp()`, get pointer to	**getlogin(3C)**
login name of the user, get character-string	**cuserid(3S)**
login name of user, obtain	**logname(3C)**
`log()`, `log10()`, `log2()`, `logf()`, `log10f()`, `log2f()` – logarithm functions	**exp(3M)**
`logname()` – return login name of user	**logname(3C)**
log, system, control	**syslog(3C)**
long double floating-point number to string, convert	**ldcvt(3C)**
long double-precision number, convert string to	**strtold(3C)**
long integer data in a machine-independent fashion, access	**sputl(3X)**
long integers and 3-byte integers, convert between	**l3tol(3C)**
long integer to base-64 ASCII string, convert	**a64l(3C)**
long integer to string, convert	**ltostr(3C)**
`_longjmp()` – restore stack environment after non-local goto	**setjmp(3C)**
look up symbol in shared library	**shl_load(3X)**
lowercase, translate characters to	**conv(3C)**
lowercase, translate wide characters to	**wconv(3C)**
low-overhead I/O on an HP-IB/GPIO/parallel channel, perform	**io_burst(3I)**
`lrand48()`, `nrand48()` – generate long-integer pseudo-random numbers	**drand48(3C)**
`lsearch()`, `lfind()` – linear search and update	**lsearch(3C)**
`lseek()` – move read/write file pointer; seek	**lseek(2)**
`lstat()`, (`stat()`, `fstat()`) – get file link status	**stat(2)**
`lsync()`, `sync()` – update super-block	**sync(2)**
`ltoa()`; convert long integer to ASCII decimal	**ltostr(3C)**
`ltol3()` – convert long integer to 3-byte integer	**l3tol(3C)**
`ltostr()`; convert long integer to string	**ltostr(3C)**
machine, get diskless cnode ID of local	**cnodeid(2)**
machines, return information about users on remote	**rnusers(3N)**
machines, write to specified remote	**rwall(3N)**
`madvise` – advise system of process' expected paging behavior	**madvise(2)**
main memory allocator	**malloc(3C)**
main memory space allocation, control	**malloc(3C)**
main memory space usage, display	**malloc(3C)**
make a directory file	**mkdir(2)**
make a directory, or a special or ordinary file	**mknod(2)**
make a FIFO special file	**mkfifo(3C)**
make a symbolic link to a file	**symlink(2)**
make a unique (usually temporary) file name	**mktemp(3C)**
`mallinfo()` – display memory space usage	**malloc(3C)**
`malloc()` – allocate block of main memory	**malloc(3C)**
`malloc`, `free()`, `realloc()`, `calloc()` `mallopt()`, `mallinfo()`, `memorymap()` – main memory allocator	**malloc(3C)**
`mallopt()` – control memory space allocation	**malloc(3C)**
manage a binary search tree	**tsearch(3C)**

Description	Entry Name(Section)
manage hash search tables	hsearch(3C)
management, program	pgm_$intro(3)
management, signal (`sigset()`, `sighold()`, `sigrelse()`, `sigignore()`, `sigpause()`)	sigset(2V)
managing signal exceptions	pfm_$intro(3)
manipulate disk quotas	quotactl(2)
manipulate, initialize, and test signal sets	sigsetops(3C)
manipulation routines, Internet address	inet(3N)
mantissa and exponent, split floating-point into	frexp(3C)
map device ID to file path	devnm(3)
map object into virtual memory	mmap(2)
mapped file or anonymous memory region, initialize semaphore in	msem_init(2)
mapped file or anonymous region, remove semaphore in	msem_remove(2)
mapped file, synchronize a	msync(2)
mapped region, unmap a	munmap(2)
mapping access protections, modify memory	mprotect(2)
map stream pointer to file descriptor	fileno(3S)
mask for file creation, set and get permissions	umask(2)
mask, set current ignorable signals	sigsetmask(2)
match filename patterns	fnmatch(3C)
matching routines, regular expression	regcomp(3C)
match routines for regular expressions	regexp(3X)
math: Bessel functions	bessel(3M)
math: complex absolute value function	hypot(3M)
math: copysign, remainder, classification, exponent manipulations	ieee(3M)
math: error function and complementary error function	erf(3M)
math: Euclidean distance (hypotenuse) function	hypot(3M)
math: exponential, logarithm, power, square root, cube root functions	exp(3M)
math: floating-point classification functions	fpclassify(3M)
math: floating-point mode control functions	fpgetround(3M)
math: floor, ceiling, remainder, absolute value, round-to-nearest functions	floor(3M)
math: hyperbolic trigonometric functions	sinh(3M)
math: inverse hyperbolic trigonometric functions	asinh(3M)
math: log gamma function	gamma(3M)
math: math library error-handling function	matherr(3M)
math: split floating-point into mantissa and exponent	frexp(3C)
math: test for INFINITY	isinf(3M)
math: test for NaN	isnan(3M)
math: trigonometric functions (degrees)	trigd(3M)
math: trigonometric functions	trig(3M)
math coprocessor or accelerator, check for presence of	is_hw_present(3C)
`matherr()` – math library error-handling function	matherr(3M)
math library error-handling function	matherr(3M)
`mblen()` – multibyte characters and strings conversions	multibyte(3C)
`mbstowcs()` – multibyte characters and strings conversions	multibyte(3C)
`mbtowc()` – multibyte characters and strings conversions	multibyte(3C)
`mcrt0.o`, `crt0.o` – C and Pascal execution startup routines	crt0(3)
`memchr()` – find first occurrence of character in memory area	memory(3C)
`memcmp()` – compare character with memory contents	memory(3C)
`memcpy()`, `memccpy()` – copy characters from memory to another memory location	memory(3C)
`memmove()` – move memory contents	memory(3C)
memory allocator for main memory	malloc(3C)
memory control operations, shared	shmctl(2)
memory, lock process into after allocating data and stack space	datalock(3C)
memory, lock process, text, or data in	plock(2)
`memorymap()` – display contents of memory allocator	malloc(3C)
memory, map object into virtual	mmap(2)
memory mapping access protections, modify	mprotect(2)
memory operations – copy, compare, test for contents, or set contents to value	memory(3C)

Index
Volume 2

Description	Entry Name(Section)
memory region, initialize semaphore in mapped file or anonymous	**msem_init(2)**
memory segment, get shared	**shmget(2)**
`memset()` – set area in memory to contain a specified character	**memory(3C)**
message catalog for reading, close or open NLS	**catopen(3C)**
message catalog support, RTE/MPE-style	**catread(3C)**
message catalogue, get message from an NLS	**catgetmsg(3C)**
message control operations	**mstctl(2)**
message for a status code, return an error	**error_$c_text(3)**
message from an NLS message catalogue, get	**catgetmsg(3C)**
message from a socket, receive	**recv(2)**
message, NLS program, get an	**catgets(3C)**
message queue, get	**msgget(2)**
message, send or receive message to or from message queue	**msgop(2)**
message, send to a socket	**send(2)**
messages, system error	**perror(3C)**
`mfrt0.o`, `frt0.o` – FORTRAN execution startup routines	**crt0(3)**
microprocessor, MC68010, check for presence of	**is_hw_present(3C)**
minimum I/O data transfer rate, inform system of required	**io_speed_ctl(3I)**
`mkdir()` – make a directory file	**mkdir(2)**
`mkfifo()` – make a FIFO special file	**mkfifo(3C)**
`mknod()` – make a directory, or a special or ordinary file	**mknod(2)**
`mknod()` – make a cnode-specific special file	**mknod(2)**
`mktemp()` – make a unique (temporary) file name	**mktemp(3C)**
`mktime()` – convert time into calendar time value	**ctime(3C)**
`mktimer` – allocate a per-process timer	**mktimer(3C)**
`mmap` – map object into virtual memory	**mmap(2)**
mode, EOI, for HP-IB file, control	**hpib_eoi_ctl(3I)**
mode (permissions) of file, change access	**chmod(2)**
`modf()`, `frexp()`, `ldexp()` – split floating-point into mantissa and exponent	**frexp(3C)**
modification and access times, set or update file	**utime(2)**
modify, add, or delete access control list entry	**setaclentry(3C)**
modify memory mapping access protections	**mprotect(2)**
module, and error texts for a status code, return subsystem,	**error_$c_get_text(3)**
monitor audio channel gain, get system or	**AGetSystemChannelGain(3X)**
monitor audio channel gain, set system or	**ASetSystemChannelGain(3X)**
monitor I/O conditions on multiple file descriptors	**poll(2)**
`monitor()` – prepare execution profile	**monitor(3C)**
`mount()`: keep track of remotely mounted file systems	**mount(3N)**
mount a file system	**vfsmount(2)**
mount a removable file system	**mount(2)**
mounted file systems, keep track of remotely	**mount(3N)**
mounted file system statistics, get	**ustat(2)**
`mount()` – mount a removable file system	**mount(2)**
move read/write file pointer; seek	**lseek(2)**
MPE clock value, return the	**clock(3X)**
MPE Native Language Support:	
append language ID to valid MPE file name	**nlappend(3X)**
check/convert time string to MPE internal format	**nlconvclock(3X)**
compare character arrays (key1, key2) using MPE collation table	**nlkeycompare(3X)**
compare strings; use MPE language-dependent collating sequence	**nlcollate(3X)**
convert ASCII number to MPE language-specific formatted number	**nlfmtnum(3X)**
convert date string to MPE packed date format	**nlconvcustdate(3X)**
convert MPE native language formatted number to ASCII	**nlconvnum(3X)**
convert string between phonetic and screen order using MPE table	**nlswitchbuf(3X)**
extract substring in string using MPE character set table	**nlsubstr(3X)**
format MPE date and time in localized format	**nlfmtdate(3X)**
format MPE packed date using custom date	**nlfmtcustdate(3X)**
format MPE packed date using localized format	**nlfmtcal(3X)**

Description	Entry Name(Section)
format MPE packed date using long calendar format	**nlfmtlongcal(3X)**
format MPE time of day using localized format	**nlfmtclock(3X)**
identify one- or multi-byte Asian character using MPE character table	**nljudge(3X)**
move, scan, case-shift strings using MPE character set table	**nlscanmove(3X)**
replace non-displayable string characters using MPE character set table	**nlrepchar(3X)**
return current user, data, or system default language	**nlgetlang(3X)**
return MPE calendar date	**calendar(3X)**
return MPE language-dependent information	**nlinfo(3X)**
return number conversion/formatting information for MPE routines	**nlnumspec(3X)**
return numeric date information in MPE format	**almanac(3X)**
search for string in a string using MPE character set definition	**nlfindstr(3X)**
translate ASCII strings to EBCDIC using MPE conversion table	**nltranslate(3X)**
MPE/RTE-style message catalog support	**catread(3C)**
`mprotect` – modify memory mapping access protections	**mprotect(2)**
`mrand48()`, `jrand48()` – generate signed long-integer pseudo-random numbers	**drand48(3C)**
`msem_init` – initialize semaphore in mapped file or anonymous memory region	**msem_init(2)**
`msem_lock` – lock a semaphore	**msem_lock(2)**
`msem_remove` – remove semaphore in mapped file or anonymous region	**msem_remove(2)**
`msem_unlock` – unlock a semaphore	**msem_unlock(2)**
`msgctl()` – message control operations	**mstctl(2)**
`msgget()` – get message queue	**msgget(2)**
`msgrcv()` – receive message from message queue	**msgop(2)**
`msgsnd()` – send message to message queue	**msgop(2)**
`msync` – synchronize a mapped file	**msync(2)**
multibyte characters and strings conversions	**multibyte(3C)**
multiplexing, synchronous I/O	**select(2)**
`munmap` – unmap a mapped region	**munmap(2)**
name:	
change the name of a file	**rename(2)**
create a name for a temporary file	**tmpnam(3S)**
find name of a terminal	**ttyname(3C)**
get character-string representation of user login name	**cuserid(3S)**
get entries from name list	**nlist(3C)**
get name and version of current HP-UX system	**uname(2)**
get name from UID (obsolete)	**getpw(3C)**
get name of current host	**gethostname(2)**
get pointer to login name in `utmp()`	**getlogin(3C)**
obtain user login name	**logname(3C)**
set the name of host cpu	**sethostname(2)**
name associated with a call socket or destination call socket, delete	**ipcnamerase(2)**
name, associate with call socket or destination call socket	**ipcname(2)**
name, get disk description by its	**getdiskbyname(3C)**
name of a slave pty, get the	**ptsname(3C)**
name of current host, obtain NetIPC node	**ipcgetnodename(2)**
name of current NIS domain, get or set	**getdomainname(2)**
name of host CPU, set NetIPC node	**ipcsetnodename(2)**
names, manipulate context-dependent file path	**getcdf(3C)**
NaN, test for	**isnan(3M)**
native languages, NLS information about	**langinfo(3C)**
native languages, NLS information about	**nl_langinfo(3C)**
`net_aton()` – network station address string conversion routines	**net_aton(3C)**
NetIPC error number, provide text describing	**ipcerrmsg(3N)**
NetIPC node name of current host, obtain	**ipcgetnodename(2)**
NetIPC node name of host CPU, set	**ipcsetnodename(2)**
NetIPC option buffer, add argument and data to	**addopt(3N)**
NetIPC option buffer, initialize	**initopt(3N)**
NetIPC option buffer, obtain option code and data from	**readopt(3N)**
NetIPC option, return number of bytes needed by a	**optoverhead(3N)**

Index
Volume 2

Description	Entry Name(Section)
NetIPC sockets, perform special operations on	**ipccontrol(2)**
NetIPC virtual circuit connection, establish or receive data on	**ipcrecv(2)**
net_ntoa() – network station address string conversion routines	**net_aton(3C)**
network and host byte order, convert values between	**byteorder(3N)**
network entry, get or set	**getnetent(3N)**
network group entry, get or set	**getnetgrent(3C)**
network host entry, get or set	**gethostent(3N)**
Network Information Service client interface	**ypclnt(3C)**
Network Information Service, update user password in	**yppasswd(3N)**
network, scatter data to check the	**spray(3N)**
network station address string conversion routines	**net_aton(3C)**
new file, create	**creat(2)**
new process, create a	**fork(2)**
nextkey() – get next key in database (old single-data-base version)	**dbm(3X)**
NFS daemons	**nfssvc(2)**
nfssvc(): NFS daemon	**nfssvc(2)**
nice() – change priority of a process	**nice(2)**
NIS domain, get or set name of current	**getdomainname(2)**
nlappend() – append language ID to valid MPE file name	**nlappend(3X)**
nl_atof() – convert string to double-precision number	**strtod(3C)**
nlcollate() – compare strings; use MPE language-dependent collating sequence	**nlcollate(3X)**
nlconvclock() – check/convert time string to MPE internal format	**nlconvclock(3X)**
nlconvcustdate() – convert date string to MPE packed date format	**nlconvcustdate(3X)**
nlconvnum() – convert MPE native language formatted number to ASCII	**nlconvnum(3X)**
nl_ctime(), **nl_asctime()** – (obsolete; backwards compatibility only)	**ctime(3C)**
nlfindstr() – search for string in a string using MPE character set definition	**nlfindstr(3X)**
nlfmtcalendar() – format MPE packed date using localized format	**nlfmtcal(3X)**
nlfmtclock() – format MPE time of day using localized format	**nlfmtclock(3X)**
nlfmtcustdate() – format MPE packed date using custom date	**nlfmtcustdate(3X)**
nlfmtdate() – format MPE date and time in localized format	**nlfmtdate(3X)**
nlfmtlongcal() – format MPE packed date using long calendar format	**nlfmtlongcal(3X)**
nlfmtnum() – convert ASCII number to MPE language-specific formatted number	**nlfmtnum(3X)**
nl_fprintf(), **fprintf()** – print formatted output to a file	**printf(3S)**
nl_fscanf(), **fscanf()** – formatted read from named input stream file	**scanf(3S)**
nl_gcvt(), **gcvt()** – convert floating-point number to string array element	**ecvt(3C)**
nlgetlang() – return current user, data, or system default language	**nlgetlang(3X)**
nlinfo() – return MPE language-dependent information	**nlinfo(3X)**
nl_init(), **langinit()** (obsolete) – initialize the NLS environment of a program	**nl_init(3C)**
nl_isalnum() – NLS character class is alphanumeric	**nl_ctype(3C)**
nl_isalpha() – NLS character class is alpha	**nl_ctype(3C)**
nl_iscntrl() – NLS character class is a control character	**nl_ctype(3C)**
nl_isdigit() – NLS character class is a digit	**nl_ctype(3C)**
nl_isgraph() – NLS character class is a visible character	**nl_ctype(3C)**
nl_islower() – NLS character class is lowercase	**nl_ctype(3C)**
nl_isprint() – NLS character class is a printing character	**nl_ctype(3C)**
nl_ispunct() – NLS character class is punctuation	**nl_ctype(3C)**
nl_isspace() – NLS character class is whitespace	**nl_ctype(3C)**
nlist() – get entries from name list	**nlist(3C)**
nl_isupper() – NLS character class is uppercase	**nl_ctype(3C)**
nl_isxdigit() – NLS character class is a hexadecimal digit	**nl_ctype(3C)**
nljudge() – identify one- or multi-byte Asian character using MPE character table	**nljudge(3X)**
nlkeycompare() – compare character arrays using MPE collation table	**nlkeycompare(3X)**
nl_langinfo() – obtain NLS string form of local language variable	**nl_langinfo(3C)**
nlnumspec() – return number conversion/formatting information for MPE routines	**nlnumspec(3X)**
nl_printf(), **printf()** – print formatted output to *standard output*	**printf(3S)**
nlrepchar() – replace non-displayable string characters using MPE character set table	**nlrepchar(3X)**
NLS: classify characters for use with NLS	**nl_ctype(3C)**
NLS: get an NLS program message,	**catgets(3C)**

Description	Entry Name(Section)
NLS: get message from an NLS message catalogue	**catgetmsg(3C)**
NLS: initialize NLS environment of a program	**nl_init(3C)**
NLS: NLS information about native languages	**langinfo(3C)**
NLS: NLS information about native languages	**nl_langinfo(3C)**
NLS: open or close message catalog for reading	**catopen(3C)**
NLS: query numeric formatting conventions of current locale	**localeconv(3C)**
NLS: translate characters for use with NLS (obsolete – use`conv()`(3C))	**nl_conv(3C)**
`nl_scanf()`, `scanf()` – formatted read from *standard input* stream file	**scanf(3S)**
`nlscanmove()` – move, scan, case-shift strings using MPE character set table	**nlscanmove(3X)**
NLS message catalog, open or close for reading	**catopen(3C)**
`nl_sprintf()`, `sprintf()` – print formatted output to a string	**printf(3S)**
`nl_sscanf()`, `sscanf()` – formatted read from character string	**scanf(3S)**
`nl_strcmp()`, `nl_strncmp()` – compare strings using language-dependent collation	**string(3C)**
`nl_strtod()` – convert string to double-precision number	**strtod(3C)**
`nlsubstr()` – extract substring in string using MPE character set table	**nlsubstr(3X)**
`nlswitchbuf()` – convert string between phonetic and screen order using MPE table	**nlswitchbuf(3X)**
`nl_toupper()`, `nl_tolower()` – (obsolete) translate characters for use with NLS	**nl_conv(3C)**
`nltranslate()` – translate ASCII strings to EBCDIC using MPE conversion table	**nltranslate(3X)**
node from a binary search tree, delete a	**tsearch(3C)**
node name of current host, obtain NetIPC	**ipcgetnodename(2)**
node name of host CPU, set NetIPC	**ipcsetnodename(2)**
non-ASCII string collation	**nl_string(3C)**
non-local goto, save/restore stack environment for	**setjmp(3C)**
`ntohl()` – convert values between host and network byte order	**byteorder(3N)**
`ntohs()` – convert values between host and network byte order	**byteorder(3N)**
NULL, set callback to	**AtRemoveCallback(3X)**
number, convert string to double-precision	**strtod(3C)**
number, convert string to floating-point	**cvtnum(3C)**
number, convert string to long double-precision	**strtold(3C)**
number, convert wide character string to double-precision	**wcstod(3C)**
number, provide text describing NetIPC error	**ipcerrmsg(3N)**
numbers, generate uniformly distributed pseudo-random	**drand48(3C)**
number to string, convert long double floating-point	**ldcvt(3C)**
number to string or string array element, convert floating-point	**ecvt(3C)**
numeric formatting conventions of current locale, query	**localeconv(3C)**
object-code file, execute an	**exec(2)**
object into virtual memory, map	**mmap(2)**
obtain a destination descriptor	**ipclookup(2)**
odd parity on ATN commands, enable/disable	**hpib_parity_ctl(3I)**
open, access, or close a directory	**directory(3C)**
open a directory and associated *directory stream* for access	**directory(3C)**
open connection to specified audio server	**AOpenAudio(3X)**
`opendir()` – open a directory and associated *directory stream* for access	**directory(3C)**
open-file control	**fcntl(2)**
open file descriptor, duplicate an	**dup(2)**
open file descriptor to a specific slot, duplicate an	**dup2(2)**
`openlog()` – initialize system log file	**syslog(3C)**
`open()` – open file for reading or writing	**open(2)**
open or close NLS message catalog for reading	**catopen(3C)**
open or close pipe I/O to or from a process	**popen(3S)**
open or re-open a stream file; convert file to stream	**fopen(3S)**
operations, message control	**mstctl(2)**
operations on a stream file, get or reposition pointer for I/O	**fseek(3S)**
operations on NetIPC sockets, perform special	**ipccontrol(2)**
operations, semaphore control	**semctl(2)**
operations, semaphore	**semop(2)**
operations, set time limit for I/O	**io_timeout_ctl(3I)**
operations, shared memory control	**shmctl(2)**

Index
Volume 2

Description	Entry Name(Section)
optarg, **optind**, **opterr** – get option letter from argument vector	**getopt(3C)**
optimization package, CRT screen handling and	**curses(3X)**
option buffer, add argument and data to NetIPC	**addopt(3N)**
option buffer, initialize NetIPC	**initopt(3N)**
option buffer, obtain option code and data from NetIPC	**readopt(3N)**
option code and data from NetIPC option buffer, obtain	**readopt(3N)**
option letter from argument vector, get	**getopt(3C)**
options on sockets, get or set	**getsockopt(2)**
options, parse suboptions from a string	**getsubopt(3C)**
optoverhead() – return number of bytes needed by a NetIPC option	**optoverhead(3N)**
order of data, convert string	**strord(3C)**
ordinary file, make a directory, or a special or	**mknod(2)**
out-bound terminal line connection, establish an	**dial(3C)**
output, formatted, print to *standard output*, file, or string	**printf(3S)**
output, formatted with numbered arguments, print to a file or string	**printmsg(3C)**
output/input, buffered, standard stream file package	**stdio(3S)**
output of a varargs argument list, print formatted	**vprintf(3S)**
owner and group of a file, change	**chown(2)**
owner and/or group, change in access control list (ACL)	**chownacl(3C)**
package, standard interprocess communication	**stdipc(3C)**
packed decimal library, HP 3000-mode	**hppac(3X)**
paging behavior, advise system of process' expected	**madvise(2)**
paging/swapping, add a swap device for interleaved	**swapon(2)**
parallel channel, perform low-overhead I/O on a	**io_burst(3I)**
parallel poll on HP-IB bus, conduct	**hpib_ppoll(3I)**
parallel poll on HP-IB, control response to	**hpib_card_ppoll_resp(3I)**
parallel poll response, define interface	**hpib_ppoll_resp_ctl(3I)**
parallel poll value occurs, wait until a particular	**hpib_wait_on_ppoll(3I)**
parent process ID, get process, process group, or	**getpid(2)**
parity on ATN commands, enable/disable odd	**hpib_parity_ctl(3I)**
parse suboptions from a string	**getsubopt(3C)**
particular parallel poll value occurs, wait until a	**hpib_wait_on_ppoll(3I)**
Pascal and C execution startup routines	**crt0(3)**
password encryption function	**crypt(3C)**
password file entry, secure, write	**putspwent(3C)**
password file entry, write	**putpwent(3C)**
password file, get entry from	**getpwent(3C)**
password in Network Information Service, update user	**yppasswd(3N)**
password, read from terminal while suppressing echo	**getpass(3C)**
pathconf(), **fpathconf()** – get configurable pathname variables	**pathconf(2)**
path, map device ID to file	**devnm(3)**
path-name of current working directory, get	**getcwd(3C)**
path names, manipulate context-dependent file	**getcdf(3C)**
pathname variables, get configurable	**pathconf(2)**
patterns, match filename	**fnmatch(3C)**
pause() – suspend process until signal	**pause(2)**
pause the specified audio transaction	**APauseAudio(3X)**
pclose() – terminate pipe I/O to or from a process	**popen(3S)**
peer, get address of connected	**getpeername(2)**
pending signals, examine	**sigpending(2)**
performance data from remote kernel, get	**rstat(3N)**
perform I/O with an HP-IB channel from buffers	**hpib_io(3I)**
perform low-overhead I/O on an HP-IB/GPIO/parallel channel	**io_burst(3I)**
perform setup required for stream data conversion	**ASetupConversion(3X)**
perform special operations on NetIPC sockets	**ipccontrol(2)**
perform word expansions	**wordexp(3C)**
permissions mask for file creation, set and get	**umask(2)**
permissions (mode) of file, change access	**chmod(2)**

Description	Entry Name(Section)
per-process timer, allocate a	mktimer(3C)
per-process timer, free a	rmtimer(3C)
per-process timer, get value of a	gettimer(3C)
per-process timer, relatively arm a	reltimer(3C)
`perror()`, `errno()`, `sys_errlist()`, `sys_nerr()` – system error messages	perror(3C)
pfm_$bad_rls_order	pfm_$intro(3)
`pfm_$cleanup()` – establish a cleanup handler	pfm_$cleanup(3)
pfm_$cleanup_not_found	pfm_$intro(3)
pfm_$cleanup_rec	pfm_$intro(3)
pfm_$cleanup_set	pfm_$intro(3)
pfm_$cleanup_set_signalled	pfm_$intro(3)
`pfm_$enable()` – enable asynchronous faults	pfm_$enable(3)
`pfm_$enable_faults()` – enable asynchronous faults	pfm_$enable_faults(3)
pfm.h	pfm_$intro(3)
`pfm_$inhibit_faults()` – inhibit asynchronous faults; allow time-sliced task switching	pfm_$inhibit_faults(3)
`pfm_$inhibit()` – inhibit asynchronous faults	pfm_$inhibit(3)
`pfm_inhibit()` – pointer entry for conflicting online manual entries	pfm_inhibit(3)
`pfm_$init()` – initialize the process fault manager package	pfm_$init(3)
pfm_$init_signal_handlers	pfm_$intro(3)
pfm_$intro – fault management	pfm_$intro(3)
pfm_$invalid_cleanup_rec	pfm_$intro(3)
pfm_$no_space	pfm_$intro(3)
PFM package, initialize the	pfm_$init(3)
`pfm_$reset_cleanup()` – reset a cleanup handler	pfm_$reset_cleanup(3)
`pfm_$rls_cleanup()` – release a cleanup handler	pfm_$rls_cleanup(3)
`pfm_$signal()` – signal the calling process	pfm_$signal(3)
`pgm_$exit()` – exit a program	pgm_$exit(3)
pgm_$intro – program management	pgm_$intro(3)
`pipe()` – create an interprocess channel	pipe(2)
pipe I/O to or from a process, open or close	popen(3S)
play attributes to use when creating a new file, select	AChoosePlayAttributes(3X)
play operation, initiate an audio widget	AuInvokePlay(3X)
play specified sound bucket and return transaction ID	APlaySBucket(3X)
play widget, audio	AuPlayWidget(3X)
play widget, create an audio	AuCreatePlay(3X)
`plock()` – lock process, text, or data in memory	plock(2)
`pmap_getmaps()` – get list of RPC program-to-port mappings	rpc(3C)
`pmap_getport()` – get port number on which waits supporting service	rpc(3C)
`pmap_rmtcall()` – instruct portmapper to make an RPC call	rpc(3C)
`pmap_set()` – set [prognum,versnum,procnum]-to-port mapping	rpc(3C)
`pmap_unset()` – destroy [prognum,versnum,procnum]-to-port mapping	rpc(3C)
pointer array, sort a directory	scandir(3C)
pointer entry for conflicting online manual entries	pfm_inhibit(3)
pointer, file, move read/write	lseek(2)
pointer for binary search tree, get data	tsearch(3C)
pointer for I/O operations on a stream file, get or reposition	fseek(3S)
pointer, stream, map to file descriptor	fileno(3S)
pointer to login name in `utmp()`, get	getlogin(3C)
`poll` – monitor I/O conditions on multiple file descriptors	poll(2)
poll on HP-IB bus, conduct a serial	hpib_spoll(3I)
poll on HP-IB bus, conduct parallel	hpib_ppoll(3I)
poll on HP-IB, control response to parallel	hpib_card_ppoll_resp(3I)
poll, parallel, define interface response	hpib_ppoll_resp_ctl(3I)
poll value occurs, wait until a particular parallel	hpib_wait_on_ppoll(3I)
`popen()` – initiate pipe I/O to or from a process	popen(3S)
portable pfm_$ interface	pfm_$intro(3)
port, IP, bind socket to a privileged	bindresvport(3N)
port number, RPC, get	getrpcport(3N)

Index
Volume 2

Description	Entry Name(Section)
power, logarithm, exponential, square root, cube root functions	**exp(3M)**
`powf()`, `pow()` – power function	**exp(3M)**
`pow()`, `powf()` – power function	**exp(3M)**
preallocate fast disk storage	**prealloc(2)**
`prealloc()` – preallocate fast disk storage	**prealloc(2)**
prepare execution profile	**monitor(3C)**
presence of hardware capabilities, check for	**is_hw_present(3C)**
preset contents of memory area to specified character	**memory(3C)**
`printf()`, `nl_printf()` – print formatted output to *standard output*	**printf(3S)**
print formatted output of a varargs argument list	**vprintf(3S)**
print formatted output to *standard output*, file, or string	**printf(3S)**
print formatted output with numbered arguments to a file or string	**printmsg(3C)**
`printmsg()` – print formatted output with numbered arguments to *standard output*	**printmsg(3C)**
priority, get process	**getpriority(2)**
priority of a process, change	**nice(2)**
priority, set process	**setpriority(2)**
privileged IP port, bind socket to a	**bindresvport(3N)**
procedure calls, remote, library routines for	**rpc(3C)**
process 16-bit characters, tools to	**nl_tools_16(3C)**
process accounting, enable or disable	**acct(2)**
process and child process times, get	**times(2)**
process, calling, set or clear auditing on	**setaudproc(2)**
process, change priority of a	**nice(2)**
process context for context-dependent file search, return	**getcontext(2)**
process, create a new	**fork(2)**
process environment, clear the	**clearenv(3C)**
process' expected paging behavior, advise system of	**madvise(2)**
process fault management	**pfm_$intro(3)**
process fault manager package, initialize the	**pfm_$init(3)**
process, get audit ID (`aid()`) for current	**getaudid(2)**
process, get audit process flag for calling	**getaudproc(2)**
process group ID, create session and set	**setsid(2)**
process group ID, foreground, get	**tcgetpgrp(3C)**
process group ID, foreground, set	**tcsetpgrp(3C)**
process group ID for job control, set	**setpgid(2)**
process interval timer, set or get value of	**getitimer(2)**
process, lock into memory after allocating data and stack space	**datalock(3C)**
process, open or close pipe I/O to or from a	**popen(3S)**
process or a group of processes, send a signal to a	**kill(2)**
processor type, determine	**sysconf(2)**
process priority, get	**getpriority(2)**
process priority, set	**setpriority(2)**
process, process group, or parent process ID, get	**getpid(2)**
process, request connection to another	**ipcconnect(2)**
process's alarm clock, set	**alarm(2)**
process, self-auditing, write audit record for	**audwrite(2)**
process, set audit ID (`aid()`) for current	**setaudid(2)**
process, signal the calling	**pfm_$signal(3)**
process, spawn new (use `fork()` instead)	**vfork(2)**
process, suspend or resume auditing on current	**audswitch(2)**
process, suspend until signal	**pause(2)**
process, terminate	**exit(2)**
process, text, or data, lock in memory	**plock(2)**
process to stop or terminate, wait for child or traced	**wait(2)**
process trace	**ptrace(2)**
profile, execution time	**profil(2)**
profile of execution, prepare	**monitor(3C)**
`profil()` – execution time profile	**profil(2)**

Description	Entry Name(Section)
program assertion, verify	**assert(3X)**
program, exit a	**pgm_$exit(3)**
program, get or set the locale of a	**setlocale(3C)**
program, initialize the NLS environment of a	**nl_init(3C)**
program management	**pgm_$intro(3)**
program message, get an NLS	**catgets(3C)**
program regions, first locations beyond allocated	**end(3C)**
program termination, register a function to be called at	**atexit(2)**
protections, modify memory mapping access	**mprotect(2)**
protocol entry, get or set	**getprotoent(3N)**
provide semaphores and record locking on files	**lockf(2)**
provide text describing NetIPC error number	**ipcerrmsg(3N)**
pseudo-random numbers, generate uniformly distributed	**drand48(3C)**
`ptrace()` – process trace	**ptrace(2)**
`ptsname` – get the name of a slave pty	**ptsname(3C)**
pty, get the name of a slave	**ptsname(3C)**
purge and/or flush the cache	**cachectl(3C)**
push character back into input stream	**ungetc(3S)**
push event onto head of audio event queue	**APutBackEvent(3X)**
push wide character back into input stream	**ungetwc(3C)**
put a string on a stream	**puts(3S)**
`putc()`, `fputc()` – put character on a stream	**putc(3S)**
put character or word on a stream	**putc(3S)**
`putchar()` – put character on stream *standard output*	**putc(3S)**
`putenv()` – change or add value to environment	**putenv(3C)**
`putpwent()` – write password file entry	**putpwent(3C)**
`putspwent()` – write secure password file entry	**putspwent(3C)**
`puts()` – write null-terminated string to stream `stdout()`	**puts(3S)**
`_pututline()` – update or create entry in a `utmp()` file	**getut(3C)**
`pututline()` – update or create entry in a `utmp()` file	**getut(3C)**
`putwc()`, `fputwc()` – put wide character on a stream	**putwc(3C)**
`putwchar()` – put wide character on stream *standard output*	**putwc(3C)**
put wide character on a stream	**putwc(3C)**
put word or character on a stream	**putc(3S)**
`putw()` – put word (integer) on a stream	**putc(3S)**
`qsort()` – quicker sort	**qsort(3C)**
query numeric formatting conventions of current locale	**localeconv(3C)**
quicker sort	**qsort(3C)**
`quotactl()` – manipulate disk quotas	**quotactl(2)**
quotas, manipulate disk	**quotactl(2)**
raise a software signal	**ssignal(3C)**
`raise()` – raise a software signal	**kill(2)**
`rand()` – generate successive random numbers	**rand(3C)**
random-number generator, simple	**rand(3C)**
rate of I/O data transfer, inform system of required minimum	**io_speed_ctl(3I)**
`rcmd()`: return a stream to a remote command	**rcmd(3N)**
read audio data into sound bucket	**ARecordAData(3X)**
`readdir()` – get pointer to current entry in open directory	**directory(3C)**
read from stream file or character string with formatted input conversion	**scanf(3S)**
reading or writing, open file for	**open(2)**
read, I/O, determine how last terminated	**io_get_term_reason(3I)**
`readlink()` – read value of a symbolic link	**readlink(2)**
`readopt()` – obtain option code and data from NetIPC option buffer	**readopt(3N)**
read or change real-time priority	**rtprio(2)**
read password from terminal while suppressing echo	**getpass(3C)**
`read()` – read contiguous data from a file	**read(2)**
read termination character on special file, set up I/O	**io_eol_ctl(3I)**
read value of a symbolic link	**readlink(2)**

Description	Entry Name(Section)
readv() – read non-contiguous data from a file	**read(2)**
read/write file pointer, move	**lseek(2)**
real, effective, and/or saved user or group IDs, set	**setresuid(2)**
realloc() – change size of allocated memory block	**malloc(3C)**
real or effective user or group ID, get	**getuid(2)**
reboot() – boot the system	**reboot(2)**
receipt of a signal, define what to do upon	**signal(2)**
receive connection request on a call socket	**ipcrecvcn(2)**
receive data on NetIPC virtual circuit connection	**ipcrecv(2)**
receive message from a socket	**recv(2)**
receive message from message queue	**msgop(2)**
record, audit, write for self-auditing process	**audwrite(2)**
record locking and semaphores on files, provide	**lockf(2)**
record operation, initiate an audio widget	**AuInvokeRecord(3X)**
record widget, audio	**AuRecordWidget(3X)**
record widget, create an audio	**AuCreateRecord(3X)**
record widget, save sound bucket data created by	**AuSaveFile(3X)**
recursively descend a directory hierarchy	**ftw(3C)**
recvfrom() – receive message from a socket	**recv(2)**
recvmsg() – receive message from a socket	**recv(2)**
recv() – receive message from a socket	**recv(2)**
regcmp() – compile a regular expression	**regcmp(3X)**
regcomp() – regular expression matching routines	**regcomp(3C)**
regerror() – regular expression matching routines	**regcomp(3C)**
regexec() – regular expression matching routines	**regcomp(3C)**
regex() – execute a regular expression against a string	**regcmp(3X)**
regfree() – regular expression matching routines	**regcomp(3C)**
region, initialize semaphore in mapped file or anonymous memory	**msem_init(2)**
region, remove semaphore in mapped file or anonymous	**msem_remove(2)**
regions, first locations beyond allocated program	**end(3C)**
region, unmap a mapped	**munmap(2)**
register a function to be called at program termination	**atexit(2)**
registerrpc() – register procedure with RPC service package	**rpc(3C)**
regular expression compile and match routines	**regexp(3X)**
regular expression, compile or execute against a string	**regcmp(3X)**
regular expression matching routines	**regcomp(3C)**
relatively arm a per-process timer	**reltimer(3C)**
release a cleanup handler	**pfm_$rls_cleanup(3)**
release a descriptor	**ipcshutdown(2)**
release blocked signals and atomically wait for interrupt	**sigpause(2)**
release server from exclusive use by this connection	**AUngrabServer(3X)**
reltimer – relatively arm a per-process timer	**reltimer(3C)**
remainder, ceiling, floor, absolute value, round-to-nearest functions	**floor(3M)**
remainder, integer division and	**div(3C)**
remainder manipulations	**ieee(3M)**
remexportent() – access exported file system information	**exportent(3N)**
remote command, return a stream to	**rcmd(3N)**
remote command, return stream to a	**rexec(3N)**
Remote Enable line on HP-IB, control the	**hpib_ren_ctl(3I)**
remote kernel, get performance data from	**rstat(3N)**
remotely mounted file systems, keep track of	**mount(3N)**
remote machines, return information about users on	**rnusers(3N)**
remote machines, write to specified	**rwall(3N)**
remote node, get file handle for file on	**getfh(2)**
remote procedure calls, library routines for	**rpc(3C)**
remove a directory file	**rmdir(2)**
remove directory entry; delete file or directory name	**unlink(2)**
remove() – remove a file	**remove(3C)**

Description	Entry Name(Section)
remove semaphore in mapped file or anonymous region	**msem_remove(2)**
rename() – change the name of a file	**rename(2)**
re-open or open a stream file; convert file to stream	**fopen(3S)**
replace default error handler with specified handler	**ASetErrorHandler(3X)**
replace default I/O error handler with specified handler	**ASetIOErrorHandler(3X)**
report CPU time used	**clock(3C)**
reposition or get pointer for I/O operations on a stream file	**fseek(3S)**
representation, library routines for external data	**xdr(3C)**
request connection to another process	**ipcconnect(2)**
requested status condition becomes true, wait until the	**hpib_status_wait(3I)**
request on a call socket, receive connection	**ipcrecvcn(2)**
request report of specified audio events	**ASelectInput(3X)**
required minimum I/O data transfer rate, inform system of	**io_speed_ctl(3I)**
reset a cleanup handler	**pfm_$reset_cleanup(3)**
reset an I/O interface	**io_reset(3I)**
res_init() – resolver routines	**resolver(3N)**
res_mkquery() – resolver routines	**resolver(3N)**
resolver routines	**resolver(3N)**
resource consumption limit, get or set system	**getrlimit(2)**
response, define interface parallel poll	**hpib_ppoll_resp_ctl(3I)**
response to parallel poll on HP-IB, control	**hpib_card_ppoll_resp(3I)**
res_query() – resolver routines	**resolver(3N)**
res_search() – resolver routines	**resolver(3N)**
res_send() – resolver routines	**resolver(3N)**
restore or save file position indicator for a stream	**fgetpos(3S)**
restore/save stack environment for non-local goto	**setjmp(3C)**
restore signal action	**sigset(2V)**
resume or suspend auditing on current process	**audswitch(2)**
resume specified audio transaction	**AResumeAudio(3X)**
resvport(): return a stream to a remote command	**rcmd(3N)**
return a stream to a remote command	**rcmd(3N)**
return but do not dequeue first event in audio event queue	**APeekEvent(3X)**
return character back into input stream	**ungetc(3S)**
return gain matrix of basic play device	**ASimplePlayer(3X)**
return gain matrix of basic recording device	**ASimpleRecorder(3X)**
return information about users on remote machines	**rnusers(3N)**
return integer absolute value	**abs(3C)**
return list of sampling rates supported by audio controller	**ASamplingRates(3X)**
return number of bytes needed by a NetIPC option	**optoverhead(3N)**
return number of data formats supported by audio controller	**ANumDataFormats(3X)**
return number of events on audio event queue	**AQLength(3X)**
return number of sampling rates supported by audio controller	**ANumSamplingRates(3X)**
return process context for context-dependent file search	**getcontext(2)**
return status of HP-IB interface	**hpib_bus_status(3I)**
return stream to a remote command	**rexec(3N)**
return the size in bytes of converted data	**ACalculateLength(3X)**
return wide character back into input stream	**ungetwc(3C)**
rewinddir() – reset position of named *directory stream* to beginning of directory	**directory(3C)**
rewind legal user shells file	**getusershell(3C)**
rewind() – set position of next I/O operation on stream file	**fseek(3S)**
rewrite an existing file	**creat(2)**
rexec(): return stream to a remote command	**rexec(3N)**
rights to a file, get a user's effective access	**getaccess(2)**
right triangle, hypotenuse of a	**hypot(3M)**
rindex() – BSD portability string routine	**string(3C)**
rint(), **fabsf()**, **fabs()**, **floor()**, **ceil()**, **fmod()**, **fmodf()** – round-to-nearest, absolute value, floor, ceiling, remainder functions	**floor(3M)**
rmdir() – remove a directory file	**rmdir(2)**

Index
Volume 2

Description	Entry Name(Section)
`rmtimer` – free a per-process timer	**rmtimer(3C)**
`rnusers()`: return information about users on remote machines	**rnusers(3N)**
root directory, change	**chroot(2)**
rounding mode (floating-point), examine and set	**fpgetround(3M)**
round-to-nearest, absolute value, floor, ceiling, remainder functions	**floor(3M)**
routine for sorted tables, binary search	**bsearch(3C)**
routines, CRT screen handling and optimization	**curses(3X)**
routines, emulate `/etc/termcap` access	**termcap(3X)**
routines for external data representation, library	**xdr(3C)**
routines, Internet address manipulation	**inet(3N)**
routines, network station address string conversion	**net_aton(3C)**
routines, resolver	**resolver(3N)**
`rpc()`: library routines for remote procedure calls	**rpc(3C)**
`rpc_createerr()` – global variable reason why client creation failed	**rpc(3C)**
RPC entry, get	**getrpcent(3C)**
RPC port number, get	**getrpcport(3N)**
`rstat()` – get performance data from remote kernel	**rstat(3N)**
RTE/MPE-style message catalog support	**catread(3C)**
`rtprio()` – change or read real-time priority	**rtprio(2)**
`ruserok()`: return a stream to a remote command	**rcmd(3N)**
`rusers()`: return information about users on remote machines	**rnusers(3N)**
`rwall()`: write to specified remote machines	**rwall(3N)**
saved, real, and/or effective user or group IDs, set	**setresuid(2)**
save or restore file position indicator for a stream	**fgetpos(3S)**
save/restore stack environment for non-local goto	**setjmp(3C)**
save sound bucket data created by record widget	**AuSaveFile(3X)**
`sbrk()` – increase data segment space allocation	**brk(2)**
`scalb()`, `logb()` – exponent manipulations	**ieee(3M)**
scan a directory	**scandir(3C)**
`scandir()` – scan a directory	**scandir(3C)**
`scanf()`, `nl_scanf()` – formatted read from *standard input* stream file	**scanf(3S)**
scatter data to check the network	**spray(3N)**
screen handling and optimization package, CRT	**curses(3X)**
search, context-dependent file, return process context for	**getcontext(2)**
search environment list for value of specified variable name	**getenv(3C)**
search routine, binary, for sorted tables	**bsearch(3C)**
search table for entry; optional update if missing	**lsearch(3C)**
search tables, hash, manage	**hsearch(3C)**
search tree, manage a binary	**tsearch(3C)**
`secof2()`, `SECof2()` – test for valid second byte in 16-bit character	**nl_tools_16(3C)**
secure password file entry, write	**putspwent(3C)**
secure password file, get entry from	**getspwent(3C)**
`seekdir()` – set position of next `readdir()` operation on named *directory stream*	**directory(3C)**
seek; move read/write file pointer	**lseek(2)**
segment, get shared memory	**shmget(2)**
select attributes to use when creating a new file	**AChooseAFileAttributes(3X)**
select attributes to use when creating a new file	**AChoosePlayAttributes(3X)**
select attributes to use with an existing file or a stream	**AChooseSourceAttributes(3X)**
`select()` – synchronous I/O multiplexing	**select(2)**
self-auditing process, write audit record for	**audwrite(2)**
semaphore control operations	**semctl(2)**
semaphore in mapped file or anonymous memory region, initialize	**msem_init(2)**
semaphore in mapped file or anonymous region, remove	**msem_remove(2)**
semaphore, lock a	**msem_lock(2)**
semaphore operations	**semop(2)**
semaphores and record locking on files, provide	**lockf(2)**
semaphores, get set of	**semget(2)**
semaphore, unlock a	**msem_unlock(2)**

Description	Entry Name(Section)
`semctl()` – semaphore control operations	**semctl(2)**
`semget()` – get set of semaphores	**semget(2)**
`semop()` – semaphore operations	**semop(2)**
send a signal to a process or a group of processes	**kill(2)**
send command bytes over HP-IB	**hpib_send_cmnd(3I)**
send data on a virtual circuit connection	**ipcsend(2)**
send message to a socket	**send(2)**
send message to message queue	**msgop(2)**
`sendmsg()` – send message to a socket	**send(2)**
`send()` – send message to a socket	**send(2)**
`sendto()` – send message to a socket	**send(2)**
separate floating-point into mantissa and exponent	**frexp(3C)**
serial poll on HP-IB bus, conduct a	**hpib_spoll(3I)**
service entry, get or set	**getservent(3N)**
session, create and set process group ID	**setsid(2)**
set: file creation (permissions) mask, set and get	**umask(2)**
set: file size limits and break value, get or set	**ulimit(2)**
set: process priority	**setpriority(2)**
set: system clock date and time	**gettimeofday(2)**
set access control list (ACL) information	**setacl(2)**
`setaclentry()` – add, modify, or delete access control list entry	**setaclentry(3C)**
`setacl()`, `fsetacl()` – set access control list (ACL) information	**setacl(2)**
set and/or get signal stack context	**sigstack(2)**
set a process's alarm clock	**alarm(2)**
`setaudid()` – set audit ID (`aid()`) for current process	**setaudid(2)**
set audit ID (`aid()`) for current process	**setaudid(2)**
`setaudproc()` – set or clear auditing on calling process	**setaudproc(2)**
`setbuf()`, `setvbuf()` – assign buffering to a stream file	**setbuf(3S)**
set callback to NULL	**AtRemoveCallback(3X)**
`setccent()` – rewind cluster configuration pointer to beginning of file	**getccent(3C)**
`setclock` – set value of system-wide clock	**setclock(3C)**
set close-down mode on specified connection	**ASetCloseDownMode(3X)**
set contents of memory area to specified character	**memory(3C)**
set current ignorable signals mask	**sigsetmask(2)**
`setdomainname()` – set name of current NIS domain	**getdomainname(2)**
`setevent()` – set current events and system calls to be audited	**setevent(2)**
`setexportent()` – access exported file system information	**exportent(3N)**
set foreground process group ID	**tcsetpgrp(3C)**
`setfsent()` – open and rewind file system descriptor file	**getfsent(3X)**
`setgid()` – set group ID	**setuid(2)**
`setgrent()` – rewind pointer to first entry in `group()` file	**getgrent(3C)**
set group access list	**setgroups(2)**
`setgroups()` – set group access list	**setgroups(2)**
`sethostent()` – get network host entry	**gethostent(3N)**
`sethostname()` – set name of host cpu	**sethostname(2)**
`setitimer()` – set value of process interval timer	**getitimer(2)**
`_setjmp()` – save stack environment for non-local goto	**setjmp(3C)**
`setkey()` – generate hashing encryption	**crypt(3C)**
`setlocale()` – set the locale of a program	**setlocale(3C)**
`setlogmask()` – set system log file priority mask	**syslog(3C)**
`setmntent()` – open a file system description file	**getmntent(3X)**
set name of current NIS domain	**getdomainname(2)**
set name of host cpu	**sethostname(2)**
`setnetent()`: get network entry	**getnetent(3N)**
`setnetgrent()` – get network group entry	**getnetgrent(3C)**
set NetIPC node name of host CPU	**ipcsetnodename(2)**
set network entry	**getnetent(3N)**
set network group entry	**getnetgrent(3C)**

Index
Volume 2

Description	Entry Name(Section)
set network host entry	**gethostent(3N)**
set of semaphores, get	**semget(2)**
set or clear auditing on calling process	**setaudproc(2)**
set or get audit files	**audctl(2)**
set or get tty baud rate	**cfspeed(3C)**
set or update file access and modification times	**utime(2)**
`setpgid()`, – set process group ID for job control	**setpgid(2)**
`setpgrp2()`: set process group ID	**setpgid(2)**
`setpgrp()` – create session and set process group ID	**setsid(2)**
set play volume or record gain of specified transaction	**ASetGain(3X)**
`setpriority` – set process priority	**setpriority(2)**
set process group ID, create session and	**setsid(2)**
set process group ID for job control	**setpgid(2)**
set protocol entry	**getprotoent(3N)**
`setprotoent()`: –get protocol entry	**getprotoent(3N)**
`setpwent()` – rewind pointer to beginning of password file	**getpwent(3C)**
set real, effective, and/or saved user or group IDs	**setresuid(2)**
`setresgid()` – set real, effective, and/or saved group IDs	**setresuid(2)**
`setresuid()` – set real, effective, and/or saved user IDs	**setresuid(2)**
`setrlimit()` – get system resource consumption limit	**getrlimit(2)**
`setservent()`: get service entry	**getservent(3N)**
`setsid()`, `setpgrp()` – create session and set process group ID	**setsid(2)**
`setsockopt()` – set options on sockets	**getsockopt(2)**
`setspwent()` – rewind pointer to beginning of secure password file	**getspwent(3C)**
set system or monitor audio channel gain	**ASetSystemChannelGain(3X)**
set system play volume	**ASetSystemPlayGain(3X)**
set system record gain	**ASetSystemRecordGain(3X)**
set system resource consumption limit	**getrlimit(2)**
set the locale of a program	**setlocale(3C)**
set time and date	**stime(2)**
set time limit for I/O operations	**io_timeout_ctl(3I)**
`settimeofday()` – set system clock date and time	**gettimeofday(2)**
set transaction channel gain	**ASetChannelGain(3X)**
set tty device operating parameters	**tcattribute(3C)**
`setuid()` – set user ID	**setuid(2)**
set up I/O read termination character on special file	**io_eol_ctl(3I)**
set user or group ID	**setuid(2)**
`setusershell()` – rewind legal user shells file	**getusershell(3C)**
`setutent()` – reset input stream to beginning of `utmp()` file	**getut(3C)**
set value of process interval timer	**getitimer(2)**
set value of system-wide clock	**setclock(3C)**
set width (in bits) of data path	**io_width_ctl(3I)**
`sgetl()` – retrieve a 4-byte long integer from memory	**sputl(3X)**
shared library, load or unload	**shl_load(3X)**
shared library, look up symbol in	**shl_load(3X)**
shared memory and data segment, attach or detach	**shmop(2)**
shared memory control operations	**shmctl(2)**
shared memory segment, get	**shmget(2)**
shell command, issue a	**system(3S)**
shells, get legal user	**getusershell(3C)**
`shl_findsym()` – look up symbol in shared library	**shl_load(3X)**
`shl_get()` – get information about shared library	**shl_load(3X)**
`shl_load()` – load shared library	**shl_load(3X)**
`shl_unload()` – unload shared library	**shl_load(3X)**
`shmat()` – attach shared memory to data segment	**shmop(2)**
`shmctl()` – shared memory control operations	**shmctl(2)**
`shmdt()` – detach shared memory from data segment	**shmop(2)**
`shmget()` – get shared memory segment	**shmget(2)**

Description	Entry Name(Section)
shut down a socket	**shutdown(2)**
`shutdown()` – shut down a socket	**shutdown(2)**
`sigaction()` – examine and change signal action	**sigaction(2)**
`sigaddset()` – initialize, manipulate, and test signal sets	**sigsetops(3C)**
`sigblock()` – block signals	**sigblock(2)**
`sigdelset()` – initialize, manipulate, and test signal sets	**sigsetops(3C)**
`sigemptyset()` – initialize, manipulate, and test signal sets	**sigsetops(3C)**
`sigfillset()` – initialize, manipulate, and test signal sets	**sigsetops(3C)**
`sighold()`, `sigrelse()`, `sigignore()`, `sigpause()`, `sigset()` – signal management	**sigset(2V)**
`sigignore()`, `sigpause()`, `sigset()`, `sighold()`, `sigrelse()` – signal management	**sigset(2V)**
`sigismember()` – initialize, manipulate, and test signal sets	**sigsetops(3C)**
`signal()` – 4.2 BSD-compatible `signal()` system call	**bsdproc(2)**
signal action, examine and change	**sigaction(2)**
signal, define what to do upon receipt of a	**signal(2)**
signal exceptions, managing	**pfm_$intro(3)**
signal facilities, software	**sigvector(2)**
signal, hold upon receipt	**sigset(2V)**
signal, ignore	**sigset(2V)**
signal management (`sigset()`, `sighold()`, `sigrelse()`, `sigignore()`, `sigpause()`)	**sigset(2V)**
signal, raise a software	**kill(2)**
signal, raise a software	**ssignal(3C)**
signal, restore action	**sigset(2V)**
signals, blocked, examine and change	**sigprocmask(2)**
signals, block	**sigblock(2)**
signal, select method of handling	**sigset(2V)**
signal, send to a process or a group of processes	**kill(2)**
signal sets, initialize, manipulate, and test	**sigsetops(3C)**
signals, examine pending	**sigpending(2)**
signals mask, set current ignorable	**sigsetmask(2)**
`signal()` – specify what to do upon receipt of a signal	**signal(2)**
signals, release blocked and atomically wait for interrupt	**sigpause(2)**
signal stack context, set and/or get	**sigstack(2)**
signal stack space, define, delete, or get amount of	**sigspace(2)**
signal, suspend calling process until received	**sigset(2V)**
signal, suspend process until	**pause(2)**
`signal()` system call, 4.2 BSD-compatible	**bsdproc(2)**
signal the calling process	**pfm_$signal(3)**
signal, wait for a	**sigsuspend(2)**
`signgam()`, `gamma()`, `lgamma()` – log gamma function	**gamma(3M)**
`sigpause()` – atomically release blocked signals and wait for interrupt	**sigpause(2)**
`sigpause()`, `sigset()`, `sighold()`, `sigrelse()`, `sigignore()` – signal management	**sigset(2V)**
`sigpending()` – examine pending signals	**sigpending(2)**
`sigprocmask()` – examine and change blocked signals	**sigprocmask(2)**
`sigrelse()`, `sigignore()`, `sigpause()`, `sigset()`, `sighold()` – signal management	**sigset(2V)**
`sigsetmask()` – set current ignorable signals mask	**sigsetmask(2)**
`sigset()`, `sighold()`, `sigrelse()`, `sigignore()`, `sigpause()` – signal management	**sigset(2V)**
`sigspace()` – define or delete additional signal stack space	**sigspace(2)**
`sigstack()` – set and/or get signal stack context	**sigstack(2)**
`sigsuspend()` – wait for a signal	**sigsuspend(2)**
`sigvec()` – 4.2 BSD-compatible `sigvec()` system call	**bsdproc(2)**
`sigvec()` system call, 4.2 BSD-compatible	**bsdproc(2)**
`sigvector()` – software signal facilities	**sigvector(2)**
`sindf()` – trigonometric sine function (float, degrees)	**trigd(3M)**
`sind()` – trigonometric sine function (degrees)	**trigd(3M)**
sine trigonometric function (degrees)	**trigd(3M)**
sine trigonometric function	**trig(3M)**
`sinf()` – trigonometric sine function (float)	**trig(3M)**
`sinhf()`, `sinh()` – hyperbolic sine functions	**sinh(3M)**

Index
Volume 2

Description	Entry Name(Section)
`sinh()`, `sinhf()` – hyperbolic sine functions	**sinh(3M)**
`sin()` – trigonometric sine function	**trig(3M)**
sixteen-bit characters, tools to process	**nl_tools_16(3C)**
slave pty, get the name of a	**ptsname(3C)**
`sleep()` – suspend execution for interval	**sleep(3C)**
slot in the `utmp()` file of the current user, find	**ttyslot(3C)**
socket, accept connection on a	**accept(2)**
socket address, get	**getsockname(2)**
socket, bind address to a	**bind(2)**
socket, bind to a privileged IP port	**bindresvport(3N)**
`socket()` – create an endpoint for communication	**socket(2)**
socket, initiate connection on a	**connect(2)**
socket, listen for connections on a	**listen(2)**
socket or destination call socket, associate name with call	**ipcname(2)**
socket or destination call socket, delete name associated with a call	**ipcnamerase(2)**
socket or VC socket, determine status of call	**ipcselect(2)**
`socketpair()` – create a pair of connected sockets	**socketpair(2)**
socket, receive connection request on a call	**ipcrecvcn(2)**
socket, receive message from a	**recv(2)**
sockets, create a pair of connected	**socketpair(2)**
socket, send message to a	**send(2)**
sockets, get or set options on	**getsockopt(2)**
socket, shut down a	**shutdown(2)**
sockets, perform special operations on NetIPC	**ipccontrol(2)**
software signal facilities	**sigvector(2)**
software signal, raise a	**kill(2)**
software signal, raise a	**ssignal(3C)**
sort a directory pointer array	**scandir(3C)**
sorted tables, binary search routine for	**bsearch(3C)**
sort, quicker	**qsort(3C)**
sound bucket data created by record widget, save	**AuSaveFile(3X)**
space allocation, change data segment	**brk(2)**
space for signal stack, define, delete, or get amount of	**sigspace(2)**
space, stack and data, allocate then lock process into memory	**datalock(3C)**
spawn new process (use `fork()` instead)	**vfork(2)**
special file, control character device	**ioctl(2)**
special file, FIFO, make a	**mkfifo(3C)**
special file, set up I/O read termination character on	**io_eol_ctl(3I)**
special operations on NetIPC sockets, perform	**ipccontrol(2)**
special or ordinary file, make a directory, or a	**mknod(2)**
specified file, get file attributes of	**AGetAFileAttributes(3X)**
specified remote machines, write to	**rwall(3N)**
specify I/O read termination character on special file	**io_eol_ctl(3I)**
specify what to do upon receipt of a signal	**signal(2)**
speed, inform system of required minimum I/O transfer	**io_speed_ctl(3I)**
split floating-point into mantissa and exponent	**frexp(3C)**
`spray`: scatter data to check the network	**spray(3N)**
`sprintf()`, `nl_sprintf()` – print formatted output to a string	**printf(3S)**
`sprintmsg()` – print formatted output with numbered arguments to a string	**printmsg(3C)**
`sputl()` – place a 4-byte long integer in memory	**sputl(3X)**
`sqrt()`, `cbrt()`, `sqrtf()`, `cbrtf()` – square root, cube root functions	**exp(3M)**
`sqrtf()`, `sqrt()`, `cbrt()`, `cbrtf()` – square root, cube root functions	**exp(3M)**
square root, power, logarithm, exponential, cube root functions	**exp(3M)**
`srand48()`, `seed48()`, `lcong48()` – initialize pseudo-random number generator	**drand48(3C)**
`srand()` – reset random-number generator to random starting point	**rand(3C)**
SRQ line on HP-IB, allow interface to enable	**hpib_rqst_srvce(3I)**
`sscanf()`, `nl_sscanf()` – formatted read from character string	**scanf(3S)**
`ssignal()` – raise a software signal and perform an action	**ssignal(3C)**

Description	Entry Name(Section)
stack and data space, allocate then lock process into memory	**datalock(3C)**
stack context, signal, set and/or get	**sigstack(2)**
stack environment, save/restore for non-local goto	**setjmp(3C)**
stack space for signals, define, delete, or get amount of	**sigspace(2)**
standard buffered input/output stream file package	**stdio(3S)**
standard input stream, input string from a	**gets(3S)**
standard input stream, input wide string from a	**getws(3C)**
standard interprocess communication package	**stdipc(3C)**
start or halt auditing system	**audctl(2)**
state with its state on disk, synchronize a file's in-core	**fsync(2)**
statfsdev(), **fstatfsdev()** – get file system statistics	**statfsdev(3C)**
statfs(), **fstatfs()** – get file system statistics	**statfs(2)**
station address string conversion routines, network	**net_aton(3C)**
statistics, get file system	**statfs(2)**
statistics, get file system	**statfsdev(3C)**
statistics, get mounted file system	**ustat(2)**
stat(), **lstat()**, **fstat()** – get file status	**stat(2)**
status code, return an error message for a	**error_$c_text(3)**
status code, return subsystem, module, and error texts for a	**error_$c_get_text(3)**
status condition becomes true, wait until the requested	**hpib_status_wait(3I)**
status, get file	**stat(2)**
status inquiries, stream	**ferror(3S)**
status lines of GPIO card, return	**gpio_get_status(3I)**
status of call socket or VC socket, determine	**ipcselect(2)**
status of HP-IB interface, return	**hpib_bus_status(3I)**
std_$call	**pfm_$intro(3)**
stdio() – standard buffered input/output stream file package	**stdio(3S)**
step() – regular expression string comparison routine	**regexp(3X)**
stime() – set time and date	**stime(2)**
stop activity on specified HP-IB	**hpib_abort(3I)**
stop or terminate, wait for child or traced process to	**wait(2)**
stop specified audio transaction	**AStopAudio(3X)**
storage, preallocate fast disk	**prealloc(2)**
store() – store data under a key (old single-data-base version)	**dbm(3X)**
strcat(), **strncat()** – append string 2 to string 1	**string(3C)**
strchr(), **strrchr()** – get pointer to character in string	**string(3C)**
strcmp16(), **strncmp16()** – non-ASCII 16-bit character string collation	**nl_string(3C)**
strcmp8(), **strncmp8()** – non-ASCII 8-bit character string collation	**nl_string(3C)**
strcmp(), **strncmp()** – compare two strings	**string(3C)**
strcoll() – process string of text tokens	**string(3C)**
strcpy(), **strncpy()** – copy string 2 to string 1	**string(3C)**
strcspn(), **strspn()** – find length of matching substrings	**string(3C)**
stream, close a	**fclose(3S)**
stream file, assign buffering to a	**setbuf(3S)**
stream file, buffered binary input/output to a	**fread(3S)**
stream file, get character or data word from a	**getc(3S)**
stream file, get or reposition pointer for I/O operations on a	**fseek(3S)**
stream file, get wide character from a	**getwc(3C)**
stream file, open or re-open; convert file to stream	**fopen(3S)**
stream file or character string, read from with formatted input conversion	**scanf(3S)**
stream file package, standard buffered input/output	**stdio(3S)**
stream, flush buffer with or without closing	**fclose(3S)**
stream, input string from a *standard input*	**gets(3S)**
stream, input wide string from a *standard input*	**getws(3C)**
stream pointer, map to file descriptor	**fileno(3S)**
stream, push character back into input	**ungetc(3S)**
stream, push wide character back into input	**ungetwc(3C)**
stream, put wide character on a	**putwc(3C)**

Index
Volume 2

Description	Entry Name(Section)
stream, put word or character on a	**putc(3S)**
stream, return to a remote command	**rcmd(3N)**
stream, return to a remote command	**rexec(3N)**
stream, save or restore file position indicator for a	**fgetpos(3S)**
stream status inquiries	**ferror(3S)**
strerror() – system error messages	**perror(3C)**
strftime() – convert date and time to string	**strftime(3C)**
string collation, non-ASCII	**nl_string(3C)**
string conversion routines, network station address	**net_aton(3C)**
string, convert between long integer and base-64 ASCII	**a64l(3C)**
string, convert date and time to	**ctime(3C)**
string, convert date and time to	**strftime(3C)**
string, convert date and time to wide-character	**wcsftime(3C)**
string, convert long double floating-point number to	**ldcvt(3C)**
string, convert long integer to	**ltostr(3C)**
string, convert to access control list (ACL) structure	**strtoacl(3C)**
string, convert to floating-point number	**cvtnum(3C)**
string, convert to long double-precision number	**strtold(3C)**
string data order, convert	**strord(3C)**
string form, convert access control list (ACL) structure to	**acltostr(3C)**
string from a *standard input* stream, input	**gets(3S)**
string operations, character	**string(3C)**
string operations, wide character	**wcstring(3C)**
string or string array element, convert floating-point number to	**ecvt(3C)**
string, parse suboptions from a	**getsubopt(3C)**
strings and characters conversions, multibyte	**multibyte(3C)**
strings, concatenate two	**string(3C)**
string to double-precision number, convert	**strtod(3C)**
string to long integer, convert	**strtol(3C)**
string-valued configuration values, get	**confstr(3C)**
strlen() – determine length of a string	**string(3C)**
strord() – convert string data order	**strord(3C)**
strpbrk() – find occurrence of character from string 2 in string 1	**string(3C)**
strrstr() – process string of text tokens	**string(3C)**
strspn(), **strcspn()** – find length of matching substrings	**string(3C)**
strstr() – process string of text tokens	**string(3C)**
strtoacl() – convert exact string form to access control list (ACL) structure	**strtoacl(3C)**
strtoaclpatt() – convert pattern string form to access control list (ACL) structure	**strtoacl(3C)**
strtod() – convert string to double-precision number	**strtod(3C)**
strtok() – process string of text tokens	**string(3C)**
strtol() – convert string to long integer	**strtol(3C)**
strtold() – convert string to long double-precision number	**strtold(3C)**
strxfrm() – process string of text tokens	**string(3C)**
stty(), **gtty()** – control terminal device (Version 6 compatibility only)	**stty(2)**
suboptions, parse from a string	**getsubopt(3C)**
subroutines and libraries, introduction to	**intro(3)**
subroutines, database (new multiple database version)	**ndbm(3X)**
subroutines, database (old version – see also **ndbm(3X)**)	**dbm(3X)**
subsystem, module, and error texts for a status code, return	**error_$c_get_text(3)**
super-block, update	**sync(2)**
support, RTE/MPE-style message catalog	**catread(3C)**
suppress echo while reading password from terminal	**getpass(3C)**
suspend execution for interval	**sleep(3C)**
suspend or resume auditing on current process	**audswitch(2)**
suspend process until signal	**pause(2)**
svc_destroy() – destroy RPC service transport handle	**rpc(3C)**
svcerr_auth() – refuse service because of authentication error	**rpc(3C)**
svcerr_decode() – service cannot decode its parameters	**rpc(3C)**

Description	Entry Name(Section)
`svcerr_noproc()` – service hasn't implemented the desired procedure	**rpc(3C)**
`svcerr_noprog()` – program not registered with RPC package	**rpc(3C)**
`svcerr_progvers()` – version not registered with RPC package	**rpc(3C)**
`svcerr_systemerr()` – service detected system error	**rpc(3C)**
`svcerr_weakauth()` – refuse service due to insufficient authentication	**rpc(3C)**
`svcfd_create()` – create RPC service from existing socket	**rpc(3C)**
`svc_fdset()` – global array with RPC service file descriptor mask	**rpc(3C)**
`svc_freeargs()` – free data allocated by RPC/XDR	**rpc(3C)**
`svc_getargs()` – decode arguments in RPC request	**rpc(3C)**
`svc_getcaller()` – get procedure caller's network address	**rpc(3C)**
`svc_getreqset()` – return when all associated sockets have been serviced	**rpc(3C)**
`svcraw_create()` – create toy RPC service transport for testing	**rpc(3C)**
`svc_run()` – wait for RPC requests to arrive and call appropriate service	**rpc(3C)**
`svc_sendreply()` – send back results of remote procedure call	**rpc(3C)**
`svctcp_create()` – create RPC service based on TCP transport	**rpc(3C)**
`svcudp_create()` – create RPC service based on UDP transport	**rpc(3C)**
`svc_unregister()` – remove mapping of [prognum,versnum] to dispatch routines	**rpc(3C)**
`swab()` – swap bytes	**swab(3C)**
swap bytes	**swab(3C)**
swap device for interleaved paging/swapping, add a	**swapon(2)**
`swapon()` – add a swap device for interleaved paging/swapping	**swapon(2)**
swapping, file system	**swapon(2)**
swapping/paging, add a swap device for interleaved	**swapon(2)**
symbolic link, read value of	**readlink(2)**
symbolic link to a file, make a	**symlink(2)**
symbol, look up in shared library	**shl_load(3X)**
`symlink()` – make symbolic link to a file	**symlink(2)**
synchronize a file's in-core state with its state on disk	**fsync(2)**
synchronize a mapped file	**msync(2)**
synchronous I/O multiplexing	**select(2)**
`sync()`, `lsync()` – update super-block	**sync(2)**
`sysconf` – get configurable system variables	**sysconf(2)**
`sys_errlist` – system error messages	**perror(3C)**
`syslog()` – write message onto system log file	**syslog(3C)**
`sys_nerr` – system error messages	**perror(3C)**
system, boot	**reboot(2)**
system calls and events currently being audited, get	**getevent(2)**
system calls and events to be audited	**setevent(2)**
system calls, BSD-4.2-compatible `kill()`, `sigvec()`, and `signal()`	**bsdproc(2)**
system-calls error indicator	**errno(2)**
system calls, introduction to	**intro(2)**
system clock date and time, get or set	**gettimeofday(2)**
system error messages	**perror(3C)**
`system()` – issue a shell command	**system(3S)**
system log, control	**syslog(3C)**
system of process' expected paging behavior, advise	**madvise(2)**
system or monitor audio channel gain, get	**AGetSystemChannelGain(3X)**
system or monitor audio channel gain, set	**ASetSystemChannelGain(3X)**
system resource consumption limit, get or set	**getrlimit(2)**
system variables, get configurable	**sysconf(2)**
system-wide clock, get current value of	**getclock(3C)**
system-wide clock, set value of	**setclock(3C)**
table, eliminate duplicate entries in a	**lsearch(3C)**
table, linear search for entry; optional update if missing	**lsearch(3C)**
tables, binary search routine for sorted	**bsearch(3C)**
tables, hash search, manage	**hsearch(3C)**
`tandf()` – trigonometric tangent function (float, degrees)	**trigd(3M)**
`tand()` – trigonometric tangent function (degrees)	**trigd(3M)**

Index
Volume 2

Description	Entry Name(Section)
`tanf()` – trigonometric tangent function (float)	**trig(3M)**
tangent trigonometric function (degrees)	**trigd(3M)**
tangent trigonometric function	**trig(3M)**
`tanhf()`, `tanh()` – hyperbolic tangent functions	**sinh(3M)**
`tanh()` – inverse hyperbolic tangent function	**asinh(3M)**
`tanh()`, `tanhf()` – hyperbolic tangent functions	**sinh(3M)**
`tan()` – trigonometric tangent function	**trig(3M)**
`tcdrain()`: tty line control function	**tccontrol(3C)**
`tcflow()`: tty line control function	**tccontrol(3C)**
`tcflush()`: tty line control function	**tccontrol(3C)**
`tcgetattr()`: get tty device operating parameters	**tcattribute(3C)**
`tcgetpgrp()`: get foreground process group ID	**tcgetpgrp(3C)**
`tcsendbreak()`: tty line control function	**tccontrol(3C)**
`tcsetattr()`: set tty device operating parameters	**tcattribute(3C)**
`tcsetpgrp()`: get foreground process group ID	**tcsetpgrp(3C)**
`tdelete()` – delete a node from a binary search tree	**tsearch(3C)**
`telldir()` – get current location of named *directory stream*	**directory(3C)**
`tempnam()` – create a name for a temporary file	**tmpnam(3S)**
temporary file, create a name for	**tmpnam(3S)**
temporary file, create a	**tmpfile(3S)**
temporary (unique) file name, make a	**mktemp(3C)**
`termcap()` access routines, emulate `/etc/`	**termcap(3X)**
terminal block-mode library interface	**blmode(3C)**
terminal, find name of	**ttyname(3C)**
terminal, generate file name of controlling	**ctermid(3S)**
terminal I/O, block-mode library interface for	**blmode(3C)**
terminal line connection, establish an out-bound	**dial(3C)**
terminal, read password from while suppressing echo	**getpass(3C)**
terminate a per-process timer	**rmtimer(3C)**
terminated, determine how last I/O read	**io_get_term_reason(3I)**
terminate, wait for child or traced process to stop or	**wait(2)**
termination character on special file, set up I/O read	**io_eol_ctl(3I)**
termination, register a function to be called at program	**atexit(2)**
test contents of memory area	**memory(3C)**
test for INFINITY	**isinf(3M)**
test for NaN	**isnan(3M)**
test, initialize, and manipulate signal sets	**sigsetops(3C)**
text database operations, error	**error_$intro(3)**
text, data, or process, lock in memory	**plock(2)**
text describing NetIPC error number, provide	**ipcerrmsg(3N)**
texts for a status code, return subsystem, module, and error	**error_$c_get_text(3)**
`tfind()` – get data pointer for binary search tree	**tsearch(3C)**
`tgetent()` – get compiled terminfo data base entry into buffer	**termcap(3X)**
`tgetflag()` – get availability of compiled boolean terminal capability	**termcap(3X)**
`tgetnum()` – get numeric value of compiled terminal capability	**termcap(3X)**
`tgetstr()` – get string value of compiled terminal capability	**termcap(3X)**
`tgoto()` – get compiled terminal cursor addressing string	**termcap(3X)**
three-byte integers and long integers, convert between	**l3tol(3C)**
time and date, convert to string	**ctime(3C)**
time and date, convert to string	**strftime(3C)**
time and date, convert to wide-character string	**wcsftime(3C)**
time and date, get more precisely (Version 7 compatibility only)	**ftime(2)**
time and date, get or set system clock	**gettimeofday(2)**
time and date, set	**stime(2)**
time, convert user format date and	**getdate(3C)**
time, get	**time(2)**
`time()` – get time	**time(2)**
time limit for I/O operations, set	**io_timeout_ctl(3I)**

Description	Entry Name(Section)
timeout limit for I/O operations, set	**io_timeout_ctl(3I)**
time profile, execution	**profil(2)**
timer, allocate a per-process	**mktimer(3C)**
timer, free a per-process	**rmtimer(3C)**
timer, get value of a per-process	**gettimer(3C)**
timer, relatively arm a per-process	**reltimer(3C)**
timer, set or get value of process interval	**getitimer(2)**
times, file access and modification, set or update	**utime(2)**
`times()` – get process and child process times	**times(2)**
times, get process and child process	**times(2)**
time used, report CPU	**clock(3C)**
`timezone()` – difference between UCT and local timezone	**ctime(3C)**
`tmpfile()` – create a temporary file	**tmpfile(3S)**
`tmpnam()` – create a name for a temporary file	**tmpnam(3S)**
`toascii()` – translate characters to 7-bit ASCII	**conv(3C)**
`tolower()`, `_tolower()` – translate characters to lowercase	**conv(3C)**
toolkit, add callback procedure for audio	**AtAddCallback(3X)**
tools to process 16-bit characters	**nl_tools_16(3C)**
`toupper()`, `_toupper()`, – translate characters to uppercase	**conv(3C)**
`towlower()` – translate wide characters to lowercase	**wconv(3C)**
`towupper()` – translate wide characters to uppercase	**wconv(3C)**
`tputs()` – decode terminal string padding information	**termcap(3X)**
traced process to stop or terminate, wait for child or	**wait(2)**
trace, process	**ptrace(2)**
transaction channel gain, get	**AGetChannelGain(3X)**
transaction channel gain, set	**ASetChannelGain(3X)**
transfer speed, inform system of required minimum I/O	**io_speed_ctl(3I)**
translate character code to another code set	**iconv(3C)**
translate characters for use with NLS (obsolete – use **conv(3C)**)	**nl_conv(3C)**
translate characters to uppercase, lowercase, or 7-bit ASCII	**conv(3C)**
translate wide characters to uppercase or lowercase	**wconv(3C)**
traverse a binary search tree	**tsearch(3C)**
traverse (walk) a file tree	**ftw(3C)**
tree, manage a binary search	**tsearch(3C)**
tree, walk a file	**ftw(3C)**
triangle, right, hypotenuse of a	**hypot(3M)**
trigonometric functions (degrees)	**trigd(3M)**
trigonometric functions, hyperbolic	**sinh(3M)**
trigonometric functions, inverse hyperbolic	**asinh(3M)**
trigonometric functions	**trig(3M)**
true, wait until the requested status condition becomes	**hpib_status_wait(3I)**
truncate an existing file to zero for rewriting	**creat(2)**
`truncate()`, `ftruncate()` – truncate a file to a specified length	**truncate(2)**
`tsearch()` – build and access a binary search tree	**tsearch(3C)**
tty baud rate, set or get	**cfspeed(3C)**
tty device operating parameters, get or set	**tcattribute(3C)**
tty line control functions	**tccontrol(3C)**
`ttyname()`, `isatty()` – find name of a terminal	**ttyname(3C)**
`ttyslot()` – find the slot in the `utmp()` file of the current user	**ttyslot(3C)**
`twalk()` – traverse a binary search tree	**tsearch(3C)**
type, classify characters according to	**ctype(3C)**
type, classify characters according to	**wctype(3C)**
type of NLS characters, classify	**nl_ctype(3C)**
`tzname()` – name of local timezone	**ctime(3C)**
`tzset()` – initialize `timezone()`, `daylight()`, and `tzname()` using TZ variable	**ctime(3C)**
UID, get name from (obsolete)	**getpw(3C)**
`ulimit()` – get or set file size limits and break value	**ulimit(2)**
`ultoa()`; convert unsigned long integer to ASCII decimal	**ltostr(3C)**

Index
Volume 2

Description	Entry Name(Section)
`ultostr()`; convert unsigned long integer to string	**ltostr(3C)**
`umask()` – set and get file creation (permissions) mask	**umask(2)**
`umount()` – unmount a file system	**umount(2)**
`uname()` – get name and version of current HP-UX system	**uname(2)**
underflow mode (floating-point), examine and set	**fpgetround(3M)**
`undial()`, `dial()` – establish an out-bound terminal line connection	**dial(3C)**
`ungetc()` – push character back into input stream	**ungetc(3S)**
`ungetwc()` – push wide character back into input stream	**ungetwc(3C)**
unique (usually temporary) file name, make a	**mktemp(3C)**
`unlink` – remove directory entry; delete file	**unlink(2)**
unload shared library	**shl_load(3X)**
unlock a semaphore	**msem_unlock(2)**
unlock or lock an I/O interface	**io_lock(3I)**
unmap a mapped region	**munmap(2)**
unmount a file system	**umount(2)**
unsigned long integer to string, convert	**ltostr(3C)**
update a file's header	**AUpdateDataLength(3X)**
update or set file access and modification times	**utime(2)**
update super-block	**sync(2)**
update table if entry missing after search	**lsearch(3C)**
update user password in Network Information Service	**yppasswd(3N)**
uppercase, translate characters to	**conv(3C)**
uppercase, translate wide characters to	**wconv(3C)**
user, current, find the slot in the `utmp()` file of the	**ttyslot(3C)**
user format date and time, convert	**getdate(3C)**
user ID, get real or effective	**getuid(2)**
user ID, set	**setuid(2)**
user login name, get character-string representation of	**cuserid(3S)**
user login name, obtain	**logname(3C)**
user or group IDs, set real, effective, and/or saved	**setresuid(2)**
user password in Network Information Service, update	**yppasswd(3N)**
user's effective access rights to a file, get a	**getaccess(2)**
user shells, get legal	**getusershell(3C)**
users on remote machines, return information about	**rnusers(3N)**
`ustat()` – get mounted file system statistics	**ustat(2)**
`utime()` – set or update file access and modification times	**utime(2)**
`utmp()` file of the current user, find the slot in the	**ttyslot(3C)**
`utmp()`, get pointer to login name in	**getlogin(3C)**
`utmpname()` – change name of `utmp()` file being examined	**getut(3C)**
`utmp()` or `wtmp()` file, access	**getut(3C)**
value, change or add to environment	**putenv(3C)**
value, get or set file size limits and break	**ulimit(2)**
value occurs, wait until a particular parallel poll	**hpib_wait_on_ppoll(3I)**
value of a per-process timer, get	**gettimer(3C)**
value of process interval timer, set or get	**getitimer(2)**
value of system-wide clock, get current	**getclock(3C)**
value of system-wide clock, set	**setclock(3C)**
value, return integer absolute	**abs(3C)**
values, convert between host and network byte order	**byteorder(3N)**
values, get string-valued configuration	**confstr(3C)**
varargs argument, formatted input conversion to a	**vscanf(3S)**
varargs argument list, print formatted output of a	**vprintf(3S)**
variable, environment, search environment list for value of	**getenv(3C)**
variables, configurable pathname, get	**pathconf(2)**
variables, system, get configurable	**sysconf(2)**
VC socket, determine status of	**ipcselect(2)**
vector, get option letter from argument	**getopt(3C)**
verify program assertion	**assert(3X)**

Description	Entry Name(Section)
version and name of current HP-UX system, get	**uname(2)**
vfork() – spawn new process (use **fork()** instead)	**vfork(2)**
vfprintf() – print formatted output of a varargs argument list	**vprintf(3S)**
vfscanf() – formatted input conversion to a varargs argument	**vscanf(3S)**
vfsmount() – mount a file system	**vfsmount(2)**
virtual circuit connection, establish or receive data on NetIPC	**ipcrecv(2)**
virtual circuit connection, send data on a	**ipcsend(2)**
Virtual Circuit socket, determine status of	**ipcselect(2)**
virtual memory, map object into	**mmap(2)**
vprintf(), **vfprintf()**, **vsprintf()** – print formatted output of a varargs argument list	**vprintf(3S)**
vscanf() – formatted input conversion to a varargs argument	**vscanf(3S)**
vsprintf() – print formatted output of a varargs argument list	**vprintf(3S)**
vsscanf() – formatted input conversion to a varargs argument	**vscanf(3S)**
wait for a signal	**sigsuspend(2)**
wait for interrupt, atomically release blocked signals and	**sigpause(2)**
wait until a particular parallel poll value occurs	**hpib_wait_on_ppoll(3I)**
wait until the requested status condition becomes true	**hpib_status_wait(3I)**
wait(), **waitpid()**, **wait3()** – wait for child or traced process to stop or terminate	**wait(2)**
walk a file tree	**ftw(3C)**
WCHARADV(), – put character in memory and advance pointer	**nl_tools_16(3C)**
WCHAR(), – put 8- or 16-bit character in memory	**nl_tools_16(3C)**
wcscat, **wcsncat** – append wide string 2 to wide string 1	**wcstring(3C)**
wcschr, **wcsrchr** – get pointer to wide character in wide string	**wcstring(3C)**
wcscmp, **wcsncmp** – compare two wide strings	**wcstring(3C)**
wcscoll – process wide string of text tokens	**wcstring(3C)**
wcscpy, **wcsncpy** – copy wide string 2 to wide string 1	**wcstring(3C)**
wcscspn, **wcsspn** – find length of matching wide substrings	**wcstring(3C)**
wcsftime() – convert date and time to wide-character string	**wcsftime(3C)**
wcslen – determine length of a wide string	**wcstring(3C)**
wcspbrk – find occurrence of wide character from wide string 2 in wide string 1	**wcstring(3C)**
wcstod() – convert wide character string to double-precision number	**wcstod(3C)**
wcstok – process wide string of text tokens	**wcstring(3C)**
wcstol() – convert wide character string to long integer	**wcstol(3C)**
wcstombs() – multibyte characters and strings conversions	**multibyte(3C)**
wcswcs – process wide string of text tokens	**wcstring(3C)**
wctomb() – multibyte characters and strings conversions	**multibyte(3C)**
wide character back into input stream, push	**ungetwc(3C)**
wide character from a stream file, get	**getwc(3C)**
wide character, put on a stream	**putwc(3C)**
wide characters, translate to uppercase or lowercase	**wconv(3C)**
wide-character string, convert date and time to	**wcsftime(3C)**
wide character string operations	**wcstring(3C)**
wide character string to double-precision number, convert	**wcstod(3C)**
wide character string to long integer, convert	**wcstol(3C)**
wide string from a *standard input* stream, input	**getws(3C)**
wide strings, concatenate two	**wcstring(3C)**
widget, audio play	**AuPlayWidget(3X)**
widget, audio record	**AuRecordWidget(3X)**
widget, create an audio play	**AuCreatePlay(3X)**
widget, create an audio record	**AuCreateRecord(3X)**
widget play operation, initiate an audio	**AuInvokePlay(3X)**
widget record operation, initiate an audio	**AuInvokeRecord(3X)**
widget, save sound bucket data created by record	**AuSaveFile(3X)**
width (in bits) of data path, set	**io_width_ctl(3I)**
word expansions, perform	**wordexp(3C)**
wordexp – perform word expansions	**wordexp(3C)**
wordfree – perform word expansions	**wordexp(3C)**
word from a stream file, get character or data	**getc(3S)**

Index
Volume 2

Description	Entry Name(Section)
word or character, put on a stream	**putc(3S)**
working directory, change	**chdir(2)**
working directory, get path-name of current	**getcwd(3C)**
write a header for an audio file	**AWriteAHeader(3X)**
write a null-terminated string on a stream	**puts(3S)**
write a null-terminated wide string on a stream	**fputws(3C)**
write audit record for self-auditing process	**audwrite(2)**
write password file entry	**putpwent(3C)**
write/read file pointer, move	**lseek(2)**
write secure password file entry	**putspwent(3C)**
write sound bucket data into file with data conversion	**ASaveSBucket(3X)**
write to specified remote machines	**rwall(3N)**
`writev()` – write non-contiguous data to a file	**write(2)**
`write()` – write contiguous data to a file	**write(2)**
writing or reading, open file for	**open(2)**
`wtmp()` or `utmp()` file, access	**getut(3C)**
`xdr()`: library routines for external data representation	**xdr(3C)**
`xdr_accepted_reply()` – generate RPC-style replies without using RPC package	**rpc(3C)**
`xdr_array()` – translate arrays to/from external representation	**xdr(3C)**
`xdr_authunix_parms()` – generate UNIX credentials without using RPC package	**rpc(3C)**
`xdr_bool()` – translate Booleans to/from external representation	**xdr(3C)**
`xdr_bytes()` – translate counted byte strings to/from external representation	**xdr(3C)**
`xdr_callhdr()` – generate RPC-style headers without using RPC package	**rpc(3C)**
`xdr_callmsg()` – generate RPC-style messages without using RPC package	**rpc(3C)**
`xdr_char()` – translate characters to/from external representation	**xdr(3C)**
`xdr_destroy()` – destroy XDR stream and free associated memory	**xdr(3C)**
`xdr_double()` – translate double precision to/from external representation	**xdr(3C)**
`xdr_enum()` – translate enumerations to/from external representation	**xdr(3C)**
`xdr_float()` – translate floating point to/from external representation	**xdr(3C)**
`xdr_free()` – free the memory allocated to create XDR data structures	**xdr(3C)**
`xdr_getpos()` – return current position in XDR stream	**xdr(3C)**
`xdr_inline()` – invoke the in-line routines associated with XDR stream	**xdr(3C)**
`xdr_int()` – translate integers to/from external representation	**xdr(3C)**
`xdr_long()` – translate long integers to/from external representation	**xdr(3C)**
`xdrmem_create()` – initialize an XDR stream	**xdr(3C)**
`xdr_opaque_auth()` – describe RPC messages externally	**rpc(3C)**
`xdr_opaque()` – translate fixed-size opaque data to/from external representation	**xdr(3C)**
`xdr_pmap()` – describe parameters for portmap procedures externally	**rpc(3C)**
`xdr_pmaplist()` – describe a list of port mappings externally	**rpc(3C)**
`xdr_pointer()` – similar to `xdr_reference()` but different	**xdr(3C)**
`xdrrec_create()` – initialize an XDR stream with record boundaries	**xdr(3C)**
`xdrrec_endofrecord()` – mark XDR record stream with an end-of-record	**xdr(3C)**
`xdrrec_eof()` – mark XDR record stream with an end-of-file	**xdr(3C)**
`xdrrec_skiprecord()` – skip remaining record in XDR record stream	**xdr(3C)**
`xdr_reference()` – chase pointers within structures	**xdr(3C)**
`xdr_rejected_reply()` – generate RPC-style rejections without using RPC package	**rpc(3C)**
`xdr_replymsg()` – generate RPC-style replies without using RPC package	**rpc(3C)**
`xdr_setpos()` – change current position in XDR stream	**xdr(3C)**
`xdr_short()` – translate short integers to/from external representation	**xdr(3C)**
`xdrstdio_create()` – initialize XDR stream as standard I/O FILE stream	**xdr(3C)**
`xdr_string()` – translate null-terminated strings to/from external representation	**xdr(3C)**
`xdr_u_char()` – translate unsigned characters to/from external representation	**xdr(3C)**
`xdr_u_int()` – translate unsigned integers to/from external representation	**xdr(3C)**
`xdr_u_long()` – translate unsigned long integers to/from external representation	**xdr(3C)**
`xdr_union()` – translate descriminated unions to/from external representation	**xdr(3C)**
`xdr_u_short()` – translate unsigned short integers to/from external representation	**xdr(3C)**
`xdr_vector()` – translate fixed-length arrays to/from external representation	**xdr(3C)**
`xdr_void()` – always return one (1)	**xdr(3C)**

Printed in USA E0892

Manual Part No.
B2355-90033

Manufacturing
Part No.
B2355-90033

B2355-90033