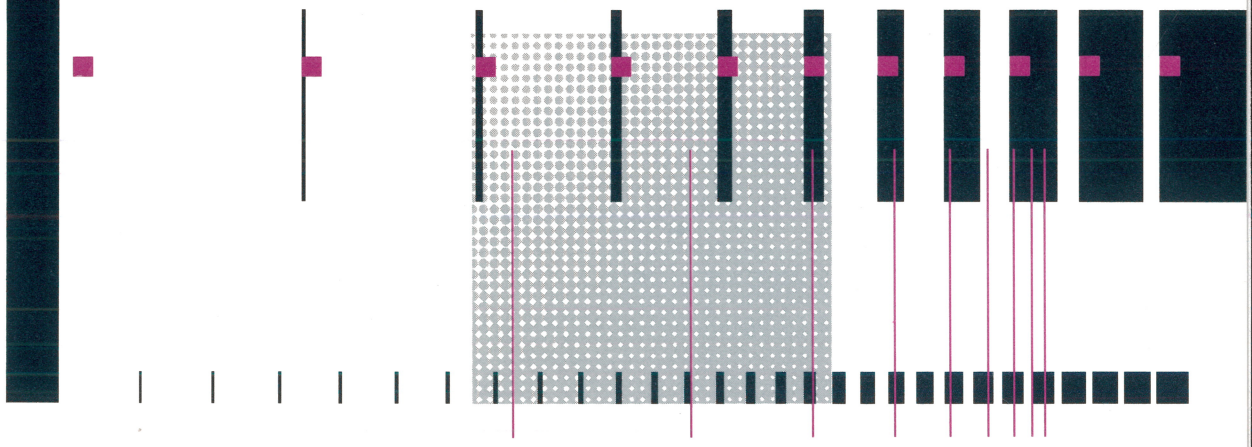
RISC/os (UMIPS)
Programmer's Reference Manual
Volume I (System V)

Order Number 3203DOC



mips

The power of RISC is in the system.

RISC/os (UMIPS)
Programmer's Reference Manual
Volume I (System V)

Order Number 3203DOC

March 1989

Your comments on our products and publications are welcome. A postage–paid form is provided for this purpose on the last page of this manual.

Mfg. Part Number 02–00133–002/84–00136–002

© 1988, 1989 MIPS Computer Systems, Inc. All Rights Reserved.

RISCompiler and RISC/os are Trademarks of MIPS Computer Systems, Inc.
UNIX is a Trademark of AT&T.
Ethernet is a Trademark of XEROX.

MIPS Computer Systems, Inc.
930 Arques Ave.
Sunnyvale, CA 94086

Customer Service Telephone Numbers:

California:	(800)	992–MIPS
All other states:	(800)	443–MIPS
International:	(415)	330–7966

Mfg. Part Number 84–00136/02–00133–001

TABLE OF CONTENTS

2. System Calls

3. Library Subroutines

4. Special Files and Hardware Support

5. Miscellaneous

PERMUTED INDEX

perror, errno, sys_errlist, sys_nerr : system error messages . . . . . perror(3c)
get information about the running system machine_info : . . . . . machine_info(3c)
mount : mount a file system . . . . . mount(2)
nfsmount : mount an NFS file system . . . . . nfsmount(2)
umount : unmount a file system . . . . . umount(2)
system : execute a UNIX command . . . . . system(3f)
command system : execute a UNIX . . . . . system(3f)
volume fs) file system : format of s51k system . . . . . fs(4)
system : issue a shell command . . . . . system(3s)
information table system : system configuration . . . . . system(4)
stat : data returned by stat system call . . . . . stat(5)
syscall : indirect system call . . . . . syscall(2)
intro : introduction to system calls and error numbers . . . . . intro(2)
table system : system configuration information . . . . . system(4)
types : primitive system data types . . . . . types(5)
endmntent, hasmntopt : get file system descriptor file entry . . . . . getmntent(3)
errno, sys_errlist, sys_nerr : system error messages perror, ... perror(3c)
perror, gerror, ierrno : get system error messages . . . . . perror(3f)
entry dirent : file system independent directory . . . . . dirent(4)
statfs, fstatfs : get file system information . . . . . statfs(2)
uname : get general system information . . . . . uname(2)
getpagesize : get system page size . . . . . getpagesize(2)
ustat : get file system statistics . . . . . ustat(2)
etc mtab : mounted file system table . . . . . mtab(4)
rmtab : remotely mounted file system table . . . . . rmtab(4)
ctime, ltime, gmtime : return system time time, ... time(3f)
timezone : set default system time zone . . . . . timezone(4)
sysfs : get file system type information . . . . . sysfs(2)
fs) file system : format of s51k system volume . . . . . fs(4)
exports : NFS file systems being exported . . . . . exports(4)
and checklist : list of file systems processed by fsck(s51k . . . . . checklist(4)
bsearch : binary search a sorted table . . . . . bsearch(3c)
: runtime procedure table _procedure_string_table . . . . . end(3)
: runtime procedure table _procedure_string_table . . . . . end(3)
etc mtab : mounted file system table . . . . . mtab(4)
: access routine for the symbol table ranhash, ranlookup . . . . . ranhash(3x)
: access routine for the symbol table ranhash, ranlookup . . . . . ranhash(3x)
: remotely mounted file system table rmtab . . . . . rmtab(4)
interface to the MIPS symbol table a binary read write . . . . . stio(3)
interface to the MIPS symbol table a binary read write . . . . . stio(3)
: routines to print the symbol table stprint . . . . . stprint(3)
:syms : MIPS symbol table . . . . . syms(4)
system configuration information table system : . . . . . system(4)
: compute the index of a symbol table entry of a common object . . . . . ldtbindex(3x)
ldtbread : read an indexed symbol table entry of a common object . . . . . ldtbread(3x)
ldtbread : read an indexed symbol table entry of a common object . . . . . ldtbread(3x)
provide a compilation unit symbol table interface : routines that . . . . . stcu(3)
provide a compilation unit symbol table interface : routines that . . . . . stcu(3)
ldtbseek : seek to the symbol table of a common object file . . . . . ldtbseek(3x)
ldtbseek : seek to the symbol table of a common object file . . . . . ldtbseek(3x)
hdestroy : manage hash search tables hsearch, hcreate, ... hsearch(3c)
trigonometric sin, cos, tan, asin, acos, atan, atan2 : . . . . . sin(3m)
trigonometric sin, cos, tan, asin, acos, atan, atan2 : . . . . . sin(3m)
intrinsic function tan, dtan : Fortran tangent . . . . . tan(3f)
tan, dtan : Fortran tangent intrinsic function . . . . . tan(3f)
tanh, dtanh : Fortran hyperbolic tangent intrinsic function . . . . . tanh(3f)
sinh, cosh, tanh : hyperbolic functions . . . . . sinh(3m)
sinh, cosh, tanh : hyperbolic functions . . . . . sinh(3m)
tangent intrinsic function tanh, dtanh : Fortran hyperbolic . . . . . tanh(3f)
tar : tape archive file format . . . . . tar(4)
tpd : format of MIPS boot tape directories . . . . . tpd(4)
tar : tape archive file format . . . . . tar(4)
search trees tsearch, tfind, tdelete, twalk : manage binary . . . . . tsearch(3c)
directory) opendir, readdir, telldir, seekdir, rewinddir, ... directory(3x)
temporary file tmpnam, tempnam : create a name for a . . . . . tmpnam(3s)
tmpfile : create a temporary file . . . . . tmpfile(3s)
tempnam : create a name for a temporary file tmpnam, ... tmpnam(3s)
terminals term : conventional names for . . . . . term(5)
file(term : format of compiled term . . . . . term(4)
term : format of compiled term file(. . . . . term(4)
ctermid : generate file name for terminal . . . . . ctermid(3s)
ttyname, isatty : find name of a terminal . . . . . ttyname(3c)
terminfo : terminal capability data base . . . . . terminfo(4)
dial : establish an out-going terminal line connection . . . . . dial(3c)
ttynam, isatty : find name of a terminal port . . . . . ttynam(3f)
optimization package curses : terminal screen handling and . . . . . curses(3x)

NAME

accept – accept a connection on a socket

SYNOPSIS

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

ns = accept(s, addr, addrlen)
int ns, s;
struct sockaddr *addr;
int *addrlen;
```

DESCRIPTION

The argument *s* is a socket that has been created with *socket*(2), bound to an address with *bind*(2), and is listening for connections after a *listen*(2). *accept* extracts the first connection on the queue of pending connections, creates a new socket with the same properties of *s* and allocates a new file descriptor, *ns*, for the socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, *accept* blocks the caller until a connection is present. 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*, may not be used to accept more connections. The original socket *s* remains open.

The argument *addr* is a result parameter that is filled in with the address of the connecting entity, as known to the communications layer. The exact format of the *addr* parameter is determined by the domain in which the communication is occurring. The *addrlen* is a value-result parameter; it should initially contain the amount of space pointed to by *addr*; on return it will contain the actual length (in bytes) of the address returned. This call is used with connection-based socket types, currently with SOCK_STREAM.

It is possible to *select*(2) a socket for the purposes of doing an *accept* by selecting it for read.

RETURN VALUE

The call returns −1 on error. If it succeeds, it returns a non-negative integer that is a descriptor for the accepted socket.

ERRORS

The *accept* will fail if:

[EBADF]	The descriptor is invalid.
[ENOTSOCK]	The descriptor references a file, not a socket.
[EOPNOTSUPP]	The referenced socket is not of type SOCK_STREAM.
[EFAULT]	The *addr* parameter is not in a writable part of the user address space.
[EWOULDBLOCK]	The socket is marked non-blocking and no connections are present to be accepted.

SEE ALSO

bind(2), connect(2), listen(2), select(2), socket(2)

NOTE

The primitives documented on this manual page are system calls, but unlike most system calls they are not resolved by libc. To compile and link a program that makes these calls, follow the procedures for section (3B) routines as described in *intro(3)*.

ORIGIN

4.3 BSD

NAME

access – determine accessibility of a file

SYNOPSIS

int access (path, amode)
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 in place of the effective user ID and the real group ID in place of the effective group ID. The bit pattern contained in *amode* is constructed as follows:

04	read
02	write
01	execute (search)
00	check existence of file

ERRORS

Access to the file is denied if one or more of the following are 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]	Permission bits of the file mode do not permit the requested access.
[EFAULT]	*path* points outside the allocated address space for the process.
[EINTR]	A signal was caught during the *access* system call.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

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.

SEE ALSO

chmod(2), stat(2).

DIAGNOSTICS

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.

NAME

acct – enable or disable process accounting

SYNOPSIS

int acct (path)
char *path;

DESCRIPTION

acct is used to enable or disable the system process accounting routine. If the routine is enabled, an accounting record will be 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*(2)]. The effective user ID of the calling process must be super-user to use this call.

path points to a pathname naming the accounting file. The accounting file format is given 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.

acct will fail if one or more of the following are true:

[EPERM]	The effective user 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.

SEE ALSO

exit(2), signal(2), acct(4).

DIAGNOSTICS

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

NAME

alarm – set a process alarm clock

SYNOPSIS

unsigned alarm (sec)
unsigned 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*(2)].

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.

SEE ALSO

pause(2), signal(2), sigset(2).

DIAGNOSTICS

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

NAME

bind – bind a name to a socket

SYNOPSIS

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

bind(s, name, namelen)
int s;
struct sockaddr *name;
int namelen;
```

DESCRIPTION

bind assigns a name to an unnamed socket. When a socket is created with *socket*(2) it exists in a name space (address family) but has no name assigned. *bind* requests that *name* be assigned to the socket.

NOTES

Binding a name in the UNIX domain creates a socket in the file system that must be deleted by the caller when it is no longer needed (using *unlink*(2)).

The rules used in name binding vary between communication domains. Consult the manual entries in section 4 for detailed information.

RETURN VALUE

If the bind is successful, a 0 value is returned. A return value of −1 indicates an error, which is further specified in the global *errno*.

ERRORS

The *bind* call will fail if:

[EBADF]	*s* is not a valid descriptor.
[ENOTSOCK]	*S* is not a socket.
[EADDRNOTAVAIL]	The specified address is not available from the local machine.
[EADDRINUSE]	The specified address is already in use.
[EINVAL]	The socket is already bound to an address.
[EACCES]	The requested address is protected, and the current user has inadequate permission to access it.
[EFAULT]	The *name* parameter is not in a valid part of the user address space.

The following errors are specific to binding names in the UNIX domain.

[ENOTDIR]	A component of the path prefix is not a directory.
[EINVAL]	The pathname contains a character with the high-order bit set.
[ENAMETOOLONG]	A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.
[ENOENT]	A prefix component of the path name does not exist.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EIO]	An I/O error occurred while making the directory entry or allocating the inode.
[EROFS]	The name would reside on a read-only file system.

[EISDIR] A null pathname was specified.

SEE ALSO

connect(2), listen(2), socket(2), getsockname(2)

ORIGIN

4.3 BSD

NAME

brk, sbrk – change data segment space allocation

SYNOPSIS

int brk (endds)
char *endds;

char *sbrk (incr)
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. Newly allocated space is set to zero. If, however, the same memory space is reallocated to the same process its contents are undefined.

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.

brk and *sbrk* will 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 the system-imposed maximum process size [see *ulimit*(2)].

[EAGAIN] Total amount of system memory available for a read during physical IO is temporarily insufficient [see *shmop*(2)]. This may occur even though the space requested was less than the system-imposed maximum process size [see *ulimit*(2)].

SEE ALSO

exec(2), shmop(2), ulimit(2).

DIAGNOSTICS

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.

NAME

cachectl – mark pages cacheable or uncacheable

SYNOPSIS

#include <mips/cachectl.h>

cachectl(addr, nbytes, op)
char *addr;
int nbytes, op;

DESCRIPTION

The *cachectl* system call allows a process to make ranges of its address space cacheable or uncacheable. Initially, a process's entire address space is cacheable.

op may be one of:

CACHEABLE	Make the indicated pages cacheable
UNCACHEABLE	Make the indicated pages uncacheable

The CACHEABLE and UNCACHEABLE op's affect the address range indicated by *addr* and *nbytes*. *addr* must be page aligned and *nbytes* must be a multiple of the page size.

Changing a page from UNCACHEABLE state to CACHEABLE state will cause both the instruction and data caches to be flushed if necessary to avoid stale cache information.

RETURN VALUE

cachectl returns 0 when no errors are detected. If errors are detected, *cachectl* returns -1 with the error cause indicated in *errno*.

ERRORS

[EINVAL]	*op* parameter is not one of CACHEABLE or UNCACHEABLE.
[EINVAL]	*addr* is not page aligned, or *nbytes* is not multiple of pagesize.
[EFAULT]	Some or all of the address range *addr* to (*addr+nbytes*-1) is not accessable.

NAME

cacheflush – flush contents of instruction and/or data cache

SYNOPSIS

```
#include <mips/cachectl.h>

cacheflush(addr, nbytes, cache)
char *addr;
int nbytes, cache;
```

DESCRIPTION

Flushes contents of indicated cache(s) for user addresses in the range *addr* to (*addr+nbytes*-1). *cache* may be one of:

ICACHE	Flush only the instruction cache
DCACHE	Flush only the data cache
BCACHE	Flush both instruction and data caches

RETURN VALUE

cacheflush returns 0 when no errors are detected. If errors are detected, *cacheflush* returns -1 with the error cause indicated in *errno*.

ERRORS

[EINVAL]	*cache* parameter is not one of ICACHE, DCACHE, or BCACHE.
[EFAULT]	Some or all of the address range *addr* to (*addr+nbytes*-1) is not accessable.

NAME

chdir – change working directory

SYNOPSIS

int chdir (path)
char *path;

DESCRIPTION

path points to the path name of a directory. *chdir* causes the named directory to become the current working directory, the starting point for path searches for path names not beginning with **/**.

chdir will fail and the current working directory will be 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.
[EINTR]	A signal was caught during the *chdir* system call.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

SEE ALSO

chroot(2).

DIAGNOSTICS

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

NAME

chmod, fchmod – change mode of file

SYNOPSIS

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

int fchmod (fd, mode)
int fd, mode;

DESCRIPTION

chmod sets the access permission portion of the mode of the named *path* or file described by the descriptor *fd* according to the bit pattern contained in *mode*.

Access permission bits are interpreted as follows:

04000	Set user ID on execution.
020#0	Set group ID on execution if # is **7**, **5**, **3**, or **1** Enable mandatory file/record locking if # is **6**, **4**, **2**, or **0**
01000	Save text image after execution.
00400	Read by owner.
00200	Write by owner.
00100	Execute (search if a directory) by owner.
00070	Read, write, execute (search) by group.
00007	Read, write, execute (search) by others.

The effective user ID of the process must match the owner of the file or be super-user to change the mode of a file.

If the effective user ID of the process is not super-user, mode bit 01000 (save text image on execution) is cleared.

If the effective user ID of the process is not super-user and the effective group ID of the process does not match the group ID of the file, mode bit 02000 (set group ID on execution) is cleared.

If a 410 executable file has the sticky bit (mode bit 01000) set, the operating system will not delete the program text from the swap area when the last user process terminates. If a 413 executable file has the sticky bit set, the operating system will not delete the program text from memory when the last user process terminates. In either case, if the sticky bit is set the text will already be available (either in a swap area or in memory) when the next user of the file executes it, thus making execution faster.

If the mode bit 02000 (set group ID on execution) is set and the mode bit 00010 (execute or search by group) is not set, mandatory file/record locking will exist on a regular file. This may effect future calls to open(2), creat(2), read(2), and write(2) on this file.

ERRORS

chmod will fail and the file mode will be unchanged if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.
[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* points outside the allocated address space of the process.
[EINTR]	A signal was caught during the *chmod* system call.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

SEE ALSO

chown(2), creat(2), fcntl(2), mknod(2), open(2), read(2), write(2).
chmod(1) in the *User's Reference Manual*.

DIAGNOSTICS

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

NAME

chown, fchown – change owner and group of a file

SYNOPSIS

int chown (path, owner, group)
char *path;
int owner, group;

int fchown (fd, owner, group)
int fd, owner, group;

DESCRIPTION

The owner ID and group ID of the named *path* or file described by file descriptor *fd* are set to the numeric values contained in *owner* and *group* respectively.

Only processes with effective user ID equal to the file owner or super-user may change the ownership of a file.

If *chown* is invoked by other than the super-user, the set-user-ID and set-group-ID bits of the file mode, 04000 and 02000 respectively, will be cleared.

ERRORS

chown will fail and the owner and group of the named file will remain unchanged if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.
[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* points outside the allocated address space of the process.
[EINTR]	A signal was caught during the *chown* system call.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

SEE ALSO

chmod(2).
chown(1) in the *User's Reference Manual.*

DIAGNOSTICS

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

NAME

chroot – change root directory

SYNOPSIS

int chroot (path)
char *path;

DESCRIPTION

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

The effective user ID of the process must be super-user 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.

ERRORS

chroot will fail and the root directory will remain unchanged if one or more of the following are true:

[ENOTDIR]	Any component of the path name is not a directory.
[ENOENT]	The named directory does not exist.
[EPERM]	The effective user ID is not super-user.
[EFAULT]	*path* points outside the allocated address space of the process.
[EINTR]	A signal was caught during the *chroot* system call.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

SEE ALSO

chdir(2).

DIAGNOSTICS

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

NAME

close – close a file descriptor

SYNOPSIS

int close (fildes)
int fildes;

DESCRIPTION

fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. *close* closes the file descriptor indicated by *fildes*. All outstanding record locks owned by the process (on the file indicated by *fildes*) are removed.

If a STREAMS [see *intro*(2)] file is closed, and the calling process had previously registered to receive a SIGPOLL signal [see *signal*(2) and *sigset*(2)] for events associated with that file [see I_SETSIG in *streamio*(7)], the calling process will be unregistered for events associated with the file. The last *close* for a *stream* causes the *stream* associated with *fildes* to be dismantled. If O_NDELAY is not set and there have been no signals posted for the *stream*, *close* waits up to 20 seconds, for each module and driver, for any output to drain before dismantling the *stream*. If the O_NDELAY flag is set or if there are any pending signals, *close* does not wait for output to drain, and dismantles the *stream* immediately.

The named file is closed unless one or more of the following are true:

[EBADF]	*fildes* is not a valid open file descriptor.
[EINTR]	A signal was caught during the *close* system call.
[ENOLINK]	*fildes* is on a remote machine and the link to that machine is no longer ctive.

SEE ALSO

creat(2), dup(2), exec(2), fcntl(2), intro(2), open(2), pipe(2), signal(2), sigset(2).
streamio(7) in the *System Administrator's Reference Manual*.

DIAGNOSTICS

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

NAME

connect – initiate a connection on a socket

SYNOPSIS

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

connect(s, name, namelen)
int s;
struct sockaddr *name;
int namelen;
```

DESCRIPTION

The parameter *s* is a socket. If it is of type SOCK_DGRAM, then this call specifies the peer with which the socket is to be associated; this address is that to which datagrams are to be sent, and the only address from which datagrams are to be received. If the socket is of type SOCK_STREAM, then this call attempts to make a connection to another socket. The other socket is specified by *name,* which is an address in the communications space of the socket. Each communications space interprets the *name* parameter in its own way. Generally, stream sockets may successfully *connect* only once; datagram sockets may use *connect* multiple times to change their association. Datagram sockets may dissolve the association by connecting to an invalid address, such as a null address.

RETURN VALUE

If the connection or binding succeeds, then 0 is returned. Otherwise a −1 is returned, and a more specific error code is stored in *errno*.

ERRORS

The call fails if:

[EBADF]	*S* is not a valid descriptor.
[ENOTSOCK]	*S* is a descriptor for a file, not a socket.
[EADDRNOTAVAIL]	The specified address is not available on this machine.
[EAFNOSUPPORT]	Addresses in the specified address family cannot be used with this socket.
[EISCONN]	The socket is already connected.
[ETIMEDOUT]	Connection establishment timed out without establishing a connection.
[ECONNREFUSED]	The attempt to connect was forcefully rejected.
[ENETUNREACH]	The network isn't reachable from this host.
[EADDRINUSE]	The address is already in use.
[EFAULT]	The *name* parameter specifies an area outside the process address space.
[EINPROGRESS]	The socket is non-blocking and the connection cannot be completed immediately. It is possible to *select*(2) for completion by selecting the socket for writing.
[EALREADY]	The socket is non-blocking and a previous connection attempt has not yet been completed.

The following errors are specific to connecting names in the UNIX domain. These errors may not apply in future versions of the UNIX IPC domain.

[ENOTDIR]	A component of the path prefix is not a directory.
[EINVAL]	The pathname contains a character with the high-order bit set.
[ENAMETOOLONG]	A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.
[ENOENT]	The named socket does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[EACCES]	Write access to the named socket is denied.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.

SEE ALSO

accept(2), select(2), socket(2), getsockname(2)

ORIGIN

4.3 BSD

NAME

creat – create a new file or rewrite an existing one

SYNOPSIS

int creat (path, mode)
char *path;
int mode;

DESCRIPTION

creat creates a new ordinary file or prepares to rewrite an existing file named by the path name pointed to by *path*.

If the file exists, the length is truncated to 0 and the mode and owner are unchanged. Otherwise, the file's owner ID is set to the effective user ID, of the process the group ID of the process is set to the effective group ID, of the process and 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, a write-only file descriptor is returned and the file is open for writing, even if the mode does not permit writing. The file pointer 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 may have more than NOFILES files open simultaneously. A new file may be created with a mode that forbids writing.

ERRORS

creat fails if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	A component of the path prefix does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.
[ENOENT]	The path name is null.
[EACCES]	The file does not exist and the directory in which the file is to be created does not permit writing.
[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.
[EACCES]	The file exists and write permission is denied.
[EISDIR]	The named file is an existing directory.
[EMFILE]	NOFILES file descriptors are currently open.
[EFAULT]	*path* points outside the allocated address space of the process.
[ENFILE]	The system file table is full.
[EAGAIN]	The file exists, mandatory file/record locking is set, and there are outstanding record locks on the file [see *chmod*(2)].
[EINTR]	A signal was caught during the *creat* system call.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.
[ENOSPC]	The file system is out of inodes.

SEE ALSO

chmod(2), close(2), dup(2), fcntl(2), lseek(2), open(2), read(2), umask(2), write(2).

DIAGNOSTICS

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.

NAME

dup – duplicate an open file descriptor

SYNOPSIS

int dup (fildes)
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).

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.

ERRORS

dup will fail if one or more of the following are true:

[EBADF]	*fildes* is not a valid open file descriptor.
[EINTR]	A signal was caught during the *dup* system call.
[EMFILE]	NOFILES file descriptors are currently open.
[ENOLINK]	*fildes* is on a remote machine and the link to that machine is no longer active.

SEE ALSO

close(2), creat(2), exec(2), fcntl(2), open(2), pipe(2), lockf(3C).

DIAGNOSTICS

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.

NAME

exec: execl, execv, execle, execve, execlp, execvp – execute a file

SYNOPSIS

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

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

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

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

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

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

DESCRIPTION

exec in all its forms transforms the calling process into a new process. The new process is constructed from an ordinary, executable file called the "new process file". The new process file may be either an "interpreter script" which begins with the characters "#!", or an a.out file.

On the first line of an interpreter script, following the "#!", is the name of a program which should be used to interpret the contents of the file. For instance, if the first line contains "#! /bin/sh", then the contents of the file are executed as a shell script. An a.out file consists of a header, a text segment, and a data segment. The data segment contains an initialized portion and an uninitialized portion (bss).

There can be no return from a successful exec because the calling process is overlaid by the new process.

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, *argv* is an array of character pointers to the arguments themselves, and *envp* is an array of character pointers to the environment strings. As indicated, *argc* is conventionally at least one and the first member of the array points to a string containing the name of the file.

path points to a path name that identifies the new process file.

file points to the new process file. The path prefix for this file is obtained by a search of the directories passed as the *environment* line "PATH =" [see *environ(5)*]. The environment is supplied by the shell [see *sh(1)* or *csh(1)*].

arg0, *arg1*, ..., *argn* are pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, at least *arg0* must be present and point to a string that is the same as *path* (or its last component).

argv is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new process. By convention, *argv* must have at least one member, and it must point to a string that is the same as *path* (or its 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 for the new process. *envp* is terminated by a null pointer. For *execl* and *execv*, the C run-time start-off routine places a pointer to the environment of the calling process in the global cell:

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

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

File descriptors open in the calling process remain open in the new process, except for those whose close-on-exec flag is set; see *fcntl(2)*. For those file descriptors that remain open, the file pointer is unchanged.

Signals set to terminate the calling process will be set to terminate the new process. Signals set to be ignored by the calling process will be set to be ignored by the new process. Signals set to be caught by the calling process will be set to terminate new process; see *signal(2)*.

For signals set by *sigset(2)*, *exec* will ensure that the new process has the same system signal action for each signal type whose action is SIG_DFL, SIG_IGN, or SIG_HOLD as the calling process. However, if the action is to catch the signal, then the action will be reset to SIG_DFL, and any pending signal for this type will be held.

If the set-user-ID mode bit of the new process file is set [see *chmod(2)*], *exec* sets the effective user ID of the new process to the owner ID of the new process file. Similarly, if the set-group-ID mode bit of the new process file is set, the effective group ID of the new process is set to the group ID of the new process file. The real user ID and real group ID of the new process remain the same as those of the calling process.

The shared memory segments attached to the calling process will not be attached to the new process [see *shmop(2)*].

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

The new process also inherits the following attributes from the calling process:

nice value [see *nice(2)*]
process ID
parent process ID
process group ID
semadj values [see *semop(2)*]
tty group ID [see *exit(2)* and *signal(2)*]
trace flag [see *ptrace(2)* request 0]
time left until an alarm clock signal [see *alarm(2)*]
current working directory
root directory
file mode creation mask [see *umask(2)*]
file size limit [see *ulimit(2)*]
utime, *stime*, *cutime*, and *cstime* [see *times(2)*]
file-locks [see *fcntl(2)* and *lockf(3C)*]

ERRORS

exec will fail and return to the calling process if one or more of the following are true:

[ENOENT]	One or more components of the new process path name of the file do not exist.
[ENOTDIR]	A component of the new process path of the file prefix is not a directory.
[EACCES]	Search permission is denied for a directory listed in the new process file's path prefix.
[EACCES]	The new process file is not an ordinary file.

[EACCES]	The new process file mode denies execution permission.
[ENOEXEC]	The *exec* is not an *execlp* or *execvp*, and the new process file has the appropriate access permission but an invalid magic number in its header.
[ETXTBSY]	The new process file is a pure procedure (shared text) file that is currently open for writing by some process.
[ENOMEM]	The new process requires more memory than is allowed by the system-imposed maximum MAXMEM.
[E2BIG]	The number of bytes in the new process's argument list is greater than the system-imposed limit of NCARGS.
[EFAULT]	Required hardware is not present.
[EFAULT]	*path*, *argv*, or *envp* point to an illegal address.
[EAGAIN]	Not enough memory.
[ELIBACC]	Required shared library does not have execute permission.
[ELIBEXEC]	Trying to *exec* a shared library directly.
[EINTR]	A signal was caught during the *exec* system call.
[ENOLINK]	*Path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.
[EINVAL]	Trying to *exec* a file that calls for a nonexistent interpreter.

SEE ALSO

alarm(2), exit(2), fcntl(2), fork(2), nice(2), ptrace(2), semop(2), signal(2), sigset(2), times(2), ulimit(2), umask(2), lockf(3C), environ(5).
sh(1) in the *User's Reference Manual.*

DIAGNOSTICS

If *exec* returns to the calling process an error has occurred; the return value will be −1 and *errno* will be set to indicate the error.

NAME

exit, _exit – terminate process

SYNOPSIS

void exit (status)
int status;
void _exit (status)
int status;

DESCRIPTION

exit terminates the calling process with the following consequences:

All of the file descriptors open in the calling process are closed.

If the parent process of the calling process is executing a *wait*, 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*, 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. The process table slot that it occupies is partially overlaid with time accounting information (see **<sys/proc.h>**) to be used by *times*.

The parent process ID of all of the calling processes' existing child processes and zombie processes is set to 1. This means the initialization process [see *intro*(2)] 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.

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)].

If the process ID, tty group ID, and process group ID of the calling process are equal, the **SIGHUP** signal is sent to each process that has a process group ID equal to that of the calling process.

A death of child signal is sent to the parent.

The C function *exit* may cause cleanup actions before the process exits. The function *_exit* circumvents all cleanup.

SEE ALSO

acct(2), intro(2), plock(2), semop(2), signal(2), sigset(2), wait(2).

WARNING

See **NOTE** in *signal*(2) and **WARNING** in *sigset*(2).

DIAGNOSTICS

None. There can be no return from an *exit* system call.

NAME

fcntl – file control

SYNOPSIS

#include <fcntl.h>

int fcntl (fildes, cmd, arg)
int fildes, cmd, arg;

DESCRIPTION

fcntl provides for control over open files. *fildes* is an open file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

The commands available are:

F_DUPFD — Return a new file descriptor as follows:

Lowest numbered available file descriptor greater than or equal to *arg*.

Same open file (or pipe) as the original file.

Same file pointer as the original file (i.e., both file descriptors share one file pointer).

Same access mode (read, write or read/write).

Same file status flags (i.e., 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*, otherwise the file will be closed upon execution of *exec*.

F_SETFD — Set the close-on-exec flag associated with *fildes* to the low-order bit of *arg* (**0** or **1** as above).

F_GETFL — Get *file* status flags.

F_SETFL — Set *file* status flags to *arg*. Only certain flags can be set [see *fcntl*(5)].

F_GETLK — Get the first lock which blocks the lock description given 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, then the structure is passed back unchanged except for the lock type which will be 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* [see *fcntl*(5)]. The *cmd* F_SETLK is used to establish read (F_RDLCK) and write (F_WRLCK) locks, as well as remove either type of lock (F_UNLCK). If a read or write lock cannot be set *fcntl* will return 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 process from write locking the protected area. More than one read lock may 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 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*), process id (*l_pid*), and RFS system id (*l_sysid*) of the segment of the file to be affected. The process id and system id fields are used only with the F_GETLK *cmd* to return the values for a blocking lock. Locks may start and extend beyond the current end of a file, but may not be negative relative to the beginning of the file. A lock may be set to always extend to the end of file by setting *l_len* to zero (0). If such a lock also has *l_whence* and *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 that is 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 mandatory file and record locking is active on a file, [see *chmod*(2)], *read* and *write* system calls issued on the file will be affected by the record locks in effect.

ERRORS

fcntl will fail if one or more of the following are true:

[EBADF]	*fildes* is not a valid open file descriptor.
[EINVAL]	*cmd* is F_DUPFD. *arg* is either negative, or greater than or equal to the configured value for the maximum number of open file descriptors allowed each user.
[EINVAL]	*cmd* is F_GETLK, F_SETLK, or SETLKW and *arg* or the data it points to is not valid.
[EACCES]	*cmd* is F_SETLK the type of lock (*l_type*) is a read (F_RDLCK) lock and the segment of a file to be locked is already write locked by another process or the type is a write (F_WRLCK) lock 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 there are no more record locks available (too many file segments locked) because the system maximum has been exceeded.
[EDEADLK]	*cmd* is F_SETLKW, the lock is blocked by some lock from another process, and putting the calling-process to sleep, waiting for that lock to become free, would cause a deadlock.
[EFAULT]	*cmd* is F_SETLK, *arg* points outside the program address space.
[EINTR]	A signal was caught during the *fcntl* system call.
[ENOLINK]	*fildes* is on a remote machine and the link to that machine is no longer active.

SEE ALSO

close(2), creat(2), dup(2), exec(2), fork(2), open(2), pipe(2), fcntl(5).

DIAGNOSTICS

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

F_DUPFD	A new file descriptor.
F_GETFD	Value of flag (only the low-order bit is defined).
F_SETFD	Value other than −1.
F_GETFL	Value of file flags.
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.

WARNINGS

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.

NAME

fork – create a new process

SYNOPSIS

int fork ()

DESCRIPTION

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

environment
close-on-exec flag [see *exec*(2)]
signal handling settings (i.e., **SIG_DFL**, **SIG_IGN**, **SIG_HOLD**, function address)
set-user-ID mode bit
set-group-ID mode bit
profiling on/off status
nice value [see *nice*(2)]
all attached shared memory segments [see *shmop*(2)]
process group ID
tty group ID [see *exit*(2)]
current working directory
root directory
file mode creation mask [see *umask*(2)]
file size limit [see *ulimit*(2)]

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

The child process has a unique process ID.

The child process has a different parent process ID (i.e., the process ID of the parent process).

The child process has its own copy of the parent's file descriptors. Each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent.

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

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

The child process's *utime*, *stime*, *cutime*, and *cstime* are set to 0. The time left until an alarm clock signal is reset to 0.

ERRORS

fork will fail and no child process will be created if one or more of the following are 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.
[EAGAIN]	Total amount of system memory available when reading via raw IO is temporarily insufficient.

SEE ALSO

exec(2), nice(2), plock(2), ptrace(2), semop(2), shmop(2), signal(2), sigset(2), times(2), ulimit(2), umask(2), wait(2).

DIAGNOSTICS

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.

NAME

getdents – read directory entries and put in a file system independent format

SYNOPSIS

#include <sys/dirent.h>

int getdents (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;

DESCRIPTION

fildes is a file descriptor obtained from an *open*(2) or *dup*(2) system call.

getdents attempts to read *nbyte* bytes from the directory associated with *fildes* and to format them as file system independent directory entries in the buffer pointed to by *buf*. Since the file system independent directory entries are of variable length, in most cases the actual number of bytes returned will be strictly less than *nbyte*.

The file system independent directory entry is specified by the *dirent* structure. For a description of this see *dirent*(4).

On devices capable of seeking, *getdents* starts at a position in the file given by the file pointer associated with *fildes*. Upon return from *getdents,* the file pointer is incremented to point to the next directory entry.

This system call was developed in order to implement the *readdir*(3X) routine [for a description see *directory*(3X)], and should not be used for other purposes.

getdents will fail if one or more of the following are true:

[EBADF]	*fildes* is not a valid file descriptor open for reading.
[EFAULT]	*Buf* points outside the allocated address space.
[EINVAL]	*nbyte* is not large enough for one directory entry.
[ENOENT]	The current file pointer for the directory is not located at a valid entry.
[ENOLINK]	*fildes* points to a remote machine and the link to that machine is no longer active.
[ENOTDIR]	*fildes* is not a directory.
[EIO]	An I/O error occurred while accessing the file system.

SEE ALSO

directory(3X), dirent(4).

DIAGNOSTICS

Upon successful completion a non-negative integer is returned indicating the number of bytes actually read. A value of 0 indicates the end of the directory has been reached. If the system call failed, a −1 is returned and *errno* is set to indicate the error.

NAME

gethostid, sethostid – get/set unique identifier of current host

SYNOPSIS

hostid = gethostid()
long hostid;

sethostid(hostid)
long hostid;

DESCRIPTION

sethostid establishes a 32-bit identifier for the current processor that is intended to be unique among all UNIX systems in existence. This is normally a DARPA Internet address for the local machine. This call is allowed only to the super-user and is normally performed at boot time.

gethostid returns the 32-bit identifier for the current processor.

SEE ALSO

hostid(1), gethostname(2)

ERRORS

32 bits for the identifier is too small.

NAME

gethostname, sethostname – get/set name of current host

SYNOPSIS

gethostname(name, namelen)
char *name;
int namelen;

sethostname(name, namelen)
char *name;
int namelen;

DESCRIPTION

gethostname returns the standard host name for the current processor, as previously set by *sethostname*. The parameter *namelen* specifies the size of the *name* array. The returned name is null-terminated unless insufficient space is provided.

sethostname sets the name of the host machine to be *name*, which has length *namelen*. This call is restricted to the super-user and is normally used only when the system is bootstrapped.

RETURN VALUE

If the call succeeds a value of 0 is returned. If the call fails, then a value of −1 is returned and an error code is placed in the global location *errno*.

ERRORS

The following errors may be returned by these calls:

[EFAULT]	The *name* or *namelen* parameter gave an invalid address.
[EPERM]	The caller tried to set the hostname and was not the super-user.

EINVAL: The size specified by I. namelen is longer than the maximum host name length.

SEE ALSO

gethostid(2)

ERRORS

Host names are limited to MAXHOSTNAMELEN (from <*sys/param.h*>) characters, currently 64.

NAME

getitimer, setitimer – get/set value of interval timer

SYNOPSIS

```
#include <bsd/sys/time.h>

#define ITIMER_REAL      0      /* real time intervals */
#define ITIMER_VIRTUAL   1      /* virtual time intervals */
#define ITIMER_PROF      2      /* user and system virtual time */

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

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

DESCRIPTION

The system provides each process with three interval timers, defined in *<bsd/sys/time.h>*. The *getitimer* call returns the current value for the timer specified in *which* in the structure at *value*. The *setitimer* call sets a timer to the specified *value* (returning the previous value of the timer if *ovalue* is nonzero).

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 (on the VAX, 10 milliseconds).

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. Because this signal may interrupt in-progress system calls, programs using this timer must be prepared to restart interrupted system calls.

NOTES

Three macros for manipulating time values are defined in *<bsd/sys/time.h>*. *Timerclear* sets a time value to zero, *timerisset* tests if a time value is non-zero, and *timercmp* compares two time values (beware that >= and <= do not work with this macro).

RETURN VALUE

If the calls succeed, a value of 0 is returned. If an error occurs, the value −1 is returned, and a more precise error code is placed in the global variable *errno*.

ERRORS

The possible errors are:

[EFAULT]	The *value* parameter specified a bad address.
[EINVAL]	A *value* parameter specified a time was too large to be handled.

NAME

getmsg – get next message off a stream

SYNOPSIS

```
#include <stropts.h>

int getmsg(fd, ctlptr, dataptr, flags)
int fd;
struct strbuf *ctlptr;
struct strbuf *dataptr;
int *flags;
```

DESCRIPTION

getmsg retrieves the contents of a message [see *intro*(2)] located at the *stream head* read queue from a STREAMS file, and places the contents into user specified buffer(s). The message must contain either a data part, a control part or both. The data and control parts of the message are placed into separate buffers, as described below. The semantics of each part is defined by the STREAMS module that generated the message.

fd specifies a file descriptor referencing an open *stream*. *ctlptr* and *dataptr* each point to a *strbuf* structure which contains the following members:

```
int maxlen;     /* maximum buffer length */
int len;        /* length of data        */
char *buf;      /* ptr to buffer         */
```

where *buf* points to a buffer in which the data or control information is to be placed, and *maxlen* indicates the maximum number of bytes this buffer can hold. On return, *len* contains the number of bytes of data or control information actually received, or is 0 if there is a zero-length control or data part, or is -1 if no data or control information is present in the message. lags may be set to the values 0 or RS_HIPRI and is used as described below.

ctlptr is used to hold the control part from the message and *dataptr* is used to hold the data part from the message. If *ctlptr* (or *dataptr*) is NULL or the *maxlen* field is -1, the control (or data) part of the message is not processed and is left on the *stream head* read queue and *len* is set to -1. If the *maxlen* field is set to 0 and there is a zero-length control (or data) part, that zero-length part is removed from the read queue and *len* is set to 0. If the *maxlen* field is set to 0 and there are more than zero bytes of control (or data) information, that information is left on the read queue and *len* is set to 0. If the *maxlen* field in *ctlptr* or *dataptr* is less than, respectively, the control or data part of the message, *maxlen* bytes are retrieved. In this case, the remainder of the message is left on the *stream head* read queue and a non-zero return value is provided, as described below under *DIAGNOSTICS*. If information is retrieved from a *priority* message, *flags* is set to RS_HIPRI on return.

By default, *getmsg* processes the first priority or non-priority message available on the *stream head* read queue. However, a user may choose to retrieve only priority messages by setting *flags* to RS_HIPRI. In this case, *getmsg* will only process the next message if it is a priority message.

If O_NDELAY has not been set, *getmsg* blocks until a message, of the type(s) specified by *flags* (priority or either), is available on the *stream head* read queue. If O_NDELAY has been set and a message of the specified type(s) is not present on the read queue, *getmsg* fails and sets *errno* to EAGAIN.

If a hangup occurs on the *stream* from which messages are to be retrieved, *getmsg* will continue to operate normally, as described above, until the *stream head* read queue is empty. Thereafter, it will return 0 in the *len* fields of *ctlptr* and *dataptr*.

ERRORS

getmsg fails if one or more of the following are true:

[EAGAIN]	The O_NDELAY flag is set, and no messages are available.
[EBADF]	*fd* is not a valid file descriptor open for reading.
[EBADMSG]	Queued message to be read is not valid for *getmsg*.
[EFAULT]	*ctlptr*, *dataptr*, or *flags* points to a location outside the allocated address space.
[EINTR]	A signal was caught during the *getmsg* system call.
[EINVAL]	An illegal value was specified in *flags*, or the *stream* referenced by *fd* is linked under a multiplexor.
[ENOSTR]	A *stream* is not associated with *fd*.

A *getmsg* can also fail if a STREAMS error message had been received at the *stream head* before the call to *getmsg*. The error returned is the value contained in the STREAMS error message.

SEE ALSO

intro(2), read(2), poll(2), putmsg(2), write(2).
STREAMS Primer
STREAMS Programmer's Guide

DIAGNOSTICS

Upon successful completion, a non-negative value is returned. A value of 0 indicates that a full message was read successfully. A return value of MORECTL indicates that more control information is waiting for retrieval. A return value of MOREDATA indicates that more data is waiting for retrieval. A return value of MORECTL|MOREDATA indicates that both types of information remain. Subsequent *getmsg* calls will retrieve the remainder of the message.

NAME

getpagesize – get system page size

SYNOPSIS

pagesize = getpagesize()
int pagesize;

DESCRIPTION

getpagesize returns the number of bytes in a page. Page granularity is the granularity of many of the memory management calls.

The page size is a *system* page size and may not be the same as the underlying hardware page size.

SEE ALSO

sbrk(2).

NAME

getpeername – get name of connected peer

SYNOPSIS

getpeername(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;

DESCRIPTION

getpeername returns the name of the peer connected to socket *s*. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return it contains the actual size of the name returned (in bytes). The name is truncated if the buffer provided is too small.

DIAGNOSTICS

A 0 is returned if the call succeeds, −1 if it fails.

ERRORS

The call succeeds unless:

[EBADF]	The argument *s* is not a valid 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 *name* parameter points to memory not in a valid part of the process address space.

SEE ALSO

accept(2), bind(2), socket(2), getsockname(2)

NAME

getpid, getpgrp, getppid – get process, process group, and parent process IDs

SYNOPSIS

int getpid ()

int getpgrp ()

int getppid ()

DESCRIPTION

getpid returns the process ID of the calling process.

getpgrp returns the process group ID of the calling process.

getppid returns the parent process ID of the calling process.

SEE ALSO

exec(2), fork(2), intro(2), setpgrp(2), signal(2).

NAME

getsockname – get socket name

SYNOPSIS

getsockname(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;

DESCRIPTION

getsockname returns the current *name* for the specified socket. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return it contains the actual size of the name returned (in bytes).

DIAGNOSTICS

A 0 is returned if the call succeeds, −1 if it fails.

ERRORS

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[EFAULT]	The *name* parameter points to memory not in a valid part of the process address space.

SEE ALSO

bind(2), socket(2)

BUGS

Names bound to sockets in the UNIX domain are inaccessible; *getsockname* returns a zero length name.

NAME

getsockopt, setsockopt – get and set options on sockets

SYNOPSIS

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

getsockopt(s, level, optname, optval, optlen)
int s, level, optname;
char *optval;
int *optlen;

setsockopt(s, level, optname, optval, optlen)
int s, level, optname;
char *optval;
int optlen;
```

DESCRIPTION

getsockopt and *setsockopt* manipulate *options* associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost "socket" level.

When manipulating socket options the level at which the option resides and the name of the option must be specified. To manipulate options at the "socket" level, *level* is specified as SOL_SOCKET. To manipulate options at any other level the protocol number of the appropriate protocol controlling the option is supplied. For example, to indicate that an option is to be interpreted by the TCP protocol, *level* should be set to the protocol number of TCP ; see *getprotoent*(3N).

The parameters *optval* and *optlen* are used to access option values for *setsockopt*. For *getsockopt* they identify a buffer in which the value for the requested option(s) are to be returned. For *getsockopt*, *optlen* is a value-result parameter, initially containing the size of the buffer pointed to by *optval*, and modified on return to indicate the actual size of the value returned. If no option value is to be supplied or returned, *optval* may be supplied as 0.

optname and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file *<bsd/sys/socket.h>* contains definitions for "socket" level options, described below. Options at other protocol levels vary in format and name; consult the appropriate entries in section (4P).

Most socket-level options take an *int* parameter for *optval*. For *setsockopt*, the parameter should non-zero to enable a boolean option, or zero if the option is to be disabled. O_LINGER uses a *struct linger* parameter, defined in *<bsd/sys/socket.h>*, which specifies the desired state of the option and the linger interval (see below).

The following options are recognized at the socket level. Except as noted, each may be examined with *getsockopt* and set with *setsockopt*.

SO_DEBUG	toggle recording of debugging information
SO_REUSEADDR	toggle local address reuse
SO_KEEPALIVE	toggle keep connections alive
SO_DONTROUTE	toggle routing bypass for outgoing messages
SO_LINGER	linger on close if data present
SO_BROADCAST	toggle permission to transmit broadcast messages
SO_OOBINLINE	toggle reception of out-of-band data in band
SO_SNDBUF	set buffer size for output
SO_RCVBUF	set buffer size for input
SO_TYPE	get the type of the socket (get only)
SO_ERROR	get and clear error on the socket (get only)

SO_DEBUG enables debugging in the underlying protocol modules. SO_REUSEADDR indicates that the rules used in validating addresses supplied in a *bind*(2) call should allow reuse of local addresses. SO_KEEPALIVE enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken and processes using the socket are notified via a SIGPIPE signal. SO_DONTROUTE indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

SO_LINGER controls the action taken when unsent messages are queued on socket and a *close*(2) is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system will block 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 (a timeout period, termed the linger interval, is specified in the *setsockopt* call when SO_LINGER is requested). If SO_LINGER is disabled and a *close* is issued, the system will process the close in a manner that allows the process to continue as quickly as possible.

The option SO_BROADCAST requests permission to send broadcast datagrams on the socket. Broadcast was a privileged operation in earlier versions of the system. With protocols that support out-of-band data, the SO_OOBINLINE option requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with *recv* or *read* calls without the MSG_OOB flag. SO_SNDBUF and SO_RCVBUF are options to adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections, or may be decreased to limit the possible backlog of incoming data. The system places an absolute limit on these values. Finally, SO_TYPE and SO_ERROR are options used only with *setsockopt*. SO_TYPE returns the type of the socket, such as SOCK_STREAM; it is useful for servers that inherit sockets on startup. SO_ERROR returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

RETURN VALUE

A 0 is returned if the call succeeds, −1 if it fails.

ERRORS

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOPROTOOPT]	The option is unknown at the level indicated.
[EFAULT]	The address pointed to by *optval* is not in a valid part of the process address space. For *getsockopt*, this error may also be returned if *optlen* is not in a valid part of the process address space.

SEE ALSO

ioctl(2), socket(2), getprotoent(3N)

BUGS

Several of the socket options should be handled at lower levels of the system.

NAME

getuid, geteuid, getgid, getegid – get real user, effective user, real group, and effective group IDs

SYNOPSIS

unsigned short getuid ()

unsigned short geteuid ()

unsigned short getgid ()

unsigned short getegid ()

DESCRIPTION

getuid returns the real user ID of the calling process.

geteuid returns the effective user ID of the calling process.

getgid returns the real group ID of the calling process.

getegid returns the effective group ID of the calling process.

SEE ALSO

intro(2), setuid(2).

NAME

intro – introduction to system calls and error numbers

SYNOPSIS

#include <errno.h>

DESCRIPTION

This section describes all of the system calls. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible returned value. This is almost always −1 or the NULL pointer; the individual descriptions specify the details. An error number is also made available in the external variable *errno*. *errno* is not cleared on successful calls, so it should be tested only after an error has been indicated.

Each system call description attempts to list all possible error numbers. The following is a complete list of the error numbers and their names as defined in **<errno.h>**.

1 EPERM Not owner

Typically this error indicates an attempt to modify a file in some way forbidden except to its owner or super-user. It is also returned for attempts by ordinary users to do things allowed only to the super-user.

2 ENOENT No such file or directory

This error occurs when a file name is specified and the file should exist but doesn't, or when one of the directories in a path name does not exist.

3 ESRCH No such process

No process can be found corresponding to that specified by *pid* in *kill(2)* or *ptrace(2)*.

4 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.

5 EIO I/O error

Some physical I/O error has occurred. This error may in some cases occur on a call following the one to which it actually applies.

6 ENXIO No such device or address

I/O on a special file refers to a subdevice which does not exist, or beyond the limits of the device. It may also occur when, for example, a tape drive is not on-line or no disk pack is loaded on a drive.

7 E2BIG Arg list too long

An argument list longer than NCARGS
(usually 5,120) bytes is presented to a member of the *exec(2)* family. NCARGS is defined in sys/param.h.

8 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)*].

9 EBADF Bad file number

Either a file descriptor refers to no open file, or a *read(2)* [respectively, *write(2)*] request is made to a file which is open only for writing (respectively, reading).

10 ECHILD No child processes

A *wait* was executed by a process that had no existing or unwaited-for child processes.

11 EAGAIN No more resources or 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 system call failed because of insufficient memory or

swap space.

12 ENOMEM Not enough space
During an *exec(2)*, *brk(2)*, or *sbrk* (see *brk(2)*), 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 may 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(2)*. If this error occurs on a resource associated with Remote File Sharing (RFS), it indicates a memory depletion which may be temporary, dependent on system activity at the time the call was invoked.

13 EACCES Permission denied
An attempt was made to access a file in a way forbidden by the protection system.

14 EFAULT Bad address
The system encountered a hardware fault in attempting to use an argument of a system call.

15 ENOTBLK Block device required
A non-block file was mentioned where a block device was required, e.g., in *mount(2)*.

16 EBUSY Device or resource busy
An attempt was made 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.

17 EEXIST File exists
An existing file was mentioned in an inappropriate context, e.g., *link(2)*.

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

19 ENODEV No such device
An attempt was made to apply an inappropriate system call to a device; e.g., read a write-only device.

20 ENOTDIR Not a directory
A non-directory was specified where a directory is required, for example in a path prefix or as an argument to *chdir(2)*.

21 EISDIR Is a directory
An attempt was made to write on a directory.

22 EINVAL Invalid argument
Some invalid argument (e.g., dismounting a non-mounted device; mentioning an undefined signal in *signal(2)* or *kill(2)*; reading or writing a file for which *lseek(2)* has generated a negative pointer). Also set by the math functions described in the (3M) entries of this manual.

23 ENFILE File table overflow
The system file table is full, and temporarily no more *opens* can be accepted.

24 EMFILE Too many open files
No process may have more than NOFILES (default 20) descriptors open at a time.

25 ENOTTY Not a character device (or) Not a typewriter
An attempt was made to *ioctl(2)* a file that is not a special character device.

26 ETXTBSY Text file busy
An attempt was made to execute a pure-procedure program that is currently open for writing. Also an attempt to open for writing or to remove a pure-procedure program that is being executed.

27 EFBIG File too large
The size of a file exceeded the maximum file size or ULIMIT [see *ulimit(2)*].

28 ENOSPC No space left on device
During a *write(2)* to an ordinary file, there is no free space left on the device. In *fcntl(2)*, the setting or removing of record locks on a file cannot be accomplished because there are no more record entries left on the system.

29 ESPIPE Illegal seek
An *lseek(2)* was issued to a pipe.

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

31 EMLINK Too many links
An attempt to make more than the maximum number of links to a file, as defined by MAXLINK in sys/param.h (usually 1000).

32 EPIPE Broken pipe
A write on a pipe for which there is no process to read the data. This condition normally generates a signal; the error is returned if the signal is ignored.

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

34 ERANGE Result too large
The value of a function in the math package (3M) is not representable within machine precision.

35 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)*].

36 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)*].

37-44 Reserved numbers

45 EDEADLK Deadlock
A deadlock situation was detected and avoided. This error pertains to file and record locking.

46 ENOLCK No lock
In *fcntl(2)* the setting or removing of record locks on a file cannot be accomplished because there are no more record entries left on the system.

60 ENOSTR Not a stream
A *putmsg(2)* or *getmsg(2)* system call was attempted on a file descriptor that is not a STREAMS device.

62 ETIME Stream ioctl timeout
The timer set for a STREAMS *ioctl(2)* call has expired. The cause of this error is device specific and could indicate either a hardware or software failure, or perhaps a timeout value that is too short for the specific operation. The status of the *ioctl(2)* operation is indeterminate.

63 ENOSR No stream resources
During a STREAMS *open(2)*, either no STREAMS queues or no STREAMS head data structures were available.

64 ENONET Machine is not on the network
This error is Remote File Sharing (RFS) specific. It occurs when users try to advertise, unadvertise, mount, or unmount remote resources while the machine has not done the proper startup to connect to the network.

65 ENOPKG No package
This error occurs when users attempt to use a system call from a package which has not been installed.

66 EREMOTE Resource is remote
This error is RFS specific. It occurs when users try to advertise a resource which is not on the local machine, or try to mount/unmount a device (or pathname) that is on a remote machine.

67 ENOLINK Virtual circuit is gone
This error is RFS specific. It occurs when the link (virtual circuit) connecting to a remote machine is gone.

68 EADV Advertise error
This error is RFS specific. It occurs when users try to advertise a resource which has been advertised already, or try to stop the RFS while there are resources still advertised, or try to force unmount a resource when it is still advertised.

69 ESRMNT Srmount error
This error is RFS specific. It occurs when users try to stop RFS while there are resources still mounted by remote machines.

70 ECOMM Communication error
This error is RFS specific. It occurs when trying to send messages to remote machines but no virtual circuit can be found.

71 EPROTO Protocol error
Some protocol error occurred. This error is device specific, but is generally not related to a hardware failure.

74 EMULTIHOP Multihop attempted
This error is RFS specific. It occurs when users try to access remote resources which are not directly accessible.

77 EBADMSG Bad message
During a *read(2)*, *getmsg(2)*, or *ioctl(2)* I_RECVFD system call to a STREAMS device, something has come to the head of the queue that can't be processed. That something depends on the system call:
read(2) - control information or a passed file descriptor.
getmsg(2) - passed file descriptor.
ioctl(2) - control or data information.

83 ELIBACC Cannot access a needed shared library
Trying to *exec(2)* an *a.out* that requires a shared library (to be linked in) and the shared library doesn't exist or the user doesn't have permission to use it.

84 ELIBBAD Accessing a corrupted shared library
Trying to *exec(2)* an *a.out* that requires a shared library (to be linked in) and *exec(2)* could not load the shared library. The shared library is probably corrupted.

85 ELIBSCN .lib section in *a.out* corrupted
Trying to *exec(2)* an *a.out* that requires a shared library (to be linked in) and there was erroneous data in the .lib section of the *a.out*. The .lib section tells *exec(2)* what shared libraries are needed. The *a.out* is probably corrupted.

86 ELIBMAX Attempting to link in more shared libraries than system limit
Trying to *exec(2)* an *a.out* that requires more shared libraries (to be linked in) than is allowed on the current configuration of the system. See the System Administrator's Guide.

87 ELIBEXEC Cannot exec a shared library directly
Trying to *exec(2)* a shared library directly. This is not allowed.

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

102 EINPROGRESS Operation now in progress
An operation that takes a long time to complete (such as a *connect(2)*) was attempted on a non_clocking object (see *fcntl(2)*).

103 EALREADY Ooperation already in progress
An operation was attempted on a non-blocking object that already had an operation in progress.

104 ENOTSOCK Socket operation on non-socket
Self-explanatory.

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

106 EMSGSIZE Message too long
A message sent on a socket was larger than the internal message buffer or some other network limit.

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

108 ENOPROTOOPT Option not supported by protocol
A bad option or level was specified in a *getsockopt(2)* or *setsockopt(2)* call.

109 EPROTONOSUPPORT Protocol not supported
The support for the socket type has not been configured into the system or no implementation for it exists.

110 ESOCKTNOSUPPORT Socket type unsupported
The support for the socket type has not been configured into the system or nor implementation for it exists.

111 EOPNOTSUPP Operation not supported on socket
For example, trying to *accept* a connection on a datagram socket.

112 EPRNOSUPPORT Protocol family unsupported
The protocol family has not been configured into the system or no implementation for it exists.

113 EAFNOSUPPORT Address family unsupported by protocol family
An address incompatible with the requested protocol was used. For example, you shouldn't necessarily expect to be able to use NS addresses with ARPA Internet protocols.

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

115 EADDRNOTAVAIL Can't assign requested address
Normally results from an attempt to create a socket with an address not on this machine. A socket operation encountered a dead network. A socket operation was attempted to an unreachable network. The host you were connected to crashed and rebooted. A connection abort was caused internal to your host machine. A connection was forcibly closed by a peer. This normally results from a loss of the connection on the remote socket due to a timeout or a reboot. an operation on a socket or pipe was not performed because the system lacked sufficient buffer space or because a queue was full. A *connect* request was made on an already connected socket; or, a *sendto* or *sendmsg* request on a connected socket specified a destination when already connected. A request to send or receive data was disallowed because the socket is not connected and (when sending on a datagram socket) no address was supplied. A request to send data was disallowed because the socket had already been shut down with a previous *shutdown(2)* call. A *connect* or *send* request failed because the connected party did not properly respond after a period of time. (The timeout period is dependent on the communication protocol.) 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. A socket operation failed because the destination host was down. A socket operation was attempted to an unreachable host. A path name lookup involved more than 8 symbolic links. A component of a path name exceeded 255 (MAXNAMELEN) characters, or an entire path name exceeded 1023 (MAXPATHLEN-1) characters. A directory with entries other than "." and ".." was supplied to a remove directory or rename call. A *write* on an ordinary file, the creation of a directory or symbolic link, or the creation of a directory entry failed because the user's quota of disk blocks was exhausted, or the allocation of an inode for a newly created file failed because the user's quota of inodes was exhausted. A client referenced an open file, when the file has been deleted. An attempt was made to mount a file remotely into a path that already has a remotely mounted component.

DEFINITIONS

Process ID Each active process in the system is uniquely identified by a positive integer called a process ID. The range of this ID is from 1 to 30,000.

Parent Process ID A new process is created by a currently active process [see *fork(2)*]. The parent process ID of a process is the process ID of its creator.

Process Group ID Each active process is a member of a process group that is identified by a positive integer called the process group ID. This ID is the process ID of the group leader. This grouping permits the signaling of related processes [see *kill(2)*].

Tty Group ID Each active process can be a member of a terminal group that is identified by a positive integer called the tty group ID. This grouping is used to terminate a group of related processes upon termination of one of the processes in the group [see *exit(2)* and *signal(2)*].

Real User ID and Real Group ID Each user allowed on the system is identified by a positive integer (0 to 65535) called a real user ID.

Each user is also a member of a group. The group is identified by a positive integer called the real group ID.

An active process has a real user ID and real group ID that are set to the real user ID and real group ID, respectively, of the user responsible for the creation of the process.

Effective User ID and Effective Group ID An active process has an effective user ID and an effective group ID that are used to determine file access permissions (see below). The effective user ID and effective group ID are equal to the process's real user ID and real group ID respectively, unless the process or one of its ancestors evolved from a file that had the set-user-ID bit or set-group ID bit set [see *exec(2)*].

Super-user A process is recognized as a *super-user* process and is granted special privileges, such as immunity from file permissions, if its effective user ID is 0.

Special Processes The processes with a process ID of 0 and a process ID of 1 are special processes and are referred to as *proc0* and *proc1*.

Proc0 is the scheduler. *Proc1* is the initialization process (*init*). Proc1 is the ancestor of every other process in the system and is used to control the process structure. *Proc2* is the daemon process that ages and steals pages (*uhand*). *Proc3* is the daemon process that flushes delayed writes.

Descriptor A descriptor is an integer used to do I/O on a file. The value of a descriptor is from 0 to (NOFILES - 1). A process may have no more than NOFILES file descriptors open simultaneously. A file descriptor is returned by system calls such as *open(2)*, *pipe(2)*, or *socketpair(2)*. The descriptor is used as an argument by calls such as *read(2)*, *write(2)*, *ioctl(2)*, and *close(2)*.

System V File Name Types Names consisting of 1 to 14 characters may be used to name an ordinary file, special file or directory.

These characters may be selected from the set of all character values excluding 0 (null) and the ASCII code for **/** (slash).

Note that it is generally unwise to use *, ?, [, or] as part of file names because of the special meaning attached to these characters by the shell [see *sh(1)*]. Although permitted, the use of unprintable characters in file names should be avoided.

BSD File Name Types Names consisting of up to 255 (MAXNAMELEN) characters can be used to name an ordinary file, special file or directory.

These characters can be selected from the set of all ASCII characters, excluding 0 (null) and the ASCII code for / (slash). (The parity bit, bit 8, must be 0.)

NOTE: It is generally unwise to use *, ?, [or] as part of file names because of the special meaning attached to these characters by the shell.

Path Name and Path Prefix A path name is a null-terminated character string starting with an optional slash (**/**), followed by zero or more directory names separated by slashes, optionally followed by a file name.

If a path name begins with a slash, the path search begins at the *root* directory. Otherwise, the search begins from the current working directory.

A slash by itself names the root directory.

For System V-type file systems, unless specifically stated otherwise, the null path name is treated as if it named a non-existent file. For BSD-type files systems, a null path name refers to the current directory.

Directory Directory entries are called links. By convention, a directory contains at least two links, . and .., referred to as *dot* and *dot-dot* respectively. Dot refers to the directory itself and dot-dot refers to its parent directory.

Root Directory and Current Working Directory Each process has associated with it a concept of a root directory and a current working directory for the purpose of resolving path name searches. The root directory of a process need not be the root directory of the root file system.

File Access Permissions Every file in the file system has a set of access permissions. These permissions are used to determine whether a process can perform a requested operation on the file (such as opening a file for writing). Access permissions are established at the time a file is created. They can be changed at some later time through the *chmod(2)* call.

File access is broken down according to whether a file can be: read, written, or executed. Directory files use the execute permission to control if the directory can be searched.

File access permissions are interpreted by the system as they apply to three different classes of users: the owner of the file, those users in the file's group, anyone else. Every file has an independent set of access permissions for each of these classes. When an access check is made, the system decides if permission should be granted by checking the access information applicable to the caller.

Read, write, and execute/search permissions on a file are granted to a process if:

The process's effective user ID is that of the super-user.

The process's effective user ID matches the user ID of the owner of the file and the owner permissions allow the access.

The process's effective user ID does not match the user ID of the owner of the file, and either the process's effective group ID matches the group ID of the file, or the group ID of the file is in the process's group access list, and the group permissions allow the access.

Neither the effective user ID nor effective group ID and group access list of the process match the corresponding user ID and group ID of the file, but the permissions for "other users" allow access.

Otherwise, permission is denied.

Sockets and Address Families A socket is an endpoint for communication between processes. Each socket has queues for sending and receiving data.

Sockets are typed according to their communications properties. These properties include whether messages sent and received at a socket require the name of the partner, whether communication is reliable, the format used in naming message recipients, etc.

Each instance of the system supports some collection of socket type; consult *socket(2)* for more information about the type available and their properties.

Each instance of the system supports some number of sets of communications protocols. Each protocol set support addresses of a certain format. An Address Family is the set addresses for a specific group of protocols. Each socket has an address chosen from the address family in which the socket was created.

Message Queue Identifier A message queue identifier (msqid) is a unique positive integer created by a *msgget(2)* system call. Each msqid has a message queue and a data structure associated with it. The data structure is referred to as *msqid_ds* and contains the following members:

```
struct   ipc_perm msg_perm;
struct   msg *msg_first;
struct   msg *msg_last;
ushort   msg_cbytes;
ushort   msg_qnum;
ushort   msg_qbytes;
ushort   msg_lspid;
ushort   msg_lrpid;
time_t   msg_stime;
time_t   msg_rtime;
time_t   msg_ctime;
```

msg_perm is an ipc_perm structure that specifies the message operation permission (see below). This structure includes the following members:

```
ushort   cuid;      /* creator user id */
ushort   cgid;      /* creator group id */
ushort   uid;       /* user id */
ushort   gid;       /* group id */
ushort   mode;      /* r/w permission */
ushort   seq;       /* slot usage sequence # */
key_t    key;       /* key */
```

msg *msg_first
is a pointer to the first message on the queue.

msg *msg_last
is a pointer to the last message on the queue.

msg_cbytes
is the current number of bytes on the queue.

msg_qnum
is the number of messages currently on the queue.

msg_qbytes
is the maximum number of bytes allowed on the queue.

msg_lspid
is the process id of the last process that performed a *msgsnd* operation.

msg_lrpid
is the process id of the last process that performed a *msgrcv* operation.

msg_stime
is the time of the last *msgsnd* operation.

msg_rtime
is the time of the last *msgrcv* operation

msg_ctime
is the time of the last *msgctl(2)* operation that changed a member of the above structure.

Message Operation Permissions In the *msgop(2)* and *msgctl(2)* system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed, interpreted as follows:

00400	Read by user
00200	Write by user
00040	Read by group
00020	Write by group
00004	Read by others
00002	Write by others

Read and write permissions on a msqid are granted to a process if one or more of the following are true:

The effective user ID of the process is super-user.

The effective user ID of the process matches **msg_perm.cuid** or **msg_perm.uid** in the data structure associated with *msqid* and the appropriate bit of the "user" portion (0600) of **msg_perm.mode** is set.

The effective group ID of the process matches **msg_perm.cgid** or **msg_perm.gid** and the appropriate bit of the "group" portion (060) of **msg_perm.mode** is set.

The appropriate bit of the "other" portion (006) of **msg_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

Semaphore Identifier A semaphore identifier (semid) is a unique positive integer created by a *semget(2)* system call. Each semid has a set of semaphores and a data structure associated with it. The data structure is referred to as *semid_ds* and contains the following members:

```
struct   ipc_perm sem_perm;  /* operation permission struct */
struct   sem *sem_base;      /* ptr to first semaphore in set */
ushort   sem_nsems;          /* number of sems in set */
time_t   sem_otime;          /* last operation time */
time_t   sem_ctime;          /* last change time */
                             /* Times measured in secs since */
                             /* 00:00:00 GMT, Jan. 1, 1970 */
```

sem_perm is an ipc_perm structure that specifies the semaphore operation permission (see below). This structure includes the following members:

```
ushort   uid;    /* user id */
ushort   gid;    /* group id */
ushort   cuid;   /* creator user id */
ushort   cgid;   /* creator group id */
ushort   mode;   /* r/a permission */
ushort   seq;    /* slot usage sequence number */
key_t    key;    /* key */
```

sem_nsems
is equal to the number of semaphores in the set. Each semaphore in the set is referenced by a positive integer referred to as a *sem_num*. Sem_num values run sequentially from 0 to the value of sem_nsems minus 1.

sem_otime
is the time of the last *semop(2)* operation.

sem_ctime
is the time of the last *semctl(2)* operation that changed a member of the above structure.

A semaphore is a data structure called *sem* that contains the following members:

```
ushort   semval;    /* semaphore value */
short    sempid;    /* pid of last operation */
ushort   semncnt;   /* # awaiting semval > cval */
ushort   semzcnt;   /* # awaiting semval = 0 */
```

semval is a non-negative integer which is the actual value of the semaphore.

sempid
is equal to the process ID of the last process that performed a semaphore operation on this semaphore.

semncnt
is a count of the number of processes that are currently suspended awaiting this semaphore's semval to become greater than its current value.

semzcnt
is a count of the number of processes that are currently suspended awaiting this semaphore's semval to become zero.

Semaphore Operation Permissions In the *semop(2)* and *semctl(2)* system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed interpreted as follows:

00400	Read by user
00200	Alter by user
00040	Read by group
00020	Alter by group
00004	Read by others
00002	Alter by others

Read and alter permissions on a semid are granted to a process if one or more of the following are true:

The effective user ID of the process is super-user.

The effective user ID of the process matches **sem_perm.cuid** or **sem_perm.uid** in the data structure associated with *semid* and the appropriate bit of the "user" portion (0600) of **sem_perm.mode** is set.

The effective group ID of the process matches **sem_perm.cgid** or **sem_perm.gid** and the appropriate bit of the "group" portion (060) of **sem_perm.mode** is set.

The appropriate bit of the "other" portion (006) of **sem_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

Shared Memory Identifier A shared memory identifier (shmid) is a unique positive integer created by a *shmget(2)* system call. Each shmid has a segment of memory (referred to as a shared memory segment) and a data structure associated with it. (Note that these shared memory segments must be explicitly removed by the user after the last reference to them is removed.) The data structure is referred to as *shmid_ds* and contains the following members:

```
struct   ipc_perm shm_perm; /* operation permission struct */
int      shm_segsz;         /* size of segment */
struct   region *shm_reg;   /*ptr to region structure */
char     pad[4];            /* for swap compatibility */
ushort   shm_lpid;          /* pid of last operation */
ushort   shm_cpid;          /* creator pid */
ushort   shm_nattch;        /* number of current attaches */
ushort   shm_cnattch;       /* used only for shminfo */
time_t   shm_atime;         /* last attach time */
time_t   shm_dtime;         /* last detach time */
time_t   shm_ctime;         /* last change time */
                            /* Times measured in secs since */
                            /* 00:00:00 GMT, Jan. 1, 1970 */
```

shm_perm is an ipc_perm structure that specifies the shared memory operation permission (see below). This structure includes the following members:

```
ushort   cuid;      /* creator user id */
ushort   cgid;      /* creator group id */
ushort   uid;       /* user id */
ushort   gid;       /* group id */
ushort   mode;      /* r/w permission */
ushort   seq;       /* slot usage sequence # */
key_t    key;       /* key */
```

shm_segsz
specifies the size of the shared memory segment in bytes.

shm_cpid
is the process id of the process that created the shared memory identifier.

shm_lpid
is the process id of the last process that performed a *shmop(2)* operation.

shm_nattch
is the number of processes that currently have this segment attached.

shm_atime
is the time of the last *shmat* (see *shmop(2)*) operation,

shm_dtime
is the time of the last *shmdt* (see *shmop(2)*) operation.

shm_ctime
is the time of the last *shmctl(2)* operation that changed one of the members of the above structure.

Shared Memory Operation Permissions In the *shmop(2)* and *shmctl(2)* system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed interpreted as follows:

00400	Read by user
00200	Write by user
00040	Read by group
00020	Write by group
00004	Read by others
00002	Write by others

Read and write permissions on a shmid are granted to a process if one or more of the following are true:

The effective user ID of the process is super-user.

The effective user ID of the process matches **shm_perm.cuid** or **shm_perm.uid** in the data structure associated with *shmid* and the appropriate bit of the "user" portion (0600) of **shm_perm.mode** is set.

The effective group ID of the process matches **shm_perm.cgid** or **shm_perm.gid** and the appropriate bit of the "group" portion (060) of **shm_perm.mode** is set.

The appropriate bit of the "other" portion (06) of **shm_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

STREAMS A set of kernel mechanisms that support the development of network services and data communication *drivers*. It defines interface standards for character input/output within the kernel and between the kernel and user level processes. The STREAMS mechanism is composed of utility routines, kernel facilities and a set of data structures.

Stream A stream is a full-duplex data path within the kernel between a user process and driver routines. The primary components are a *stream head*, a *driver* and zero or more *modules* between the *stream head* and *driver*. A *stream* is analogous to a Shell pipeline except that data flow and processing are bidirectional.

Stream Head In a *stream*, the *stream head* is the end of the *stream* that provides the interface between the *stream* and a user process. The principle functions of the *stream head* are processing STREAMS-related system calls, and passing data and information between a user process and the *stream*.

Driver In a *stream*, the *driver* provides the interface between peripheral hardware and the *stream*. A *driver* can also be a pseudo-*driver*, such as a *multiplexor* or log *driver* [see *log(7)*], which is not associated with a hardware device.

Module A module is an entity containing processing routines for input and output data. It always exists in the middle of a *stream*, between the stream's head and a *driver*. A *module* is the STREAMS counterpart to the commands in a Shell pipeline except that a module contains a pair of functions which allow independent bidirectional (*downstream* and *upstream*) data flow and processing.

Downstream In a *stream*, the direction from *stream head* to *driver*.

Upstream In a *stream*, the direction from *driver* to *stream head*.

Message In a *stream*, one or more blocks of data or information, with associated STREAMS control structures. *Messages* can be of several defined types, which identify the *message* contents. *Messages* are the only means of transferring data and communicating within a *stream*.

Message Queue In a *stream*, a linked list of *messages* awaiting processing by a *module* or *driver*.

Read Queue In a *stream*, the *message queue* in a *module* or *driver* containing *messages* moving *upstream*.

Write Queue In a *stream*, the *message queue* in a *module* or *driver* containing *messages* moving *downstream*.

Multiplexor A multiplexor is a driver that allows *streams* associated with several user processes to be connected to a single *driver*, or several *drivers* to be connected to a single user process. STREAMS does not provide a general multiplexing *driver*, but does provide the facilities for constructing them, and for connecting multiplexed configurations of *streams*.

SEE ALSO

intro(3), perror(3).

NAME

ioctl – control device

SYNOPSIS

int ioctl (fildes, request, arg)
int fildes, request;

DESCRIPTION

ioctl performs a variety of control functions on devices and STREAMS. For non-STREAMS files, the functions performed by this call are *device-specific* control functions. The arguments *request* and *arg* are passed to the file designated by *fildes* and are interpreted by the device driver. This control is infrequently used on non-STREAMS devices, with the basic input/output functions performed through the *read*(2) and *write*(2) system calls.

For STREAMS files, specific functions are performed by the *ioctl* call as described in *streamio*(7).

fildes is an open file descriptor that refers to a device. *request* selects the control function to be performed and will depend on the device being addressed. *arg* represents additional information that is needed by this specific device to perform the requested function. The data type of *arg* depends upon the particular control request, but it is either an integer or a pointer to a device-specific data structure.

In addition to device-specific and STREAMS functions, generic functions are provided by more than one device driver, for example, the general terminal interface [see *termio*(7)].

ioctl will fail for any type of file if one or more of the following are true:

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

[ENOTTY] *fildes* is not associated with a device driver that accepts control functions.

[EINTR] A signal was caught during the *ioctl* system call.

ERRORS

ioctl will also fail if the device driver detects an error. In this case, the error is passed through *ioctl* without change to the caller. A particular driver might not have all of the following error cases. Other requests to device drivers will fail if one or more of the following are true:

[EFAULT] *request* requires a data transfer to or from a buffer pointed to by *arg*, but some part of the buffer is outside the process's allocated space.

[EINVAL] *request* or *arg* is not valid for this device.

[EIO] Some physical I/O error has occurred.

[ENXIO] The *request* and *arg* are valid for this device driver, but the service requested can not be performed on this particular subdevice.

[ENOLINK] *fildes* is on a remote machine and the link to that machine is no longer active.

STREAMS errors are described in *streamio*(7).

SEE ALSO

streamio(7), termio(7) in the *System Administrator's Reference Manual.*

DIAGNOSTICS

Upon successful completion, the value returned depends upon the device control function, but must be a non-negative integer. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

kill – send a signal to a process or a group of processes

SYNOPSIS

int kill (pid, sig)
int pid, 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 that is 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 effective user ID of the receiving process, unless the effective user ID of the sending process is super-user.

The processes with a process ID of 0 and a process ID of 1 are special processes [see *intro*(2)] and will be referred to below as *proc0* and *proc1*, respectively.

If *pid* is greater than zero, *sig* will be sent to the process whose process ID is equal to *pid*. *Pid* may equal 1.

If *pid* is 0, *sig* will be sent to all processes excluding *proc0* and *proc1* 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 super-user, *sig* will be sent to all processes excluding *proc0* and *proc1* whose real user ID{is equal to the effective user ID of the sender.

If *pid* is −1 and the effective user ID of the sender is super-user, *sig* will be sent to all processes excluding *proc0* and *proc1*.

If *pid* is negative but not −1, *sig* will be sent to all processes whose process group ID is equal to the absolute value of *pid*.

ERRORS

kill will fail and no signal will be sent if one or more of the following are true:

[EINVAL]	*sig* is not a valid signal number.
[EINVAL]	*sig* is SIGKILL and *pid* is 1 (proc1).
[ESRCH]	No process can be found corresponding to that specified by *pid*.
[EPERM]	The user ID of the sending process is not super-user, and its real or effective user ID does not match the real or effective user ID of the receiving process.

SEE ALSO

getpid(2), setpgrp(2), signal(2), sigset(2).
kill(1) in the *User's Reference Manual*.

DIAGNOSTICS

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

NAME

link – link to a file

SYNOPSIS

int link (path1, path2)
char *path1, *path2;

DESCRIPTION

path1 points to a path name naming an existing file. *path2* points to a path name naming the new directory entry to be created. *link* creates a new link (directory entry) for the existing file.

ERRORS

link will fail and no link will be created if one or more of the following are true:

[ENOTDIR]	A component of either path prefix is not a directory.
[ENOENT]	A component of either path prefix does not exist.
[EACCES]	A component of either path prefix denies search permission.
[ENOENT]	The file named by *path1* does not exist.
[EEXIST]	The link named by *path2* exists.
[EPERM]	The file named by *path1* is a directory and the effective user ID is not super-user.
[EXDEV]	The link named by *path2* and the file named by *path1* are on different logical devices (file systems).
[ENOENT]	*path2* points to a null path name.
[EACCES]	The requested link requires writing in a directory with a mode that denies write permission.
[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.
[EMLINK]	The maximum number of links to a file would be exceeded.
[EINTR]	A signal was caught during the *link* system call.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

SEE ALSO

unlink(2).

DIAGNOSTICS

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

NAME

listen – listen for connections on a socket

SYNOPSIS

listen(s, backlog)
int s, backlog;

DESCRIPTION

To accept connections, a socket is first created with *socket*(2), a willingness to accept incoming connections and a queue limit for incoming connections are specified with *listen*(2), and then the connections are accepted with *accept*(2). The *listen* call applies only to sockets of type SOCK_STREAM or SOCK_SEQPACKET.

The *backlog* parameter defines the maximum length the queue of pending connections may grow to. If a connection request arrives with the queue full the client may receive an error with an indication of ECONNREFUSED, or, if the underlying protocol supports retransmission, the request may be ignored so that retries may succeed.

RETURN VALUE

A 0 return value indicates success; –1 indicates an error.

ERRORS

The call fails if:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is not a socket.
[EOPNOTSUPP]	The socket is not of a type that supports the operation *listen*.

SEE ALSO

accept(2), connect(2), socket(2)

BUGS

The *backlog* is currently limited (silently) to 5.

NAME

lseek – move read/write file pointer

SYNOPSIS

long lseek (fildes, offset, whence)
int fildes;
long offset;
int whence;

DESCRIPTION

fildes is a file descriptor returned from a *creat*, *open*, *dup*, or *fcntl* system call. *lseek* sets the file pointer associated with *fildes* as follows:

If *whence* is 0, the pointer is set to *offset* bytes.

If *whence* is 1, the pointer is set to its current location plus *offset*.

If *whence* is 2, the pointer is set to the size of the file plus *offset*.

Upon successful completion, the resulting pointer location, as measured in bytes from the beginning of the file, is returned. Note that if *fildes* is a remote file descriptor and *offset* is negative, *lseek* will return the file pointer even if it is negative.

lseek will fail and the file pointer will remain unchanged if one or more of the following are true:

[EBADF] *fildes* is not an open file descriptor.

[ESPIPE] *fildes* is associated with a pipe or fifo.

[EINVAL and SIGSYS signal] *Whence* is not 0, 1, or 2.

[EINVAL] *fildes* is not a remote file descriptor, and the resulting file pointer would be negative.

Some devices are incapable of seeking. The value of the file pointer associated with such a device is undefined.

SEE ALSO

creat(2), dup(2), fcntl(2), open(2).

DIAGNOSTICS

Upon successful completion, a non-negative integer indicating the file pointer value is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

mkdir – make a directory

SYNOPSIS

int mkdir (path, mode)
char *path;
int mode;

DESCRIPTION

The routine *mkdir* creates a new directory withthe name *path*. The mode of the new directory is initialized from the *mode*. The protection part of the *mode* argument is modified by the process's mode mask [see *umask*(2)].

The directory's owner ID is set to the process's effective user ID. The directory's group ID is set to the process's effective group ID. The newly created directory is empty with the possible exception of entries for "." and "..". *mkdir* will fail and no directory will be created if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	A component of the path prefix does not exist.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.
[EACCES]	Either a component of the path prefix denies search permission or write permission is denied on the parent directory of the directory to be created.
[ENOENT]	The path is longer than the maximum allowed.
[EEXIST]	The named file already exists.
[EROFS]	The path prefix resides on a read-only file system.
[EFAULT]	*path* points outside the allocated address space of the process.
[EMLINK]	The maximum number of links to the parent directory would be exceeded.
[EIO]	An I/O error has occurred while accessing the file system.

DIAGNOSTICS

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

NAME

mknod – make a directory, or a special or ordinary file

SYNOPSIS

int mknod (path, mode, dev)
char *path;
int mode, dev;

DESCRIPTION

mknod creates a new file named by the path name pointed to by *path*. The mode of the new file is initialized from *mode*. Where the value of *mode* is interpreted as follows:

0170000 file type; one of the following:

0010000 fifo special
0020000 character special
0040000 directory
0060000 block special
0100000 or 0000000 ordinary file
012000 symbolic link

0004000 set user ID on execution
00020#0 set group ID on execution if # is **7**, **5**, **3**, or **1**
enable mandatory file/record locking if # is **6**, **4**, **2**, or **0**
0001000 save text image after execution
0000777 access permissions; constructed from the following:

0000400 read by owner
0000200 write by owner
0000100 execute (search on directory) by owner
0000070 read, write, execute (search) by group
0000007 read, write, execute (search) by others

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.

Values of *mode* other than those above are undefined and should not be used. The low-order 9 bits of *mode* are modified by the process's file mode creation mask: all bits set in the process's file mode creation mask are cleared [see *umask*(2)]. If *mode* indicates a block or character special file, *dev* is a configuration-dependent specification of a character or block I/O device. If *mode* does not indicate a block special or character special device, *dev* is ignored.

mknod may be invoked only by the super-user for file types other than FIFO special.

ERRORS

mknod will fail and the new file will not be created if one or more of the following are true:

[EPERM]	The effective user ID of the process is not super-user.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	A component of the path prefix does not exist.
[EROFS]	The directory in which the file is to be created is located on a read-only file system.
[EEXIST]	The named file exists.
[EFAULT]	*path* points outside the allocated address space of the process.

[ENOSPC]	No space is available.
[EINTR]	A signal was caught during the *mknod* system call.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

SEE ALSO

chmod(2), exec(2), umask(2), fs(4).
mkdir(1) in the *User's Reference Manual*.

DIAGNOSTICS

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

WARNING

If **mknod** is used to create a device in a remote directory (Remote File Sharing), the major and minor device numbers are interpreted by the server.

NAME

mmap, munmap – map or unmap pages of memory

SYNOPSIS

```
#include <sys/mman.h>
#include <sys/types.h>

mmap(addr, len, prot, share, fd, off)
caddr_t addr;
int len, prot, share, fd;
off_t off;

munmap (addr, len)
caddr_t addr;
int len;
```

DESCRIPTION

mmap maps pages of memory from the memory device associated with the file *fd* into the address space of the calling process, one page at a time. Pages are mapped from the memory device, beginning at *off,* and into the caller's address space, beginning at *addr,* and continuing for *len* bytes. *fd* is a file descriptor obtained by opening the device from which to map pages. Only character-special devices are currently supported.

share specifies whether modifications made to mapped-in copies of pages are to be kept "private" or are to be "shared" with other references. Currently, it must be set to MAP_SHARED.

The parameter *prot* specifies the read/write accessibility of the mapped pages. The *addr* and *len* parameters, and the sum of the current position in *fd* and *off* parameters, must be multiples of *pagesize* (found using the *getpagesize*(2) call). *malloc*(2) returns a properly aligned buffer if the request is for *pagesize* or larger bytes.

Currently, only 1 device may be mapped by a process. The file descriptor must be closed to allow mapping of another device.

All pages are automatically unmapped when *fd* is closed. Specific pages can be unmapped explicitly using *munmap.*

mmap can sometimes be used to install memory-mapped devices without writing a device driver. However, this does not always work. In particular, devices that are *mmap*'ed into user space and then accessed by user programs will see those accesses in user mode. If the device contains registers that must be accessed in supervisor mode, *mmap* cannot be used to drive it.

The virtual pages mapped by *mmap* may not be part of a region shared with any other process. If a caller attempts to mmap shared pages, the request will be rejected with error EACCES.

munmap unmaps previously mapped pages starting at *addr* and continuing for *len* bytes. Unmapped pages refer, once again, to private pages within the caller's address space. Unmapped pages are initialized to zero.

RETURN VALUE

Each call returns 0 on success, −1 on failure.

ERRORS

Both calls fail when:

EINVAL	The argument address or length is not a multiple of the page size as returned by *getpagesize*(2), or the length is negative.
EINVAL	The entire range of pages specified in the call is not part of data space.

In addition *mmap* fails when:

EINVAL	The specified *fd* does not refer to a character special device which supports mapping (e.g. a frame buffer).
EINVAL	The specified *fd* is not open for reading and read access is requested, or not open for writing when write access is requested.
EINVAL	The sharing mode was not specified as MAP_SHARED.
EINVAL	Another file mapped by *mmap* is open.
EACCESS	An attempt was made to share pages with another process.

SEE ALSO

getpagesize(2), munmap(2), close(2), malloc(2).

NAME

mount – mount a file system

SYNOPSIS

#include <sys/mount.h>

int mount (spec, dir, mflag, fstyp)
char *spec, *dir;
int mflag, fstyp;

DESCRIPTION

mount requests that a removable file system contained on the block special file identified by *spec* be mounted on the directory identified by *dir*. *spec* and *dir* are pointers to path names. *fstyp* is the file system type number. The *sysfs*(2) system call can be used to determine the file system type number. Note that if the MS_FSS flag bit of *mflag* is off, the file system type will default to the root file system type. Only if the bit is on will *fstyp* be used to indicate the file system type.

Upon successful completion, references to the file *dir* will refer to the root directory on the mounted file system.

The low-order bit of *mflag* is used to control write permission on the mounted file system; if **1**, writing is forbidden, otherwise writing is permitted according to individual file accessibility.

mount may be invoked only by the super-user. It is intended for use only by the *mount*(1M) utility.

ERRORS

mount will fail if one or more of the following are true:

[EPERM]	The effective user ID is not super-user.
[ENOENT]	Any of the named files does not exist.
[ENOTDIR]	A component of a path prefix is not a directory.
[EREMOTE]	*spec* is remote and cannot be mounted.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.
[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.
[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.
[EROFS]	*spec* is write protected and *mflag* requests write permission.
[ENOSPC]	The file system state in the super-block is not FsOKAY and *mflag* requests write permission.
[EINVAL]	The super block has an invalid magic number or the *fstyp* is invalid or *mflag* is not valid.

SEE ALSO

sysfs(2), umount(2), fs(4FFS), ffs(4S51K).
mount(1M) in the *System Administrator's Reference Manual*.

DIAGNOSTICS

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

NAME

msgctl – message control operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl (msqid, cmd, buf)
int msqid, 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 *intro*(2). {READ}

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 **msg_perm.cuid** or **msg_perm.uid** 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 **msg_perm.cuid** or **msg_perm.uid** in the data structure associated with *msqid*.

ERRORS

msgctl will fail if one or more of the following are 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 *intro*(2)].

[EPERM] *cmd* is equal to **IPC_RMID** or **IPC_SET**. The effective user ID of the calling process is not equal to that of super user, or to the value of **msg_perm.cuid** or **msg_perm.uid** 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.

SEE ALSO

intro(2), msgget(2), msgop(2).

DIAGNOSTICS

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

NAME

msgget – get message queue

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget (key, msgflg)
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 [see *intro*(2)] are created for *key* if one of the following are true:

key is equal to *IPC_PRIVATE*.

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.

ERRORS

msgget will fail if one or more of the following are true:

[EACCES]	A message queue identifier exists for *key*, but operation permission [see *intro*(2)] 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

intro(2), msgctl(2), msgop(2).

DIAGNOSTICS

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.

NAME

msgop – message operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd (msqid, msgp, msgsz, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz, msgflg;

int msgrcv (msqid, msgp, msgsz, msgtyp, msgflg)
int msqid;
struct msgbuf *msgp;
int 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*. {WRITE} *msgp* points to a structure containing the message. This structure is composed of the following members:

```
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 are true:

The number of bytes already on the queue is equal to **msg_qbytes** [see *intro*(2)].

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 will not be sent and the calling process will return immediately.

If (*msgflg* & **IPC_NOWAIT**) is "false", the calling process will suspend 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 that is to be caught. In this case the message is not sent and the calling process resumes execution in the manner prescribed in *signal*(2).

ERRORS

msgsnd will fail and no message will be sent if one or more of the following are true:

[EINVAL]	*msqid* is not a valid message queue identifier.
[EACCES]	Operation permission is denied to the calling process [see *intro*(2)].

[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.

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid* [see intro (2)].

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*. {READ} 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* & **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:

If *msgtyp* is equal to 0, the first message on the queue is received.

If *msgtyp* is greater than 0, the first message of type *msgtyp* is received.

If *msgtyp* is less than 0, the 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:

If (*msgflg* & **IPC_NOWAIT**) is "true", the calling process will return immediately with a return value of −1 and *errno* set to ENOMSG.

If (*msgflg* & **IPC_NOWAIT**) is "false", the calling process will suspend 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*(2).

ERRORS

msgrcv will fail and no message will be received if one or more of the following are 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 (*msgtyp* & **IPC_NOWAIT**) is "true".

[EFAULT] *msgp* points to an illegal address.

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid* [see intro (2)].

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.

SEE ALSO

intro(2), msgctl(2), msgget(2), signal(2).

DIAGNOSTICS

If *msgsnd* or *msgrcv* return due to the receipt of a signal, a value of −1 is returned to the calling process and *errno* is set to EINTR. If they return due to removal of *msqid* from the system, a value of −1 is returned and *errno* is set to EIDRM.

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.

NAME

nfsmount – mount an NFS file system

SYNOPSIS

nfsmount(argp, dir, readonly)
struct nfs_args *argp;
char *dir;
int readonly;

DESCRIPTION

nfsmount mounts an NFS file system on the directory *dir*. *argp* points to a structure of the following form:

```
#include        <netinet/in.h>
#include        <nfs/nfs_export.h>
struct nfs_args {
        struct sockaddr_in  *addr;  /* file server address */
        fhandle_t   *fh;      /* File handle to be mounted */
        int         flags;    /* flags */
        int         wsize;    /* write size in bytes */
        int         rsize;    /* read size in bytes */
        int         timeo;    /* initial timeout in .1 secs */
        int         retrans;  /* times to retry send */
};
```

The *readonly* argument determines whether the file system can be written on; if it is 0 writing is allowed; if non-zero no writing is done.

RETURN VALUE

nfsmount returns 0 if the action occurred, −1 if some error occurred.

ERRORS

nfsmount will fail when one of the following occurs:

[EPERM]	The caller is not the super-user.
[ENAMETOOLONG]	The path name for *dir* is too long.
[ELOOP]	*dir* contains a symbolic link loop.
[ETIMEDOUT]	The server at *addr* is not accessible. This can only happen if the *NFSMNT_SOFT* bit is set in *argp->flags*.
[ENOTDIR]	A component of the path prefix in *dir* is not a directory.
[EBUSY]	Another process currently holds a reference to *argp->fh*.
[EFAULT]	*argp*, *argp->addr*, or *argp->fh* does not point within the user's address space.
[EPFNOSUPPORT]	NFS is not supported by the protocol family of *argp->addr*.
[EINVAL]	One of *argp->timeo*, *argp->rsize*, or *argp->wsize* is not positive, or *argp->retrans* is negative.
[EINVAL]	An invalid or malformed response was returned to a remote procedure call (RPC) to the server named by argp.

SEE ALSO

mount(2), *umount*(2)
mount(1M) in the *System Administrator's Guide.*

NAME

nfssvc, async_daemon – NFS daemons

SYNOPSIS

nfssvc(sock)
int sock;

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 will return 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

These two system calls allow kernel processes to have user context.

SEE ALSO

mountd(1M)

ORIGIN

Sun Microsystems

NAME

nice – change priority of a process

SYNOPSIS

int nice (incr)
int incr;

DESCRIPTION

nice adds the value of *incr* to the nice value of the calling process. A process's *nice value* is a non-negative number for which a more positive value results in lower CPU priority.

A maximum nice value of ((2* NZERO)-1) and a minimum nice value of 0 are imposed by the system. (The default nice value is NZERO being set to the corresponding limit.

[EPERM] *nice* will fail and not change the nice value if *incr* is negative or greater than or equal to 2*NZERO and the effective user ID of the calling process is not super-user.

SEE ALSO

exec(2).
nice(1) in the *User's Reference Manual*.

DIAGNOSTICS

Upon successful completion, *nice* returns the new nice value minus NZERO. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

open – open for reading or writing

SYNOPSIS

```
#include <fcntl.h>
int open (path, oflag [, mode] )
char *path;
int oflag, mode;
```

DESCRIPTION

path points to a path name naming a file. *open* opens a file descriptor for the named file and sets the file status flags according to the value of *oflag*. For non-STREAMS [see *intro*(2)] files, *oflag* values are constructed by or-ing flags from the following list (only one of the first three flags below may be used):

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

O_NDELAY This flag may 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 will return without delay. An *open* for writing-only will return an error if no process currently has the file open for reading.

If O_NDELAY is clear:

An *open* for reading-only will block until a process opens the file for writing. An *open* for writing-only will block until a process opens the file for reading.

When opening a file associated with a communication line:

If O_NDELAY is set:

The open will return without waiting for carrier.

If O_NDELAY is clear:

The open will block until carrier is present.

O_APPEND If set, the file pointer will be set to the end of the file prior to each write.

O_SYNC When opening a regular file, this flag affects subsequent writes. If set, each *write*(2) will wait for both the file data and file status to be physically updated.

O_CREAT If the file exists, this flag has no effect. 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, and the low-order 12 bits of the file mode are set to the value of *mode* modified as follows [see *creat*(2)]:

All bits set in the file mode creation mask of the process are cleared [see *umask*(2)].

The "save text image after execution bit" of the mode is cleared [see *chmod*(2)].

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* will fail if the file exists.

When opening a STREAMS file, *oflag* may be constructed from O_NDELAY or-ed with either O_RDONLY, O_WRONLY or O_RDWR. Other flag values are not applicable to STREAMS devices and have no effect on them. The value of O_NDELAY affects the operation of STREAMS drivers and certain system calls [see *read*(2), *getmsg*(2), *putmsg*(2) and *write*(2)]. For drivers, the implementation of O_NDELAY is device-specific. Each STREAMS device driver may treat this option differently.

Certain flag values can be set following *open* as described in *fcntl*(2).

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)].

The named file is opened unless one or more of the following are true:

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

[EACCES] *oflag* permission is denied for the named file.

[EAGAIN] The file exists, mandatory file/record locking is set, and there are outstanding record locks on the file [see *chmod* (2)].

[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.

[EIO] A hangup or error occurred during a STREAMS *open*.

[EISDIR] The named file is a directory and *oflag* is write or read/write.

[EMFILE] NOFILES file descriptors are currently open.

[EMULTIHOP] Components of *path* require hopping to multiple remote machines.

[ENFILE] The system file table is full.

[ENOENT] O_CREAT is not set and the named file does not exist.

[ENOLINK] *path* points to a remote machine, and the link to that machine is no longer active.

[ENOMEM] The system is unable to allocate a send descriptor.

[ENOSPC] O_CREAT and O_EXCL are set, and the file system is out of inodes.

[ENOSR] Unable to allocate a *stream*.

[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.

[ENXIO] O_NDELAY is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading.

[ENXIO] A STREAMS module or driver open routine failed.

[EROFS] The named file resides on a read-only file system and *oflag* is write or read/write.

[ETXTBSY] The file is a pure procedure (shared text) file that is being executed and *oflag* is write or read/write.

SEE ALSO

chmod(2), close(2), creat(2), dup(2), fcntl(2), intro(2), lseek(2), read(2), getmsg(2), putmsg(2), umask(2), write(2).

DIAGNOSTICS

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

NAME

pause – suspend process until signal

SYNOPSIS

pause ()

DESCRIPTION

pause suspends the calling process until it receives a signal. The signal must be one that is not currently set to be ignored by the calling process.

If the signal causes termination of the calling process, *pause* will not return.

If the signal is *caught* by the calling process and control is returned from the signal-catching function [see *signal*(2)], the calling process resumes execution from the point of suspension; with a return value of −1 from *pause* and *errno* set to EINTR.

SEE ALSO

alarm(2), kill(2), signal(2), sigpause(2), wait(2).

NAME

pipe – create an interprocess channel

SYNOPSIS

int pipe (fildes)
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.

Up to 5120 bytes of data are buffered by the pipe before the writing process is blocked. A read only file descriptor *fildes*[0] accesses the data written to *fildes*[1] on a first-in-first-out (FIFO) basis.

ERRORS

pipe will fail if:

[EMFILE]	NOFILES file descriptors are currently open.
[ENFILE]	The system file table is full.

SEE ALSO

read(2), write(2).
sh(1) in the *User's Reference Manual.*

DIAGNOSTICS

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

NAME

plock – lock process, text, or data in memory

SYNOPSIS

#include <sys/lock.h>

int plock (op)
int op;

DESCRIPTION

plock allows the calling process to lock its text segment (text lock), its data segment (data lock), or both its text and data segments (process lock) into memory. Locked segments are immune to all routine swapping. *plock* also allows these segments to be unlocked. The effective user ID of the calling process must be super-user to use this call. *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

ERRORS

plock will fail and not perform the requested operation if one or more of the following are true:

[EPERM]	The effective user ID of the calling process is not super-user.
[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 a process lock already exists on the calling process.
[EINVAL]	*op* is equal to **DATLOCK** and a data lock, or a process lock already exists on the calling process.
[EINVAL]	*op* is equal to **UNLOCK** and no type of lock exists on the calling process.
[EAGAIN]	Not enough memory.

SEE ALSO

exec(2), exit(2), fork(2).

DIAGNOSTICS

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

NAME

poll – STREAMS input/output multiplexing

SYNOPSIS

```
#include <stropts.h>
#include <poll.h>

int poll(fds, nfds, timeout)
struct pollfd fds[];
unsigned long nfds;
int timeout;
```

DESCRIPTION

poll provides users with a mechanism for multiplexing input/output over a set of file descriptors that reference open *streams* [see *intro*(2)]. *poll* identifies those *streams* on which a user can send or receive messages, or on which certain events have occurred. A user can receive messages using *read*(2) or *getmsg*(2) and can send messages using *write*(2) and *putmsg*(2). Certain *ioctl*(2) calls, such as I_RECVFD and I_SENDFD [see *streamio*(7)], can also be used to receive and send messages.

fds specifies the file descriptors to be examined and the events of interest for each file descriptor. It is a pointer to an array with one element for each open file descriptor of interest. The array's elements are *pollfd* structures which contain the following members:

```
int fd;          /* file descriptor */
short events;    /* requested events */
short revents;   /* returned events */
```

where *fd* specifies an open file descriptor and *events* and *revents* are bitmasks constructed by or-ing any combination of the following event flags:

POLLIN	A non-priority or file descriptor passing message (see I_RECVFD) is present on the *stream head* read queue. This flag is set even if the message is of zero length. In *revents*, this flag is mutually exclusive with POLLPRI.
POLLPRI	A priority message is present on the *stream head* read queue. This flag is set even if the message is of zero length. In *revents*, this flag is mutually exclusive with POLLIN.
POLLOUT	The first downstream write queue in the *stream* is not full. Priority control messages can be sent (see *putmsg*) at any time.
POLLERR	An error message has arrived at the *stream head*. This flag is only valid in the *revents* bitmask; it is not used in the *events* field.
POLLHUP	A hangup has occurred on the *stream*. This event and POLLOUT are mutually exclusive; a *stream* can never be writable if a hangup has occurred. However, this event and POLLIN or POLLPRI are not mutually exclusive. This flag is only valid in the *revents* bitmask; it is not used in the *events* field.
POLLNVAL	The specified *fd* value does not belong to an open *stream*. This flag is only valid in the *revents* field; it is not used in the *events* field.

For each element of the array pointed to by *fds*, *poll* examines the given file descriptor for the event(s) specified in *events*. The number of file descriptors to be examined is specified by *nfds*. If *nfds* exceeds NOFILES, the system limit of open files [see *ulimit*(2)], *poll* will fail.

If the value *fd* is less than zero, *events* is ignored and *revents* is set to 0 in that entry on return from *poll*.

The results of the *poll* query are stored in the *revents* field in the *pollfd* structure. Bits are set in the *revents* bitmask to indicate which of the requested events are true. If none are true, none of the specified bits is set in *revents* when the *poll* call returns. The event flags POLLHUP, POLLERR and POLLNVAL are always set in *revents* if the conditions they indicate are true; this occurs even though these flags were not present in *events*.

If none of the defined events have occurred on any selected file descriptor, *poll* waits at least *timeout* msec for an event to occur on any of the selected file descriptors. On a computer where millisecond timing accuracy is not available, *timeout* is rounded up to the nearest legal value available on that system. If the value *timeout* is 0, *poll* returns immediately. If the value of *timeout* is -1, *poll* blocks until a requested event occurs or until the call is interrupted. *poll* is not affected by the O_NDELAY flag.

ERRORS

poll fails if one or more of the following are true:

[EAGAIN]	Allocation of internal data structures failed but request should be attempted again.
[EFAULT]	Some argument points outside the allocated address space.
[EINTR]	A signal was caught during the *poll* system call.
[EINVAL]	The argument *nfds* is less than zero, or *nfds* is greater than NOFILES.

SEE ALSO

intro(2), read(2), getmsg(2), putmsg(2), write(2).
streamio(7) in the *System Administrator's Reference Manual*.
STREAMS Primer.
STREAMS Programmer's Guide.

DIAGNOSTICS

Upon successful completion, a non-negative value is returned. A positive value indicates the total number of file descriptors that has been selected (i.e., file descriptors for which the *revents* field is non-zero). A value of 0 indicates that the call timed out and no file descriptors have been selected. Upon failure, a value of -1 is returned and *errno* is set to indicate the error.

NAME

profil – execution time profile

SYNOPSIS

void profil (buff, bufsiz, offset, scale)
char *buff;
int bufsiz, offset, scale;

DESCRIPTION

buff points to an area of core whose length (in bytes) is given by *bufsiz*. After this call, the user's program counter (pc) is examined each clock tick. Then the value of *offset* is subtracted from it, and the remainder multiplied by *scale*. If the resulting number corresponds to an entry inside *buff*, that entry is incremented. An entry is defined as a series of bytes with length *sizeof(short)*.

The scale is interpreted as an unsigned, fixed-point fraction with binary point at the left: 0177777 (octal) gives a 1-1 mapping of pc's to entries in *buff*; 077777 (octal) maps each pair of instruction entries 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 an *exec* is executed, but remains on in child and parent both after a *fork*. Profiling will be turned off if an update in *buff* would cause a memory fault.

SEE ALSO

prof(1), times(2), monitor(3C).

DIAGNOSTICS

Not defined.

NAME

ptrace – process trace

SYNOPSIS

#include <signal.h>
#include <sys/ptrace.h>

ptrace(request, pid, addr, data)
int request, pid, *addr, data;

DESCRIPTION

ptrace provides a means by which a process may control the execution of another process, and examine and change its core image. Its primary use is for the implementation of breakpoint debugging. There are four arguments whose interpretation depends on a *request* argument. Generally, *pid* is the process ID of the traced process. A process being traced behaves normally until it encounters some signal whether internally generated like "illegal instruction" or externally generated like "interrupt". See *sigset*(2) or *signal*(2) for the list.

Upon encountering a signal the traced process enters a stopped state and its tracing process is notified via *wait*(2). If the the traced process stops with a SIGTRAP the process may have been stopped for a number of reasons. Two status words addressable as registers in the traced process's uarea qualify SIGTRAP ss: TRAPCAUSE , which contains the cause of the trap, and TRAPINFO , which contains extra information concerning the trap.

When the traced process is in the stopped state, its core image can be examined and modified using *ptrace*. If desired, another *ptrace* request can then cause the traced process either to terminate or to continue, possibly ignoring the signal.

The value of the *request* argument determines the precise action of the call:

0 — This request is the only one that may be used by a child process; it may declare that it is to be traced by its parent. All other arguments are ignored. Peculiar results will ensue if the parent does not expect to trace the child.

1,2 — The word in the traced process's address space at *addr* is returned. If I and D space are separated (e.g. historically on a pdp-11), request 1 indicates I space, 2 D space. *addr* must be 20-byte aligned. The traced process must be stopped. The input *data* is ignored.

3 — The word of the system's per-process data area corresponding to *addr* is returned. *addr* is a constant defined in ptrace.h This space contains the registers and other information about the process; the constants correspond to fields in the *user* structure in the system.

4,5 — The given *data* is written at the word in the process's address space corresponding to *addr,* which must be 20-byte aligned. The old value at the address is returned. If I and D space are separated, request 20 indicates I space, 5 D space. Attempts to write in pure procedure fail if another process is executing the same file.

6 — The process's system data is written, as it is read with request 3. Only a few locations can be written in this way: the general registers, the floating point status and registers, and certain bits of the processor status word. The old value at the address is returned.

7 — The *data* argument is taken as a signal number and the traced process's execution continues at location *addr* as if it had incurred that signal. Normally the signal number will be either 0 to indicate that the signal that caused the stop should be ignored, or that value fetched out of the

process's image indicating which signal caused the stop. If *addr* is (int *)1 then execution continues from where it stopped.

8 The traced process terminates.

9 Execution continues as in request 7; however, as soon as possible after execution of at least one instruction, execution stops again. The signal number from the stop is SIGTRAP. TRAPCAUSE will contain CAUSESINGLE. This is part of the mechanism for implementing breakpoints.

As indicated, these calls (except for request 0 and 20) can be used only when the subject process has stopped. The *wait* call is used to determine when a process stops; in such a case the "termination" status returned by *wait* has the value 0177 to indicate stoppage rather than genuine termination. If multiple processes are being traced, wait can be called multiple times and will return the status for the next stopped or terminated child or traced process.

To forestall possible fraud, *ptrace* inhibits the set-user-id and set-group-id facilities on subsequent *exec*(2) calls. If a traced process calls *execve*, it will stop before executing the first instruction of the new image showing signal SIGTRAP. In this case TRAPCAUSE will contain CAUSEEXEC and TRAPINFO will not contain anything interesting. If a traced process execs again, the same thing will happen.

If a traced process forks, both parent and child will be traced. Breakpoints from the parent will not be copied into the child. At the time of the fork, the child will be stopped with a SIGTRAP. The tracing process may then terminate the trace if desired. TRAPCAUSE will contain CAUSEFORK and TRAPINFO
will contain the pid of its parent.

RETURN VALUE

A 0 value is returned if the call succeeds. If the call fails then a −1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

[EINVAL]	The request code is invalid.
[EINVAL]	The specified process does not exist.
[EINVAL]	The given signal number is invalid.
[EFAULT]	The specified address is out of bounds.
[EPERM]	The specified process cannot be traced.

SEE ALSO

wait(2), sigset(2), signal(2).

BUGS

ptrace is unique and arcane; it should be replaced with a special file which can be opened and read and written. The control functions could then be implemented with *ioctl*(2) calls on this file. This would be simpler to understand and have much higher performance.

The request 0 call should be able to specify signals which are to be treated normally and not cause a stop. In this way, for example, programs with simulated floating point (which use "illegal instruction" signals at a very high rate) could be efficiently debugged.

The error indication, −1, is a legitimate function value; *errno,* see *intro*(2), can be used to disambiguate.

It should be possible to stop a process on occurrence of a system call; in this way a completely controlled environment could be provided.

NAME

putmsg – send a message on a stream

SYNOPSIS

```
#include <stropts.h>

int putmsg (fd, ctlptr, dataptr, flags)
int fd;
struct strbuf *ctlptr;
struct strbuf *dataptr;
int flags;
```

DESCRIPTION

putmsg creates a message [see *intro*(2)] from user specified buffer(s) and sends the message to a STREAMS file. The message may contain either a data part, a control part or both. The data and control parts to be sent are distinguished by placement in separate buffers, as described below. The semantics of each part is defined by the STREAMS module that receives the message.

fd specifies a file descriptor referencing an open *stream*. *ctlptr* and *dataptr* each point to a *strbuf* structure which contains the following members:

```
int maxlen;     /* not used */
int len;        /* length of data */
char *buf;      /* ptr to buffer */
```

ctlptr points to the structure describing the control part, if any, to be included in the message. The *buf* field in the *strbuf* structure points to the buffer where the control information resides, and the *len* field indicates the number of bytes to be sent. The *maxlen* field is not used in *putmsg* [see *getmsg*(2)]. In a similar manner, *dataptr* specifies the data, if any, to be included in the message. *flags* may be set to the values 0 or RS_HIPRI and is used as described below.

To send the data part of a message, *dataptr* must be non-NULL and the *len* field of *dataptr* must have a value of 0 or greater. To send the control part of a message, the corresponding values must be set for *ctlptr*. No data (control) part will be sent if either *dataptr* (*ctlptr*) is NULL or the *len* field of *dataptr* (*ctlptr*) is set to -1.

If a control part is specified, and *flags* is set to RS_HIPRI, a *priority* message is sent. If *flags* is set to 0, a non-priority message is sent. If no control part is specified, and *flags* is set to RS_HIPRI, *putmsg* fails and sets *errno* to EINVAL. If no control part and no data part are specified, and *flags* is set to 0, no message is sent, and 0 is returned.

For non-priority messages, *putmsg* will block if the *stream* write queue is full due to internal flow control conditions. For priority messages, *putmsg* does not block on this condition. For non-priority messages, *putmsg* does not block when the write queue is full and O_NDELAY is set. Instead, it fails and sets *errno* to EAGAIN.

putmsg also blocks, unless prevented by lack of internal resources, waiting for the availability of message blocks in the *stream*, regardless of priority or whether O_NDELAY has been specified. No partial message is sent.

ERRORS

putmsg fails if one or more of the following are true:

[EAGAIN]	A non-priority message was specified, the O_NDELAY flag is set and the *stream* write queue is full due to internal flow control conditions.
[EAGAIN]	Buffers could not be allocated for the message that was to be created.

[EBADF]	*fd* is not a valid file descriptor open for writing.
[EFAULT]	*ctlptr* or *dataptr* points outside the allocated address space.
[EINTR]	A signal was caught during the *putmsg* system call.
[EINVAL]	An undefined value was specified in *flags*, or *flags* is set to RS_HIPRI and no control part was supplied.
[EINVAL]	The *stream* referenced by *fd* is linked below a multiplexor.
[ENOSTR]	A *stream* is not associated with *fd*.
[ENXIO]	A hangup condition was generated downstream for the specified *stream*.
[ERANGE]	The size of the data part of the message does not fall within the range specified by the maximum and minimum packet sizes of the topmost *stream* module. This value is also returned if the control part of the message is larger than the maximum configured size of the control part of a message, or if the data part of a message is larger than the maximum configured size of the data part of a message.

A *putmsg* also fails if a STREAMS error message had been processed by the *stream* head before the call to *putmsg*. The error returned is the value contained in the STREAMS error message.

SEE ALSO

intro(2), read(2), getmsg(2), poll(2), write(2).
STREAMS Primer.
STREAMS Programmer's Guide.

DIAGNOSTICS

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

NAME

read – read from file

SYNOPSIS

int read (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;

DESCRIPTION

fildes is a file descriptor obtained from a *creat*(2), *open*(2), *dup*(2), *fcntl*(2), or *pipe*(2) system call.

read attempts to read *nbyte* bytes from the file associated with *fildes* into the buffer pointed to by *buf*.

On devices capable of seeking, the *read* starts at a position in the file given by the file pointer associated with *fildes*. Upon return from *read*, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. The value of a file pointer associated with such a file is undefined.

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 if the number of bytes left in the file is less than *nbyte* bytes. A value of 0 is returned when an end-of-file has been reached.

A *read* from a STREAMS [see *intro*(2)] file can operate in three different modes: "byte-stream" mode, "message-nondiscard" mode, and "message-discard" mode. The default is byte-stream mode. This can be changed using the I_SRDOPT *ioctl* request [see *streamio*(7)], and can be tested with the I_GRDOPT *ioctl*. In byte-stream mode, *read* will retrieve data from the *stream* until it has retrieved *nbyte* bytes, or until there is no more data to be retrieved. Byte-stream mode ignores message boundaries.

In STREAMS message-nondiscard mode, *read* retrieves data until it has read *nbyte* bytes, or until it reaches a message boundary. If the *read* does not retrieve all the data in a message, the remaining data are replaced on the *stream*, and can be retrieved by the next *read* or *getmsg*(2) call. Message-discard mode also retrieves data until it has retrieved *nbyte* bytes, or it reaches a message boundary. However, unread data remaining in a message after the *read* returns are discarded, and are not available for a subsequent *read* or *getmsg*.

When attempting to read from a regular file with mandatory file/record locking set [see *chmod*(2)], and there is a blocking (i.e. owned by another process) write lock on the segment of the file to be read:

If O_NDELAY is set, the read will return a −1 and set errno to EAGAIN.

If O_NDELAY is clear, the read will sleep until the blocking record lock is removed.

When attempting to read from an empty pipe (or FIFO):

If O_NDELAY is set, the read will return a 0.

If O_NDELAY is clear, the read will block 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_NDELAY is set, the read will return a 0.

If O_NDELAY is clear, the read will block until data becomes available.

When attempting to read a file associated with a *stream* that has no data currently available:

If O_NDELAY is set, the read will return a −1 and set errno to EAGAIN.

If O_NDELAY is clear, the read will block until data becomes available.

When reading from a STREAMS file, handling of zero-byte messages is determined by the current read mode setting. In byte-stream mode, *read* accepts data until it has read *nbyte* bytes, or until there is no more data to read, or until a zero-byte message block is encountered. *read*

then returns the number of bytes read, and places the zero-byte message back on the *stream* to be retrieved by the next *read* or *getmsg*. In the two other modes, a zero-byte message returns a value of 0 and the message is removed from the *stream*. When a zero-byte message is read as the first message on a *stream*, a value of 0 is returned regardless of the read mode.

A *read* from a STREAMS file can only process data messages. It cannot process any type of protocol message and will fail if a protocol message is encountered at the *stream head*.

ERRORS

read will fail if one or more of the following are true:

[EAGAIN]	Mandatory file/record locking was set, O_NDELAY was set, and there was a blocking record lock.
[EAGAIN]	Total amount of system memory available when reading via raw IO is temporarily insufficient.
[EAGAIN]	No message waiting to be read on a *stream* and O_NDELAY flag set.
[EBADF]	*fildes* is not a valid file descriptor open for reading.
[EBADMSG]	Message waiting to be read on a *stream* is not a data message.
[EDEADLK]	The read was going to go to sleep and cause a deadlock situation to occur.
[EFAULT]	*Buf* points outside the allocated address space.
[EINTR]	A signal was caught during the *read* system call.
[EINVAL]	Attempted to read from a *stream* linked to a multiplexor.
[ENOLCK]	The system record lock table was full, so the read could not go to sleep until the blocking record lock was removed.
[ENOLINK]	*Fildes* is on a remote machine and the link to that machine is no longer active.

A *read* from a STREAMS file will also fail if an error message is received at the *stream head*. In this case, *errno* is set to the value returned in the error message. If a hangup occurs on the *stream* being read, *read* will continue to operate normally until the *stream head* read queue is empty. Thereafter, it will return 0.

SEE ALSO

creat(2), dup(2), fcntl(2), ioctl(2),intro(2), open(2), pipe(2), getmsg(2).
streamio(7), termio(7) in the *System Administrator's Reference Manual.*

DIAGNOSTICS

Upon successful completion a non-negative integer is returned indicating the number of bytes actually read. Otherwise, a −1 is returned and *errno* is set to indicate the error.

NAME

readlink – read value of a symbolic link

SYNOPSIS

cc = readlink(path, buf, bufsiz)
int cc;
char *path, *buf;
int bufsiz;

DESCRIPTION

readlink places the contents of the symbolic link *name* in the buffer *buf,* which has size *bufsiz*. The contents of the link are not null terminated when returned.

RETURN VALUE

The call returns the count of characters placed in the buffer if it succeeds, or a −1 if an error occurs, placing the error code in the global variable *errno*.

ERRORS

readlink will fail and the file mode will be unchanged if:

[ENOTDIR]	A component of the path prefix is not a directory.
[EINVAL]	The pathname contains a character with the high-order bit set.
[ENAMETOOLONG]	A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.
[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 pathname.
[EINVAL]	The named file is not a symbolic link.
[EIO]	An I/O error occurred while reading from the file system.
[EFAULT]	*Buf* extends outside the process's allocated address space.

SEE ALSO

stat(2), symlink(2)

NAME

recv, recvfrom, recvmsg – receive a message from a socket

SYNOPSIS

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

cc = recv(s, buf, len, flags)
int cc, s;
char *buf;
int len, flags;

cc = recvfrom(s, buf, len, flags, from, fromlen)
int cc, s;
char *buf;
int len, flags;
struct sockaddr *from;
int *fromlen;

cc = recvmsg(s, msg, flags)
int cc, s;
struct msghdr msg[];
int flags;
```

DESCRIPTION

recv, *recvfrom*, and *recvmsg* are used to receive messages from a socket.

The *recv* call is normally used only on a *connected* socket (see *connect*(2)), while *recvfrom* and *recvmsg* may be used to receive data on a socket whether it is in a connected state or not.

If *from* is non-zero, the source address of the message is filled in. *fromlen* is a value-result parameter, initialized to the size of the buffer associated with *from*, and modified on return to indicate the actual size of the address stored there. The length of the message is returned in *cc*. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from (see *socket*(2)).

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is nonblocking (see *ioctl*(2)) in which case a *cc* of −1 is returned with the external variable errno set to EWOULDBLOCK.

The *select*(2) call may be used to determine when more data arrives.

The *flags* argument to a recv call is formed by *or*'ing one or more of the values,

```
#define MSG_OOB         0x1    /* process out-of-band data */
#define MSG_PEEK        0x2    /* peek at incoming message */
```

The *recvmsg* call uses a *msghdr* structure to minimize the number of directly supplied parameters. This structure has the following form, as defined in *<bsd/sys/socket.h>*:

```
struct msghdr {
        caddr_t msg_name;           /* optional address */
        int     msg_namelen;        /* size of address */
        struct  iovec *msg_iov;     /* scatter/gather array */
        int     msg_iovlen;         /* # elements in msg_iov */
        caddr_t msg_accrights;      /* access rights sent/received */
        int     msg_accrightslen;
};
```

Here *msg_name* and *msg_namelen* specify the destination address if the socket is unconnected; *msg_name* may be given as a null pointer if no names are desired or required. The *msg_iov* and *msg_iovlen* describe the scatter gather locations, as described in *read*(2). A buffer to receive any access rights sent along with the message is specified in *msg_accrights*, which has length *msg_accrightslen*. Access rights are currently limited to file descriptors, which each occupy the size of an **int**.

RETURN VALUE

These calls return the number of bytes received, or −1 if an error occurred.

ERRORS

The calls fail if:

[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]	The data was specified to be received into a non-existent or protected part of the process address space.

SEE ALSO

fcntl(2), read(2), send(2), select(2), getsockopt(2), socket(2)

NAME

rename – change the name of a file

SYNOPSIS

rename(from, to)
char *from, *to;

DESCRIPTION

rename causes the link named *from* to be renamed as *to*. If *to* exists, then it is first removed. Both *from* and *to* must be of the same type (that is, both directories or both non-directories), and must reside on the same file system.

rename guarantees that an instance of *to* will always exist, even if the system should crash in the middle of the operation.

If the final component of *from* is a symbolic link, the symbolic link is renamed, not the file or directory to which it points.

CAVEAT

The system can deadlock if a loop in the file system graph is present. This loop takes the form of an entry in directory "a", say "a/foo", being a hard link to directory "b", and an entry in directory "b", say "b/bar", being a hard link to directory "a". When such a loop exists and two separate processes attempt to perform "rename a/foo b/bar" and "rename b/bar a/foo", respectively, the system may deadlock attempting to lock both directories for modification. Hard links to directories should be replaced by symbolic links by the system administrator.

RETURN VALUE

A 0 value is returned if the operation succeeds, otherwise *rename* returns –1 and the global variable *errno* indicates the reason for the failure.

ERRORS

rename will fail and neither of the argument files will be affected if any of the following are true:

[ENAMETOOLONG]	A component of either pathname exceeded 255 characters, or the entire length of either path name exceeded 1023 characters.
[ENOENT]	A component of the *from* path does not exist, or a path prefix of *to* does not exist.
[EACCES]	A component of either path prefix denies search permission.
[EACCES]	The requested link requires writing in a directory with a mode that denies write permission.
[EPERM]	The directory containing *from* is marked sticky, and neither the containing directory nor *from* are owned by the effective user ID.
[EPERM]	The *to* file exists, the directory containing *to* is marked sticky, and neither the containing directory nor *to* are owned by the effective user ID.
[ELOOP]	Too many symbolic links were encountered in translating either pathname.
[ENOTDIR]	A component of either path prefix is not a directory.
[ENOTDIR]	*from* is a directory, but *to* is not a directory.
[EISDIR]	*to* is a directory, but *from* is not a directory.
[EXDEV]	The link named by *to* and the file named by *from* are on different logical devices (file systems). Note that this error code will not be returned if the implementation permits cross-device links.

[ENOSPC]	The directory in which the entry for the new name is being placed cannot be extended because there is no space left on the file system containing the directory.
[EDQUOT]	The directory in which the entry for the new name is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.
[EIO]	An I/O error occurred while making or updating a directory entry.
[EROFS]	The requested link requires writing in a directory on a read-only file system.
[EFAULT]	*path* points outside the process's allocated address space.
[EINVAL]	*from* is a parent directory of *to*, or an attempt is made to rename "." or "..".
[ENOTEMPTY]	*to* is a directory and is not empty.

SEE ALSO

open(2)

NAME

rmdir – remove a directory

SYNOPSIS

int rmdir (path)
char *path;

DESCRIPTION

rmdir removes the directory named by the path name pointed to by *path*. The directory must not have any entries other than "." and "..".

The named directory is removed unless one or more of the following are true:

[EINVAL]	The current directory may not be removed.
[EINVAL]	The "." entry of a directory may not be removed.
[EEXIST]	The directory contains entries other than those for "." and "..".
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named directory does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[EACCES]	Write permission is denied on the directory containing the directory to be removed.
[EBUSY]	The directory to be removed is the mount point for a mounted file system.
[EROFS]	The directory entry to be removed is part of a read-only file system.
[EFAULT]	*path* points outside the process's allocated address space.
[EIO]	An I/O error occurred while accessing the file system.
[ENOLINK]	*path* points to a remote machine, and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

DIAGNOSTICS

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

SEE ALSO

mkdir(2).
rmdir(1), rm(1), and mkdir(1) in the *User's Reference Manual*.

NAME

select – synchronous I/O multiplexing

SYNOPSIS

```
#include <bsd/sys/types.h>
#include <bsd/sys/time.h>

nfound = select(nfds, readfds, writefds, exceptfds, timeout)
int nfound, nfds;
fd_set *readfds, *writefds, *exceptfds;
struct timeval *timeout;

FD_SET(fd, &fdset)
FD_CLR(fd, &fdset)
FD_ISSET(fd, &fdset)
FD_ZERO(&fdset)
int fd;
fd_set fdset;
```

DESCRIPTION

select examines the I/O descriptor sets whose addresses are passed in *readfds*, *writefds*, and *exceptfds* to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. The first *nfds* descriptors are checked in each set; i.e. the descriptors from 0 through *nfds*-1 in the descriptor sets are examined. On return, *select* replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. The total number of ready descriptors in all the sets is returned in *nfound*.

The descriptor sets are stored as bit fields in arrays of integers. The following macros are provided for manipulating such descriptor sets: " *FD_ZERO(&fdset)*" initializes a descriptor set *fdset* to the null set. *FD_SET(fd, &fdset)*" includes a particular descriptor *fd* in *fdset*. *FD_CLR(fd, &fdset)* removes *fd* from *fdset*. *FD_ISSET(fd, &fdset)* is nonzero if *fd* is a member of *fdset*, zero otherwise. The behavior of these macros is undefined if a descriptor value is less than zero or greater than or equal to *FD_SETSIZE*, which is normally at least equal to the maximum number of descriptors supported by the system.

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 blocks indefinitely. To affect a poll, the *timeout* argument should be non-zero, pointing to a zero-valued timeval structure.

Any of *readfds*, *writefds*, and *exceptfds* may be given as zero pointers if no descriptors are of interest.

RETURN VALUE

select returns the number of ready descriptors that are contained in the descriptor sets, or –1 if an error occurred. If the time limit expires then *select* returns 0.

If *select* returns with an error, including one due to an interrupted call, the descriptor sets will be unmodified.

ERRORS

An error return from *select* indicates:

[EBADF]	One of the descriptor sets specified an invalid descriptor.
[EINTR]	A signal was delivered before the time limit expired and before any of the selected events occurred.
[EINVAL]	The specified time limit is invalid. One of its components is negative or too large.

SEE ALSO

accept(2), connect(2), read(2), write(2), recv(2), send(2)

NAME

semctl – semaphore control operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl (semid, semnum, cmd, arg)
int semid, cmd;
int semnum;
union semun {
    int val;
    struct semid_ds *buf;
    ushort *array;
} 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 *intro*(2)]. {READ}

SETVAL Set the value of semval to *arg.val*. {ALTER} When this cmd is successfully executed, the semadj value corresponding to the specified semaphore in all processes is cleared.

GETPID Return the value of sempid. {READ}

GETNCNT Return the value of semncnt. {READ}

GETZCNT Return the value of semzcnt. {READ}

The following *cmd*s return and set, respectively, every semval in the set of semaphores.

GETALL Place semvals into array pointed to by *arg.array*. {READ}

SETALL Set semvals according to the array pointed to by *arg.array*. {ALTER} When this cmd is successfully executed the semadj values corresponding to each specified semaphore in all processes are cleared.

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.buf*. The contents of this structure are defined in *intro*(2). {READ}

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.buf*:
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 **sem_perm.cuid** or **sem_perm.uid** 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 **sem_perm.cuid** or **sem_perm.uid** in the data structure associated with *semid*.

ERRORS

semctl fails if one or more of the following are true:

[EINVAL]	*semid* is not a valid semaphore identifier.
[EINVAL]	*semnum* is less than zero or greater than **sem_nsems**.
[EINVAL]	*cmd* is not a valid command.
[EACCES]	Operation permission is denied to the calling process [see *intro*(2)].
[ERANGE]	*cmd* is **SETVAL** or **SETALL** and the value to which semval is to be set is greater than the system imposed maximum.
[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, or to the value of **sem_perm.cuid** or **sem_perm.uid** in the data structure associated with *semid*.
[EFAULT]	*Arg.buf* points to an illegal address.

SEE ALSO

intro(2), semget(2), semop(2).

DIAGNOSTICS

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

GETVAL	The value of semval.
GETPID	The value of sempid.
GETNCNT	The value of semncnt.
GETZCNT	The value of semzcnt.
All others	A value of 0.

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

NAME

semget – get set of semaphores

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget (key, nsems, semflg)
key_t key;
int nsems, semflg;
```

DESCRIPTION

semget returns the semaphore identifier associated with *key*.

A semaphore identifier and associated data structure and set containing *nsems* semaphores [see *intro*(2)] are created for *key* if one of the following is true:

key is equal to **IPC_PRIVATE**.

key does not already have a semaphore identifier associated with it, and (*semflg* & **IPC_CREAT**) is "true".

Upon creation, the data structure associated with the new semaphore identifier is initialized as follows:

Sem_perm.cuid, **sem_perm.uid**, **sem_perm.cgid**, and **sem_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 **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.

ERRORS

semget fails if one or more of the following are 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 [see *intro*(2)] 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.

[ENOSPC]	A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphores system wide would be exceeded.
[EEXIST]	A semaphore identifier exists for *key* but ((*semflg* & **IPC_CREAT**) and (*semflg* & **IPC_EXCL**)) is "true".

SEE ALSO

intro(2), semctl(2), semop(2).

DIAGNOSTICS

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.

NAME

semop – semaphore operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop (semid, sops, nsops)
int semid;
struct sembuf **sops;
unsigned nsops;
```

DESCRIPTION

semop is used to automatically 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:

```
short   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*.

sem_op specifies one of three semaphore operations as follows:

If *sem_op* is a negative integer, one of the following will occur: {ALTER}

If semval [see *intro*(2)] 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 *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* will return immediately.

If semval is less than the absolute value of *sem_op* and (*sem_flg* & **IPC_NOWAIT**) is "false", *semop* will increment 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*(2).

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. {ALTER}

If *sem_op* is zero, one of the following will occur: {READ}

If semval is zero, *semop* will return immediately.

If semval is not equal to zero and (*sem_flg* & **IPC_NOWAIT**) is "true", *semop* will return immediately.

If semval is not equal to zero and (*sem_flg* & **IPC_NOWAIT**) is "false", *semop* will increment the semzcnt associated with the specified semaphore and suspend 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*(2).

ERRORS

semop will fail if one or more of the following are 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 *intro*(2)]
[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.

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.

SEE ALSO

exec(2), exit(2), fork(2), intro(2), semctl(2), semget(2).

DIAGNOSTICS

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 value of zero is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

send, sendto, sendmsg – send a message from a socket

SYNOPSIS

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

cc = send(s, msg, len, flags)
int cc, s;
char *msg;
int len, flags;

cc = sendto(s, msg, len, flags, to, tolen)
int cc, s;
char *msg;
int len, flags;
struct sockaddr *to;
int tolen;

cc = sendmsg(s, msg, flags)
int cc, s;
struct msghdr msg[];
int flags;
```

DESCRIPTION

send , *sendto*, and *sendmsg* are used to transmit a message to another socket. *send* may be used only when the socket is in a *connected* state, while *sendto* and *sendmsg* may be used at any time.

The address of the target is given by *to* with *tolen* specifying its size. The length of the message is given by *len*. If the message is too long to pass atomically through the underlying protocol, then the error EMSGSIZE is returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a *send*. Return values of −1 indicate some locally detected errors.

If no messages space is available at the socket to hold the message to be transmitted, then *send* normally blocks, unless the socket has been placed in non-blocking I/O mode. The *select*(2) call may be used to determine when it is possible to send more data.

The *flags* parameter may include one or more of the following:

```
#define MSG_OOB          0x1    /* process out-of-band data */
#define MSG_DONTROUTE    0x4    /* bypass routing,
                                   use direct interface */
```

The flag MSG_OOB is used to send "out-of-band" data on sockets that support this notion (e.g. SOCK_STREAM); the underlying protocol must also support "out-of-band" data. MSG_DONTROUTE is usually used only by diagnostic or routing programs.

See *recv*(2) for a description of the *msghdr* structure.

RETURN VALUE

The call returns the number of characters sent, or −1 if an error occurred.

ERRORS

[EBADF]	An invalid descriptor was specified.
[ENOTSOCK]	The argument *s* is not a socket.
[EFAULT]	An invalid user space address was specified for a parameter.

[EMSGSIZE]	The socket requires that message be sent atomically, and the size of the message to be sent made this impossible.
[EWOULDBLOCK]	The socket is marked non-blocking and the requested operation would block.
[ENOBUFS]	The system was unable to allocate an internal buffer. The operation may succeed when buffers become available.
[ENOBUFS]	The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but may be caused by transient congestion.

SEE ALSO

fcntl(2), recv(2), select(2), getsockopt(2), socket(2), write(2)

ORIGIN

4.3 BSD

NAME

setpgid – set process group ID for job control

SYNOPSIS

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

int setpgid (pid, pgid)
pid_t pid,pgid;
```

DESCRIPTION

The *setpgid*() function is used to either join an existing proces group or create a new process group within the session of the calling process. The process group ID of a session leader shall not change. Upon successfull completion, the process group ID of the process with a process ID that matches *pid* shall be set to *pgid*. As a special, if *pid* is zero, the process ID of the calling process shall be used. Also, if *pgid* is zero, the process ID of the indicated process shall be used.

RETURNS

Upon successful completion, the *setpgid*() function returns a value of zero. Otherwise, a value of −1 is returned an d*errno* is set to indicate the error.

ERRORS

If any of the following conditions occur, the *setpgid*() function shall return −1 and set *errno* to the corresponding value:

[EINVAL] The value of the *pgid* argument is less than or equal to zero or is not a value supported by the implementation.

[EPERM] The process indicated by the *pid* argument is a session leader.
The value of the *pid* argument 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.
The value of the *pgid* argument does not match the process ID of the process indicated by the *pid* argument and there is no process with a process group ID that matches the value of the *pgid* argument in the same session as the calling process.

[ESRCH] The value of the *pid* argument does not match the process ID of the calling process or of a child process of the calling process.

SEE ALSO

setpgrp()
ioctl

NAME

setpgrp – set process group ID

SYNOPSIS

int setpgrp ()

DESCRIPTION

setpgrp sets the process group ID of the calling process to the process ID of the calling process and returns the new process group ID.

SEE ALSO

exec(2), fork(2), getpid(2), intro(2), kill(2), signal(2).

DIAGNOSTICS

setpgrp returns the value of the new process group ID.

NAME

setuid, setgid – set user and group IDs

SYNOPSIS

int setuid (uid)
int uid;

int setgid (gid)
int gid;

DESCRIPTION

setuid (*setgid*) is used to set the real user (group) ID and effective user (group) ID of the calling process.

If the effective user ID of the calling process is super-user, the real user (group) ID and effective user (group) ID are set to *uid* (*gid*).

If the effective user ID of the calling process is not super-user, but its real user (group) ID is equal to *uid* (*gid*), the effective user (group) ID is set to *uid* (*gid*).

If the effective user ID of the calling process is not super-user, but the saved set-user (group) ID from *exec*(2) is equal to *uid* (*gid*), the effective user (group) ID is set to *uid* (*gid*).

setuid (*setgid*) will fail if the real user (group) ID of the calling process is not equal to *uid* (*gid*) and its effective user ID is not super-user. [EPERM]

The *uid* is out of range. [EINVAL]

SEE ALSO

getuid(2), intro(2).

DIAGNOSTICS

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

NAME

shmctl – shared memory control operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl (shmid, cmd, buf)
int shmid, 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 *intro*(2). {READ}

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 that of super user, or to the value of **shm_perm.cuid** or **shm_perm.uid** 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. This *cmd* can only be executed by a process that has an effective user ID equal to that of super user, or to the value of **shm_perm.cuid** or **shm_perm.uid** 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 has an effective user ID equal to super user.

SHM_UNLOCK
Unlock the shared memory segment specified by *shmid*. This *cmd* can only be executed by a process that has an effective user ID equal to super user.

ERRORS

shmctl will fail if one or more of the following are true:

[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 *intro*(2)].

[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, or to the value of **shm_perm.cuid** or **shm_perm.uid** 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 super user.

[EFAULT] *Buf* points to an illegal address.

[ENOMEM] *cmd* is equal to **SHM_LOCK** and there is not enough memory.

SEE ALSO

shmget(2), shmop(2).

DIAGNOSTICS

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

NOTES

The user must explicitly remove shared memory segments after the last reference to them has been removed.

NAME

shmget – get shared memory segment identifier

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget (key, size, shmflg)
key_t key;
int size, shmflg;
```

DESCRIPTION

shmget returns the shared memory identifier associated with *key*.

A shared memory identifier and associated data structure and shared memory segment of at least *size* bytes [see *intro*(2)] are created for *key* if one of the following are true:

key is equal to **IPC_PRIVATE**.

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.

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.

ERRORS

shmget will fail if one or more of the following are true:

[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 *intro*(2)] 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 memory is not sufficient to fill the request.
[EEXIST]	A shared memory identifier exists for *key* but ((*shmflg* & **IPC_CREAT**) and (*shmflg*& **IPC_EXCL**)) is "true".

SEE ALSO

intro(2), shmctl(2), shmop(2).

DIAGNOSTICS

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.

NOTES

The user must explicitly remove shared memory segments after the last reference to them has been removed.

NAME

shmop, shmat, shmdt – shared memory operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

char *shmat (shmid, shmaddr, shmflg)
int shmid;
char *shmaddr;
int shmflg;

int shmdt (shmaddr)
char *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. 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.

If *shmaddr* is not equal to zero and (*shmflg* & **SHM_RND**) is "true", the segment is attached at the address given by (*shmaddr* - (*shmaddr* modulus **SHMLBA**)).

If *shmaddr* is not equal to zero and (*shmflg* & **SHM_RND**) is "false", the segment is attached at the address given by *shmaddr*.

shmdt detaches from the calling process's data segment the shared memory segment located at the address specified by *shmaddr*.

The segment is attached for reading if (*shmflg* & **SHM_RDONLY**) is "true" {READ}, otherwise it is attached for reading and writing {READ/WRITE}.

ERRORS

shmat will fail and not attach the shared memory segment if one or more of the following are true:

[EINVAL]	*shmid* is not a valid shared memory identifier.
[EACCES]	Operation permission is denied to the calling process [see *intro*(2)].
[ENOMEM]	The available data space is not large enough to accommodate the shared memory segment.
[EINVAL]	*shmaddr* is not equal to zero, and the value of (*shmaddr* - (*shmaddr* modulus **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.
[EMFILE]	The number of shared memory segments attached to the calling process would exceed the system-imposed limit.
[EINVAL]	*shmdt* will fail and not detach the shared memory segment if *shmaddr* is not the data segment start address of a shared memory segment.

SEE ALSO

exec(2), exit(2), fork(2), intro(2), shmctl(2), shmget(2).

DIAGNOSTICS

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.

NOTES

The user must explicitly remove shared memory segments after the last reference to them has been removed.

NAME

signal – specify what to do upon receipt of a signal

SYNOPSIS

```
#include <signal.h>

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

DESCRIPTION

signal allows the calling process to choose one of three ways in which it is possible to handle the receipt of a specific signal. *sig* specifies the signal and *func* specifies the choice.

sig can be assigned any one of the following except **SIGKILL**:

SIGHUP	01	hangup
SIGINT	02	interrupt
SIGQUIT	03[1]	quit
SIGILL	04[1]	illegal instruction (not reset when caught)
SIGTRAP	05[1]	trace trap (not reset when caught)
SIGIOT	06[1]	IOT instruction
SIGEMT	07[1]	EMT instruction
SIGFPE	08[1]	floating point exception
SIGKILL	09	kill (cannot be caught or ignored)
SIGBUS	10[1]	bus error
SIGSEGV	11[1]	segmentation violation
SIGSYS	12[1]	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGUSR1	16	user-defined signal 1
SIGUSR2	17	user-defined signal 2
SIGCLD	18[2]	death of a child
SIGPWR	19[2]	power fail
SIGPOLL	22[3]	selectable event pending

func is assigned one of three values: **SIG_DFL**, **SIG_IGN**, or a *function address*. **SIG_DFL**, and **SIG_IGN**, are defined in the include file *signal.h*. Each is a macro that expands to a constant expression of type pointer to function returning *void*, and has a unique value that matches no declarable function.

The actions prescribed by the values of *func* are as follows:

SIG_DFL – terminate process upon receipt of a signal

Upon receipt of the signal *sig*, the receiving process is to be terminated with all of the consequences outlined in *exit(2)*. See NOTE [1] below.

SIG_IGN – ignore signal

The signal *sig* is to be ignored.

Note: the signal **SIGKILL** cannot be ignored.

function address – catch signal

Upon receipt of the signal *sig*, the receiving process is to execute the signal-catching function pointed to by *func*. The signal number *sig* will be passed as the only argument to the signal-catching function. Additional arguments are passed to the signal-catching function for hardware-generated signals. Before entering the signal-catching function, the value of *func* for the caught signal will be set to

SIG_DFL unless the signal is **SIGILL**, **SIGTRAP**, or **SIGPWR**.

Upon return from the signal-catching function, the receiving process will resume execution at the point it was interrupted.

When a signal that is to be caught occurs during a *read(2)*, a *write(2)*, an *open(2)* ,*or*an *ioctl(2)* system call on a slow device (like a terminal; but not a file), during a *pause(2)* system call, or during a *wait(2)* system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal catching function will be executed and then the interrupted system call may return a −1 to the calling process with *errno* set to EINTR.

signal will not catch an invalid function argument, *func,* and results are undefined when an attempt is made to execute the function at the bad address.

Note: The signal **SIGKILL** cannot be caught.

A call to **signal** cancels a pending signal *sig* except for a pending **SIGKILL** signal.

signal will fail if *sig* is an illegal signal number, including **SIGKILL**.

NOTES

[1] If **sIG_DFL** is assigned for these signals, in addition to the process being terminated, a "core image" will be constructed in the current working directory of the process, if the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

An ordinary file named **core** exists and is writable or can be created. If the file must be created, it will have the following properties:

a mode of 0666 modified by the file creation mask [see *umask(2)*]

a file owner ID that is the same as the effective user ID of the receiving process.

a file group ID that is the same as the effective group ID of the receiving process

[2] For the signals **SIGCLD** and **SIGPWR**, *func* is assigned one of three values: **SIG_DFL**, **SIG_IGN**, or a *function address*. The actions prescribed by these values are:

sIG_DFL - ignore signal
The signal is to be ignored.

sIG_IGN - ignore signal
The signal is to be ignored. Also, if *sig* is **SIGCLD**, the calling process's child processes will not create zombie processes when they terminate [see *exit(2)*].

function address - catch signal
If the signal is **SIGPWR**, the action to be taken is the same as that described above for *func* equal to *function address*. The same is true if the signal is **sIGCLD** with one exception: while the process is executing the signal-catching function, any received **sIGCLD** signals will be ignored. (This is the default action.)

In addition, **sIGCLD** affects the *wait*, and *exit* system calls as follows:

wait If the *func* value of **sIGCLD** is set to **sIG_IGN** and a *wait* is executed, the *wait* will block until all of the calling process's child processes terminate; it will then return a value of −1 with *errno* set to ECHILD.

exit If in the exiting process's parent process the *func* value of **sIGCLD** is set to **SIG_IGN**, the exiting process will not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the proceeding processes. A process that may be piped into in this manner (and thus become the parent of other processes) should take care not to set **sIGCLD** to be caught.

[3] **SIGPOLL** is issued when a file descriptor corresponding to a STREAMS [see *intro(2)*] file has a "selectable" event pending. A process must specifically request that this signal be sent using the I_SETSIG *ioctl* call. Otherwise, the process will never receive **SIGPOLL**.

SEE ALSO

intro(2), kill(2), pause(2), ptrace(2), wait(2), setjmp(3C), sigset(2).
kill(1) in the *User's Reference Manual.*
R2010 Floating Point Coprocessor Architecture
R2360 Floating Point Board Product Description

DIAGNOSTICS

Upon successful completion, **signal** returns the previous value of *func* 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 the include file *signal.h*.

NOTES (MIPS)

The handler routine can be declared:

```
handler(sig, code, scp)
int sig, code;
struct sigcontext *scp;
```

Here *sig* is the signal number. MIPS hardware exceptions are mapped to specific signals as defined by the table below. *code* is a parameter that is either a constant as given below or zero. *scp* is a pointer to the *sigcontext* structure (defined in <*signal.h*>), that is the context at the time of the signal and is used to restore the context if the signal handler returns.

The following defines the mapping of MIPS hardware exceptions to signals and codes. All of these symbols are defined in either <*signal.h*> or <*sys/sbd.h*>:

Hardware exception	Signal	Code
Integer overflow	SIGFPE	EXC_OV
Segmentation violation	SIGSEGV	SEXC_SEGV
Illegal Instruction	SIGILL	EXC_II
Coprocessor Unusable	SIGILL	SEXC_CPU
Data Bus Error	SIGBUS	EXC_DBE
Instruction Bus Error	SIGBUS	EXC_IBE
Read Address Error	SIGBUS	EXC_RADE
Write Address Error	SIGBUS	EXC_WADE
User Breakpoint (used by debuggers)	SIGTRAP	BRK_USERBP
Kernel Breakpoint (used by prom)	SIGTRAP	BRK_KERNELBP
aken Branch Delay Emulation	SIGTRAP	BRK_BD_TAKEN
Not Taken Branch Delay Emulation	SIGTRAP	BRK_BD_NOTTAKEN
User Single Step (used by debuggers)	SIGTRAP	BRK_SSTEPBP
Overflow Check	SIGTRAP	BRK_OVERFLOW
Divide by Zero Check	SIGTRAP	BRK_DIVZERO
Range Error Check	SIGTRAP	BRK_RANGE

When a signal handler is reached, the program counter in the signal context structure (*sc_pc*) points at the instruction that caused the exception as modified by the *branch delay* bit in the *cause* register. The *cause* register at the time of the exception is also saved in the sigcontext structure (*sc_cause*). If the instruction that caused the exception is at a valid user address it can be retrieved with the following code sequence:

```
if(scp->sc_cause & CAUSE_BD){
    branch_instruction = *(unsigned long *)(scp->sc_pc);
    exception_instruction = *(unsigned long *)(scp->sc_pc + 4);
}
else
    exception_instruction = *(unsigned long *)(scp->sc_pc);
```

Where CAUSE_BD is defined in *<sys/sbd.h>*.

The signal handler may fix the cause of the exception and re-execute the instruction, emulate the instruction and then step over it or perform some non-local goto such as a *longjump()* or an *exit()*.

If corrective action is performed in the signal handler and the instruction that caused the exception would then execute without a further exception, the signal handler simply returns and re-executes the instruction (even when the *branch delay* bit is set).

If execution is to continue after stepping over the instruction that caused the exception the program counter must be advanced. If the *branch delay* bit is set the program counter is set to the target of the branch else it is incremented by 4. This can be done with the following code sequence:

```
if(scp->sc_cause & CAUSE_BD)
    emulate_branch(scp, branch_instruction);
else
    scp->sc_pc += 4;
```

Emulate_branch() modifies the program counter value in the sigcontext structure to the target of the branch instruction. See *emulate_branch(3)* for more details.

For SIGFPE's generated by floating-point instructions (*code* == 0) the *floating-point control and status* register at the time of the exception is also saved in the sigcontext structure (*sc_fpc_csr*). This register has the information on which exceptions have occurred. When a signal handler is entered the register contains the value at the time of the exception but with the *exceptions bits* cleared. On a return from the signal handler the exception bits in the floating-point control and status register are also cleared so that another SIGFPE will not occur (all other bits are restored from *sc_fpc_csr*).

If the floating-point unit is a R2360 (a floating-point board) and a SIGFPE is generated by the floating-point unit (*code* == 0) and program counter does not point at the instruction that caused the exception. In this case the instruction that caused the exception is in the *floating-point instruction exception* register. The floating-point instruction exception register at the time of the exception is also saved in the sigcontext structure (*sc_fpc_eir*). In this case the instruction that caused the exception can be retrieved with the following code sequence:

```
union fpc_irr fpc_irr;

fpc_irr.fi_word = get_fpc_irr();
if(sig == SIGFPE && code == 0 &&
   fpc_irr.fi_struct.implementation == IMPLEMENTATION_R2360)
    exception_instruction = scp->sc_fpc_eir;
```

The union *fpc_irr,* and the constant IMPLEMENTATION_R2360 are defined in *<sys/fpu.h>*. For the description of the routine *get_fpc_irr()* see *fpc(3)*. All other floating-point implementations are handled in the normal manner with the instruction that caused the exception at the program counter as modified by the *branch delay* bit.

For SIGSEGV and SIGBUS errors the faulting virtual address is saved in *sc_badvaddr* in the signal context structure.

The SIGTRAP's caused by **break** instructions noted in the above table and all other yet to be defined **break** instructions fill the *code* parameter with the first argument to the **break** instruction (bits 25-16 of the instruction).

NAME

sigset, sighold, sigrelse, sigignore, sigpause – signal management

SYNOPSIS

```
#include <signal.h>

void (*sigset (sig, func))()
int sig;
void (*func)();

int sighold (sig)
int sig;

int sigrelse (sig)
int sig;

int sigignore (sig)
int sig;

int sigpause (sig)
int sig;
```

DESCRIPTION

These functions provide signal management for application processes. *sigset* specifies the system signal action to be taken upon receipt of signal *sig*. This action is either calling a process signal-catching handler *func* or performing a system-defined action.

sig can be assigned any one of the following values except SIGKILL. Machine or implementation dependent signals are not included (see *NOTES* below). Each value of *sig* is a macro, defined in *<signal.h>*, that expands to an integer constant expression.

SIGHUP	hangup
SIGINT	interrupt
SIGQUIT*	quit
SIGILL*	illegal instruction (not held when caught)
SIGTRAP*	trace trap (not held when caught)
SIGABRT*	abort
SIGFPE*	floating point exception
SIGKILL	kill (can not be caught or ignored)
SIGSYS*	bad argument to system call
SIGPIPE	write on a pipe with no one to read it
SIGALRM	alarm clock
SIGTERM	software termination signal
SIGUSR1	user-defined signal 1
SIGUSR2	user-defined signal 2
SIGCLD	death of a child (see *WARNING* below)
SIGPWR	power fail (see *WARNING* below)
SIGPOLL	selectable event pending (see *NOTES* below)

See below under SIG_DFL regarding asterisks (*) in the above list.

The following values for the system-defined actions of *func* are also defined in *<signal.h>*. Each is a macro that expands to a constant expression of type pointer to function returning *void* and has a unique value that matches no declarable function.

SIG_DFL – default system action

Upon receipt of the signal *sig*, the receiving process is to be terminated with all of the consequences outlined in *exit*(2). In addition a "core image" will be made in the current working directory of the receiving process if *sig* is one for which an asterisk appears in the above list *and* the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

An ordinary file named *core* exists and is writable or can be created. If the file must be created, it will have the following properties:

a mode of 0666 modified by the file creation mask [see *umask*(2)]

a file owner ID that is the same as the effective user ID of the receiving process.

a file group ID that is the same as the effective group ID of the receiving process

SIG_IGN – ignore signal

Any pending signal *sig* is discarded and the system signal action is set to ignore future occurrences of this signal type.

SIG_HOLD – hold signal

The signal *sig* is to be held upon receipt. Any pending signal of this type remains held. Only one signal of each type is held.

Otherwise, *func* must be a pointer to a function, the signal-catching handler, that is to be called when signal *sig* occurs. In this case, *sigset* specifies that the process will call 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.

When a signal occurs, the signal number *sig* will be passed as the only argument to the signal-catching handler. Before calling the signal-catching handler, the system signal action will be set to SIG_HOLD . During normal return from the signal-catching handler, the system signal action is restored to *func* and any held signal of this type released. If a non-local goto (*longjmp*) is taken, then *sigrelse* must be called to restore the system signal action and release any held signal of this type.

In general, upon return from the signal-catching handler, the receiving process will resume execution at the point it was interrupted. However, when a signal is caught during a *read*(2), a *write*(2), an *open*(2), or an *ioctl*(2) system call during a *sigpause* system call, or during a *wait*(2) system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal-catching handler will be executed and then the interrupted system call may return a −1 to the calling process with *errno* set to EINTR.

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*. *sigrelse* restores the system signal action to that specified previously by *sigset*.

sigignore sets the action for signal *sig* to SIG_IGN (see above).

sigpause suspends the calling process until it receives a signal, the same as *pause*(2). However, if the signal *sig* had been received and held, it is released and the system signal action taken. This system call is useful for testing variables that are changed on the occurrence of a signal. The correct usage is to use *sighold* to block the signal first, then test the variables. If they have not changed, then call *sigpause* to wait for the signal.

ERRORS

sigset will fail if one or more of the following are true:

[EINVAL] *sig* is an illegal signal number (including **SIGKILL**) or the default handling of *sig* cannot be changed.

[EINTR] A signal was caught during the system call *sigpause*.

DIAGNOSTICS

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

SEE ALSO

kill(2), pause(2), signal(2), wait(2), setjmp(3C).
R2010 Floating Point Coprocessor Architecture
R2360 Floating Point Board Product Description

WARNING

Two signals that behave differently than the signals described above exist in this release of the system:

SIGCLD	death of a child (reset when caught)
SIGPWR	power fail (not reset when caught)

For these signals, *func* 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 - ignore signal
The signal is to be ignored.

SIG_IGN - ignore signal
The signal is to be ignored. Also, if *sig* is **SIGCLD**, the calling process's child processes will not create zombie processes when they terminate [see *exit*(2)].

function address - catch signal
If the signal is **SIGPWR**, the action to be taken is the same as that described above for *func* equal to *function address*. The same is true if the signal is **SIGCLD** with one exception: while the process is executing the signal-catching function, any received **SIGCLD** signals will be ignored. (This is the default action.)

The **SIGCLD** affects two other system calls [*wait*(2), and *exit*(2)] in the following ways:

wait If the *func* value of **SIGCLD** is set to SIG_IGN and a *wait* is executed, the *wait* will block until all of the calling process's child processes terminate; it will then return a value of −1 with *errno* set to ECHILD.

exit If in the exiting process's parent process the *func* value of **SIGCLD** is set to SIG_IGN , the exiting process will not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the proceeding processes. A process that may be piped into in this manner (and thus become the parent of other processes) should take care not to set **SIGCLD** to be caught.

NOTES

SIGPOLL is issued when a file descriptor corresponding to a STREAMS [see *intro*(2)] file has a "selectable" event pending. A process must specifically request that this signal be sent using the I_SETSIG *ioctl*(2) call [see *streamio*(7)]. Otherwise, the process will never receive **SIGPOLL**.

For portability, applications should use only the symbolic names of signals rather than their values and use only the set of signals defined here. The action for the signal **SIGKILL** can not be changed from the default system action.

Specific implementations may have other implementation-defined signals. Also, additional implementation-defined arguments may be passed to the signal-catching handler for hardware-generated signals. For certain hardware-generated signals, it may not be possible to resume

execution at the point of interruption.

The signal type **SIGSEGV** is reserved for the condition that occurs on an invalid access to a data object. If an implementation can detect this condition, this signal type should be used.

The other signal management functions, *signal*(2) and *pause*(2), should not be used in conjunction with these routines for a particular signal type.

NOTES (MIPS)

The handler routine can be declared:

```
handler(sig, code, scp)
int sig, code;
struct sigcontext *scp;
```

Here *sig* is the signal number. MIPS hardware exceptions are mapped to specific signals as defined by the table below. *code* is a parameter that is either a constant as given below or zero. *scp* is a pointer to the *sigcontext* structure (defined in <*signal.h*>), that is the context at the time of the signal and is used to restore the context if the signal handler returns.

The following defines the mapping of MIPS hardware exceptions to signals and codes. All of these symbols are defined in either <*signal.h*> or <*sys/sbd.h*>:

Hardware exception	Signal	Code
Integer overflow	SIGFPE	EXC_OV
Segmentation violation	SIGSEGV	SEXC_SEGV
Illegal Instruction	SIGILL	EXC_II
Coprocessor Unusable	SIGILL	SEXC_CPU
Data Bus Error	SIGBUS	EXC_DBE
Instruction Bus Error	SIGBUS	EXC_IBE
Read Address Error	SIGBUS	EXC_RADE
Write Address Error	SIGBUS	EXC_WADE
User Breakpoint (used by debuggers)	SIGTRAP	BRK_USERBP
Kernel Breakpoint (used by prom)	SIGTRAP	BRK_KERNELBP
Taken Branch Delay Emulation	SIGTRAP	BRK_BD_TAKEN
Not Taken Branch Delay Emulation	SIGTRAP	BRK_BD_NOTTAKEN
User Single Step (used by debuggers)	SIGTRAP	BRK_SSTEPBP
Overflow Check	SIGTRAP	BRK_OVERFLOW
Divide by Zero Check	SIGTRAP	BRK_DIVZERO
Range Error Check	SIGTRAP	BRK_RANGE

When a signal handler is reached, the program counter in the signal context structure (*sc_pc*) points at the instruction that caused the exception as modified by the *branch delay* bit in the *cause* register. The *cause* register at the time of the exception is also saved in the sigcontext structure (*sc_cause*). If the instruction that caused the exception is at a valid user address it can be retrieved with the following code sequence:

```
if(scp->sc_cause & CAUSE_BD){
    branch_instruction = *(unsigned long *)(scp->sc_pc);
    exception_instruction = *(unsigned long *)(scp->sc_pc + 4);
}
else
    exception_instruction = *(unsigned long *)(scp->sc_pc);
```

Where CAUSE_BD is defined in <*sys/sbd.h*>.

The signal handler may fix the cause of the exception and re-execute the instruction, emulate the instruction and then step over it or perform some non-local goto such as a *longjump()* or an *exit()*.

If corrective action is performed in the signal handler and the instruction that caused the exception would then execute without a further exception, the signal handler simply returns and re-executes the instruction (even when the *branch delay* bit is set).

If execution is to continue after stepping over the instruction that caused the exception the program counter must be advanced. If the *branch delay* bit is set the program counter is set to the target of the branch else it is incremented by 4. This can be done with the following code sequence:

```
if(scp->sc_cause & CAUSE_BD)
    emulate_branch(scp, branch_instruction);
else
    scp->sc_pc += 4;
```

emulate_branch() modifies the program counter value in the sigcontext structure to the target of the branch instruction. See *emulate_branch*(3) for more details.

For SIGFPE's generated by floating-point instructions (*code* == 0) the *floating-point control and status* register at the time of the exception is also saved in the sigcontext structure (*sc_fpc_csr*). This register has the information on which exceptions have occurred. When a signal handler is entered the register contains the value at the time of the exception but with the *exceptions bits* cleared. On a return from the signal handler the exception bits in the floating-point control and status register are also cleared so that another SIGFPE will not occur (all other bits are restored from *sc_fpc_csr*).

If the floating-point unit is a R2360 (a floating-point board) and a SIGFPE is generated by the floating-point unit (*code* == 0) and program counter does not point at the instruction that caused the exception. In this case the instruction that caused the exception is in the *floating-point instruction exception* register. The floating-point instruction exception register at the time of the exception is also saved in the sigcontext structure (*sc_fpc_eir*). In this case the instruction that caused the exception can be retrieved with the following code sequence:

```
union fpc_irr fpc_irr;

fpc_irr.fi_word = get_fpc_irr();
if(sig == SIGFPE && code == 0 &&
  fpc_irr.fi_struct.implementation == IMPLEMENTATION_R2360)
   exception_instruction = scp->sc_fpc_eir;
```

The union *fpc_irr,* and the constant IMPLEMENTATION_R2360 are defined in *<sys/fpu.h>*. For the description of the routine *get_fpc_irr()* see *fpc*(3). All other floating-point implementations are handled in the normal manner with the instruction that caused the exception at the program counter as modified by the *branch delay* bit.

For SIGSEGV and SIGBUS errors the faulting virtual address is saved in *sc_badvaddr* in the signal context structure.

The SIGTRAP's caused by **break** instructions noted in the above table and all other yet to be defined **break** instructions fill the *code* parameter with the first argument to the **break** instruction (bits 25-16 of the instruction).

NAME

socket – create an endpoint for communication – TCP

SYNOPSIS

#include <bsd/sys/types.h>
#include <bsd/sys/socket.h>

s = socket(domain, type, protocol)
int s, domain, type, protocol;

DESCRIPTION

socket creates an endpoint for communication and returns a descriptor.

The *domain* parameter specifies a communications domain within which communication will take place; this selects the protocol family which should be used. The protocol family generally is the same as the address family for the addresses supplied in later operations on the socket. These families are defined in the include file *<bsd/sys/socket.h>*. The currently understood formats are:

PF_UNIX	(UNIX internal protocols),
PF_INET	(ARPA Internet protocols),
PF_NS	(Xerox Network Systems protocols), and
PF_IMPLINK	(IMP "host at IMP" link layer).

The socket has the indicated *type,* which specifies the semantics of communication. Currently defined types are:

SOCK_STREAM
SOCK_DGRAM
SOCK_RAW
SOCK_SEQPACKET
SOCK_RDM

A SOCK_STREAM type provides sequenced, reliable, two-way connection based byte streams. An out-of-band data transmission mechanism may be supported. A SOCK_DGRAM socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length). A SOCK_SEQPACKET socket may provide a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length; a consumer may be required to read an entire packet with each read system call. This facility is protocol specific, and presently implemented only for PF_NS. SOCK_RAW sockets provide access to internal network protocols and interfaces. The types SOCK_RAW , which is available only to the super-user, and SOCK_RDM , which is planned, but not yet implemented, are not described here.

The *protocol* specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is particular to the "communication domain" in which communication is to take place; see *protocols*(4).

Sockets of type SOCK_STREAM are full-duplex byte streams, similar to pipes. A stream socket must be in a *connected* state before any data may be sent or received on it. A connection to another socket is created with a *connect*(2) call. Once connected, data may be transferred using *read*(2) and *write*(2) calls or some variant of the *send*(2) and *recv*(2) calls. When a session has been completed a *close*(2) may be performed. Out-of-band data may also be transmitted as described in *send*(2) and received as described in *recv*(2).

The communications protocols used to implement a SOCK_STREAM insure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with −1 returns and with ETIMEDOUT as the specific code in the global variable errno. The protocols optionally keep sockets "warm" by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for a extended period (e.g. 5 minutes). A SIGPIPE signal is raised if a process sends on a broken stream; this causes naive processes, which do not handle the signal, to exit.

SOCK_SEQPACKET sockets employ the same system calls as SOCK_STREAM sockets. The only difference is that *read*(2) calls will return only the amount of data requested, and any remaining in the arriving packet will be discarded.

SOCK_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents named in *send*(2) calls. Datagrams are generally received with *recvfrom*(2), which returns the next datagram with its return address.

An *fcntl*(2) call can be used to specify a process group to receive a SIGURG signal when the out-of-band data It may also enable non-blocking I/O and asynchronous notification of I/O events via SIGIO.

The operation of sockets is controlled by socket level *options*. These options are defined in the file *<bsd/sys/socket.h>*. *setsockopt*(2) and *getsockopt*(2) are used to set and get options, respectively.

RETURN VALUE

A −1 is returned if an error occurs, otherwise the return value is a descriptor referencing the socket.

ERRORS

The *socket* call fails if:

[EPROTONOSUPPORT]	The protocol type or the specified protocol is not supported within this domain.
[EMFILE]	The per-process descriptor table is full.
[ENFILE]	The system file table is full.
[EACCESS]	Permission to create a socket of the specified type and/or protocol is denied.
[ENOBUFS]	Insufficient buffer space is available. The socket cannot be created until sufficient resources are freed.

SEE ALSO

accept(2), bind(2), connect(2), getsockname(2), getsockopt(2), ioctl(2), listen(2), read(2), recv(2), select(2), send(2), socketpair(2), write(2) "An Introductory 4.3BSD Interprocess Communication Tutorial", "An Advanced 4.3BSD Interprocess Comunication Tutorial."

ORIGIN

4.3 BSD

NAME

stat, lstat, fstat – get file status

SYNOPSIS

```
#include <sys/types.h>
#include lstat,<sys/stat.h>

int stat (path, buf)
char *path;
struct stat *buf;

int lstat (path, buf)
char *path;
struct stat *buf;

int fstat (fildes, buf)
int fildes;
struct stat *buf;
```

DESCRIPTION

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. *stat* obtains information about the named file. *lstat* is the same as *stat* except that when *path* names a symbolic link, the information retrieved is for the symbolic link instead of the file it points to.

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.

buf is a pointer to a *stat* structure into which information is placed concerning the file.

The contents of the structure pointed to by *buf* include the following members:

```
ushort   st_mode;   /* File mode [see mknod(2)] */
ino_t    st_ino;    /* Inode number */
dev_t    st_dev;    /* ID of device containing */
                    /* a directory entry for this file */
dev_t    st_rdev;   /* ID of device */
                    /* This entry is defined only for */
                    /* character special or block special files */
short    st_nlink;  /* Number of links */
ushort   st_uid;    /* User ID of the file's owner */
ushort   st_gid;    /* Group ID of the file's group */
off_t    st_size;   /* File size in bytes */
time_t   st_atime;  /* Time of last access */
time_t   st_mtime;  /* Time of last data modification */
time_t   st_ctime;  /* Time of last file status change */
                    /* Times measured in seconds since */
                    /* 00:00:00 GMT, Jan. 1, 1970 */
```

st_mode The mode of the file as described in the *mknod*(2) system call.

st_ino This field uniquely identifies the file in a given file system. The pair st_ino and st_dev uniquely identifies regular files.

st_dev This field uniquely identifies the file system that contains the file. Its value may be used as input to the *ustat*(2) system call to determine more information about this file system. No other meaning is associated with this value.

st_rdev This field should be used only by administrative commands. It is valid

only for block special or character special files and only has meaning on the system where the file was configured.

st_nlink This field should be used only by administrative commands.

st_uid The user ID of the file's owner.

st_gid The group ID of the file's group.

st_size For regular files, this is the address of the end of the file. For pipes or fifos, this is the count of the data currently in the file. For block special or character special, this is not defined.

st_atime Time when file data was last accessed. Changed by the following system calls: *creat*(2), *mknod*(2), *pipe*(2), *utime*(2), and *read*(2).

st_mtime Time when data was last modified. Changed by the following system calls: *creat*(2), *mknod*(2), *pipe*(2), *utime*(2), and *write*(2).

st_ctime Time when file status was last changed. Changed by the following system calls: *chmod*(2), *chown*(2), *creat*(2), *link*(2), *mknod*(2), *pipe*(2), *unlink*(2), *utime*(2), and *write*(2).

ERRORS

stat and *lstat* will fail if one or more of the following are true:

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

[ENOENT] The named file does not exist.

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

[EFAULT] *buf* or *path* points to an invalid address.

[EINTR] A signal was caught during the *stat* system call.

[ENOLINK] *path* points to a remote machine and the link to that machine is no longer active.

[EMULTIHOP] Components of *path* require hopping to multiple remote machines.

fstat will fail if one or more of the following are true:

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

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

[ENOLINK] *fildes* points to a remote machine and the link to that machine is no longer active.

SEE ALSO

chmod(2), chown(2), creat(2), link(2), mknod(2), pipe(2), read(2), time(2), unlink(2), utime(2), write(2).

DIAGNOSTICS

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

NAME

statfs, fstatfs – get file system information

SYNOPSIS

```
#include <sys/types.h>
#include <sys/statfs.h>

int statfs (path, buf, len, fstyp)
char *path;
struct statfs *buf;
int len, fstyp;

int fstatfs (fildes, buf, len, fstyp)
int fildes;
struct statfs *buf;
int len, fstyp;
```

DESCRIPTION

statfs returns a "generic superblock" describing a file system. It can be used to acquire information about mounted as well as unmounted file systems, and usage is slightly different in the two cases. In all cases, *buf* is a pointer to a structure (described below) which will be filled by the system call, and *len* is the number of bytes of information which the system should return in the structure. *len* must be no greater than **sizeof (struct statfs)** and ordinarily it will contain exactly that value; if it holds a smaller value the system will fill the structure with that number of bytes. (This allows future versions of the system to grow the structure without invalidating older binary programs.)

If the file system of interest is currently mounted, *path* can name a file which resides on that file system. In this case the file system type is known to the operating system and the *fstyp* argument must be zero. 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.

For either mounted or unmounted file systems, *path* can name the block special file for the partition containing the file system. In this case, the *fstype* argument must be set to the correct file system type.

The *statfs* structure pointed to by *buf* includes the following members:

```
short   f_fstyp;    /* File system type */
long    f_bsize;    /* Block size */
long    f_frsize;   /* Fragment size */
long    f_blocks;   /* Total number of blocks */
long    f_bfree;    /* Count of free blocks*/
long    f_files;    /* Total number of file nodes */
long    f_ffree;    /* Count of free file nodes */
char    f_fname[6]; /* Volume name */
char    f_fpack[6]; /* Pack name */
```

fstatfs is similar, except that the file named by *path* in *statfs* is instead identified by an open file descriptor *filedes* obtained from a successful *open*(2), *creat*(2), *dup*(2), *fcntl*(2), or *pipe*(2) system call.

statfs obsoletes *ustat*(2) and should be used in preference to it in new programs.

ERRORS

statfs and *fstatfs* will fail if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.

[EFAULT]	*buf* or *path* points to an invalid address.
[EBADF]	*fildes* is not a valid open file descriptor.
[EINVAL]	*fstyp* is an invalid file system type; *path* is not a block special file and *fstyp* is nonzero; *len* is negative or is greater than **sizeof (struct statfs)**.
[ENOLINK]	*path* points to a remote machine, and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

DIAGNOSTICS

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

SEE ALSO

chmod(2), chown(2), creat(2), link(2), mknod(2), pipe(2), read(2), time(2), unl ink(2), utime(2), write(2), fs(4).

NAME

stime – set time

SYNOPSIS

int stime (tp)
long *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 GMT January 1, 1970.

[EPERM] *stime* will fail if the effective user ID of the calling process is not super-user.

SEE ALSO

time(2).

DIAGNOSTICS

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

NAME

symlink – make symbolic link to a file

SYNOPSIS

symlink(name1, name2)
char *name1, *name2;

DESCRIPTION

A symbolic link *name2* is created to *na(name2isthenameofthe* file created, *name1* is the string used in creating the symbolic link). Either name may 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 on or more of the following are true:

[ENOTDIR]	A component of the *name2* prefix is not a directory.
[EINVAL]	Either *name1* or *name2* contains a character with the high-order bit set.
[ENAMETOOLONG]	A component of either pathname exceeded 255 characters, or the entire length of either path name exceeded 1023 characters.
[ENOENT]	The named file does not exist.
[EACCES]	A component of the *name2* path prefix denies search permission.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EEXIST]	*name2* already exists.
[EIO]	An I/O error occurred while making the directory entry for *name2*, or allocating the inode for *name2*, or writing out the link contents of *name2*.
[EROFS]	The file *name2* would reside 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 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.
[EDQUOT]	The directory in which the entry for the new symbolic link is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.
[EDQUOT]	The new symbolic link cannot be created because the user's quota of disk blocks on the file system that will contain the symbolic link has been exhausted.
[EDQUOT]	The user's quota of inodes on the file system on which the symbolic link is being created has been exhausted.
[EIO]	An I/O error occurred while making the directory entry or allocating the inode.
[EFAULT]	*Name1* or *name2* points outside the process's allocated address space.

SEE ALSO

link(2), ln(1), unlink(2)

NAME

sync – update super block

SYNOPSIS

void sync ( )

DESCRIPTION

sync causes all information in memory that should be on disk to be written out. This includes modified super blocks, modified i-nodes, and delayed block I/O.

It should be used by programs which examine a file system, for example *fsck*, *df*, etc. It is mandatory before a re-boot.

The writing, although scheduled, is not necessarily complete upon return from *sync*.

NAME

syscall – indirect system call

SYNOPSIS

#include <syscall.h>

syscall(number, arg, ...) (VAX-11)

DESCRIPTION

syscall performs the system call whose assembly language interface has the specified *number,* register arguments *r0* and *r1* and further arguments *arg*. Symbolic constants for system calls can be found in the header file *<syscall.h>*.

The r0 value of the system call is returned.

DIAGNOSTICS

When the C-bit is set, *syscall* returns –1 and sets the external variable *errno* (see *intro*(2)).

ERRORS

There is no way to simulate system calls such as *pipe*(2), which return values in register r1.

NAME

sysfs – get file system type information

SYNOPSIS

```
#include <sys/fstyp.h>
#include <sys/fsid.h>

int sysfs (opcode, fsname)
int opcode;
char *fsname;

int sysfs (opcode, fs_index, buf)
int opcode;
int fs_index;
char *buf;

int sysfs (opcode)
int opcode;
```

DESCRIPTION

sysfs returns information about the file system types configured in the system. The number of arguments accepted by *sysfs* varies and depends on the *opcode*. The currently recognized *opcodes* and their functions are described below:

GETFSIND	translates *fsname*, a null-terminated file-system identifier, into a file-system type index.
GETFSTYP	translates *fs_index*, a file-system type index, into a null-terminated file-system identifier and writes it into the buffer pointed to by *buf*; this buffer must be at least of size **FSTYPSZ** as defined in *<sys/fstyp.h>*.
GETNFSTYP	returns the total number of file system types configured in the system.

ERRORS

sysfs will fail if one or more of the following are true:

[EINVAL]	*fsname* points to an invalid file-system identifier; *fs_index* is zero, or invalid; *opcode* is invalid.
[EFAULT]	*buf* or *fsname* point to an invalid user address.

DIAGNOSTICS

Upon successful completion, *sysfs* returns the file-system type index if the *opcode* is **GETFSIND**, a value of 0 if the *opcode* is **GETFSTYP**, or the number of file system types configured if the *opcode* is **GETNFSTYP**. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

NAME

sysmips – machine specific functions

SYNOPSIS

#include <sys/sysmips.h>

int sysmips (cmd, arg1, arg2, arg3)
int cmd, arg1, arg2, arg3;

DESCRIPTION

sysmips implements machine specific functions. The *cmd* argument determines the function performed. The number of arguments expected is dependent on the function.

Command SETNAME

When *cmd* is SETNAME, an argument of type char * is expected. This points to a string that has a length less than SYS_NMLN (defined in *syslimts.h*). This function renames the system, which is sometimes referred to as the node name or host name. This feature is important for networking.

Command SMIPSSWPI

When *cmd* is SMIPSSWPI, individual swapping areas may be added, deleted or the current areas determined. The address of an appropriately primed swap buffer is passed as the only argument. (Refer to *sys/swap.h* header file for details of loading the buffer.)

The format of the swap buffer is:

```
struct swapint {
   char        si_cmd;        /*command: list, add, delete*/
   char        *si_buf;/*swap file path pointer*/
   int  si_swplo;      /*start block*/
   int  si_nblks;      /*swap size*/
}
```

Note that the add and delete options of the command may only be exercised by the super-user.

Typically, a swap area is added by a single call to *sysmips*. First, the swap buffer is primed with appropriate entries for the structure members. Then *sysmips* is invoked.

```
#include <sys/sysmips.h>
#include <sys/swap.h>

struct swapint swapbuf;          /*swap into buffer ptr*/

sysmips(SMIPSSWPI, &swapbuf);
```

If this command succeeds, it returns 0 to the calling process.

ERRORS

This command fails, returning -1, if one or more of the following is true:

[EFAULT]
Swapbuf points to an invalid address

[EFAULT]
swapbuf.si_buf points to an invalid address

[ENOTBLK]
Swap area specified is not a block special device

[EEXIST]
Swap area specified has already been added

[ENOSPC]
Too many swap areas in use (if adding)

[ENOMEM]
Tried to delete last remaining swap area

[EINVAL]
Bad arguments

[ENOMEM]
No place to put swapped pages when deleting a swap area

Command STIME
When *cmd* is STIME, an argument of type long is expected. This function sets the system time and date. The argument contains the time as measured in seconds from 00:00:00 GMT January 1, 1970. Note that this command is only available to the super-user.

Command FLUSH_CACHE
When command is FLUSH_CACHE , no arguments are expected. This function flushes both the instruction and data caches.

Command MIPS_FIXADE
When *cmd* is MIPS_FIXADE , an argument of type long is expected. This system call enables or disables kernel fix up of misaligned memory references. A non-zero argument enables and a zero argument disables this fix up. The MIPS hardware traps load and store operations where the address is not a multiple of the number of bytes loaded or stored. Usually this trap indicates incorrect program operation and so by default the kernel converts this trap into a SIGBUS signal to the process, typically causing a core dump for debugging.

Older programs developed on systems with lax alignment constraints sometimes make occasional misaligned references in course of correct operation. The best way to port such programs to MIPS hardware is to correct the program by aligning the data. A SIGBUS handler exists to assist the programmer in locating unaligned references. See *unaligned(3)*.

Some applications, however, must deal with unaligned data. The MIPS architecture provides special instructions, supported by builtin assembler macros, for loading and storing unaligned data. These applications can use these instructions where appropriate. Non-assembler programs can access these instructions via calls, also described in *unaligned(3)*.

When it is inappropriate to modify the application to either align the data properly, or to use special access methods for unaligned data, this system call, fixade, can be used as a method of last resort. This system call directs the kernel to handle misaligned traps and emulate an unaligned reference. The program no longer receives a SIGBUS signal. This emulation is slow, significantly slow down program execution.

If the program gets an address exception when making a reference outside its address space, it will still get a SIGBUS signal even if this is enabled.

Command MIPS_FPSIGINT
When *cmd* is MIPS_FPSIGINT , an argument of type long is expected. This system call causes every other floating-point interrupt to generate a SIGFPE signal. If the argument is 1 the next floating-point interrupt will cause a signal with the following one not causing a signal. If the argument is a 2 then the the next floating-point interrupt will not cause a signal with the following one causing a signal. If the argument is a 0 then the this feature is disabled and floating-point interrupts will not cause a signal.

This is intended for use by *fpi(3)* to analyze the causes of floating-point interrupts.

Command MIPS_KEYLOCKED
When *cmd* is MIPS_KEYLOCKED, no arguments are expected. If the system has a keyswitch, and the keyswitch is in the locked position, then this function returns 1. If the switch is not

locked, or if the system has no switch, the function returns 0.

SEE ALSO

sync(2), fpi(3), unaligned(3), a.out(4).
swap(1M) in the *System Administrator's Reference Manual.*
R2010 Floating Point Coprocessor Architecture
R2360 Floating Point Board Product Description

DIAGNOSTICS

Upon successful completion, the value returned is zero.

Otherwise, a value of −1 is returned and *errno* is set to indicate the error. When *cmd* is invalid, *errno* is set to EINVAL on return.

NAME

time – get time

SYNOPSIS

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

time_t time (tloc)
long *tloc;
```

DESCRIPTION

time returns the value of time in seconds since 00:00:00 GMT, January 1, 1970.

If *tloc* is non-zero, the return value is also stored in the location to which *tloc* points.

SEE ALSO

stime(2).

WARNING

time fails and its actions are undefined if *tloc* points to an illegal address.

DIAGNOSTICS

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

NAME

times – get process and child process times

SYNOPSIS

```
#include <sys/types.h>
#include <sys/times.h>

long times (buffer)
struct tms *buffer;
```

DESCRIPTION

times fills the structure pointed to by *buffer* with time-accounting information. The following are the contents of this structure:

```
struct  tms {
        time_t  tms_utime;
        time_t  tms_stime;
        time_t  tms_cutime;
        time_t  tms_cstime;
};
```

This information comes from the calling process and each of its terminated child processes for which it has executed a *wait*. All times are reported in clock ticks per second. Clock ticks are a system-dependent parameter. The specific value for an implementation is defined by the variable HZ, found in the include file param.h.

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_utime*s and *tms_cutime*s of the child processes.

tms_cstime is the sum of the *tms_stime*s and *tms_cstime*s of the child processes.

ERRORS

[EFAULT] *times* will fail if *buffer* points to an illegal address.

SEE ALSO

exec(2), fork(2), time(2), wait(2).

DIAGNOSTICS

Upon successful completion, *times* returns the elapsed real time, in clock ticks per second, from an arbitrary point in the past (e.g., system start-up time). This point does not change from one invocation of *times* to another. If *times* fails, a −1 is returned and *errno* is set to indicate the error.

NAME

truncate, ftruncate – truncate a file to a specified length

SYNOPSIS

```
truncate(path, length)
char *path;
off_t length;

ftruncate(fd, length)
int fd;
off_t length;
```

DESCRIPTION

truncate causes the file named by *path* or referenced by *fd* to be truncated to at most *length* bytes in size. If the file previously was larger than this size, the extra data is lost. With *ftruncate*, the file must be open for writing.

RETURN VALUES

A value of 0 is returned if the call succeeds. If the call fails a −1 is returned, and the global variable *errno* specifies the error.

ERRORS

truncate succeeds unless:

[ENOTDIR]	A component of the path prefix is not a directory.
[EINVAL]	The pathname contains a character with the high-order bit set.
[ENAMETOOLONG]	A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[EACCES]	The named file is not writable by the user.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[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.
[EIO]	An I/O error occurred updating the inode.
[EFAULT]	*path* points outside the process's allocated address space.

ftruncate succeeds unless:

[EBADF]	The *fd* is not a valid descriptor.
[EINVAL]	The *fd* references a socket, not a file.
[EINVAL]	The *fd* is not open for writing.

SEE ALSO

open(2)

BUGS

These calls should be generalized to allow ranges of bytes in a file to be discarded.

NAME

uadmin – administrative control

SYNOPSIS

#include <sys/uadmin.h>

int uadmin (cmd, fcn, mdep)
int cmd, fcn, mdep;

DESCRIPTION

uadmin provides control for basic administrative functions. This system call is tightly coupled to the system administrative procedures and is not intended for general use. The argument *mdep* is provided for machine-dependent use and is not defined here.

As specified by *cmd*, the following commands are available:

A_SHUTDOWN	The system is shutdown. All user processes are killed, the buffer cache is flushed, and the root file system is unmounted. The action to be taken after the system has been shut down is specified by *fcn*. The functions are generic; the hardware capabilities vary on specific machines.
AD_HALT	Halt the processor and turn off the power.
AD_BOOT	Reboot the system, using /unix.
AD_IBOOT	Interactive reboot; user is prompted for system name.
A_REBOOT	The system stops immediately without any further processing. The action to be taken next is specified by *fcn* as above.
A_REMOUNT	The root file system is mounted again after having been fixed. This should be used only during the startup process.

ERRORS

uadmin fails if any of the following are true:

[EPERM]	The effective user ID is not super-user.

DIAGNOSTICS

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

A_SHUTDOWN	Never returns.
A_REBOOT	Never returns.
A_REMOUNT	0

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

NAME

ulimit – get and set user limits

SYNOPSIS

long ulimit (cmd, newlimit)
int cmd;
long newlimit;

DESCRIPTION

This function provides for control over process limits. The *cmd* values available are:

Value	Action
1	Get the regular 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.
2	Set the regular file size limit of the process to the value of *newlimit*. Any process may decrease this limit, but only a process with an effective user ID of super-user may increase the limit. **Ulimit** fails and the limit is unchanged if a process with an effective user ID other than super-user attempts to increase its regular file size limit.
3	Get the maximum possible break value [see *brk(2)*].
4	Gets the maximum number of open files that a user can legally open.

SEE ALSO

brk(2), write(2).

WARNING

Ulimit is effective in limiting the growth of regular files. Pipes are currently limited to 5,120 bytes.

DIAGNOSTICS

Upon successful completion, a non-negative value is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

umask – set and get file creation mask

SYNOPSIS

int umask (cmask)
int cmask;

DESCRIPTION

umask sets the process's file mode creation mask to *cmask* and returns the previous value of the mask. Only the low-order 9 bits of *cmask* and the file mode creation mask are used.

SEE ALSO

chmod(2), creat(2), mknod(2), open(2).
mkdir(1), sh(1) in the *User's Reference Manual*.

DIAGNOSTICS

The previous value of the file mode creation mask is returned.

NAME

umount – unmount a file system

SYNOPSIS

int umount (file)
char *file;

DESCRIPTION

umount requests that a previously mounted file system contained on the block special device or directory identified by *file* be unmounted. *file* 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 may be invoked only by the super-user.

ERRORS

umount will fail if one or more of the following are true:

[EPERM]	The process's effective user ID is not super-user.
[EINVAL]	*file* does not exist.
[ENOTBLK]	*file* is not a block special device.
[EINVAL]	*file* is not mounted.
[EBUSY]	A file on *file* is busy.
[EFAULT]	*file* points to an illegal address.
[EREMOTE]	*file* is remote.
[ENOLINK]	*file* is on a remote machine, and the link to that machine is no longer active.
[EMULTIHOP]	Components of the path pointed to by *file* require hopping to multiple remote machines.

SEE ALSO

mount(2).

DIAGNOSTICS

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

NAME

uname – get general system information

SYNOPSIS

#include <sys/utsname.h>

int uname(un)
struct utsname *un;

DESCRIPTION

uname stores information identifying the current operating system and machine into the structure pointed to by the argument.

The *utsname* structure is defined in the include file <sys/utsname.h>. It consists of 13 fields, 7 of which are defined and the rest of which are reserved for future use. The currently defined fields (with available values) are:

sysname	The network identification name (same as the hostname).
nodename	The network identification name (same as the hostname and the above **sysname** field).
release	The operating system release name.
version	The MIPS system version number.
machine	The hardware type.
m_type	(MIPS-specific) The MIPS hardware type..TP **base_rel** (MIPS-specific) The base release for the system.

The valid values for these fields are defined in the *utsname.h* include file.

RETURN VALUE

If successful, *uname* will return a non-negative value; otherwise, it will return -1 and *errno* will indicate the error.

SEE ALSO

gethostname(2).

NAME

unlink – remove directory entry

SYNOPSIS

int unlink (path)
char *path;

DESCRIPTION

unlink removes the directory entry named by the path name pointed to by *path*.

ERRORS

The named file is not unlinked if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[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.
[EPERM]	The named file is a directory and the effective user ID of the process is not super-user.
[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.
[EINTR]	A signal was caught during the *unlink* system call.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

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, the removal is postponed until all references to the file have been closed.

SEE ALSO

close(2), link(2), open(2).
rm(1) in the *User's Reference Manual*.

DIAGNOSTICS

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

NAME

ustat – get file system statistics

SYNOPSIS

```
#include <sys/types.h>
#include <ustat.h>

int ustat (dev, buf)
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 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 */
```

ERRORS

ustat will fail if one or more of the following are 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.

[EINTR] A signal was caught during a *ustat* system call.

[ENOLINK] *dev* is on a remote machine and the link to that machine is no longer active.

[ECOMM] *dev* is on a remote machine and the link to that machine is no longer active.

SEE ALSO

stat(2), fs(4).

DIAGNOSTICS

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

NAME

utime – set file access and modification times

SYNOPSIS

```
#include <sys/types.h>
int utime (path, times)
char *path;
struct timeval tvp[2];
```

DESCRIPTION

path points to a path namel naming a file. *utime* sets the access and modification times of the named file.

The *utime* call uses the "acessed" and "updated" times in that order from the tvp vector to set the corresponding recorded times for *file*.

The caller must be the owner of the file or the super-user. The "inode-changed" time of the file is set to the current time.

ERRORS

utime will fail if one or more of the following are 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 super-user and not the owner of the file and *times* is not **NULL**.
[EACCES]	The effective user ID is not super-user and not the owner of the file and *times* is **NULL** and write access is denied.
[EROFS]	The file system containing the file is mounted read-only.
[EFAULT]	*times* is not **NULL** and points outside the process's allocated address space.
[EFAULT]	*path* points outside the process's allocated address space.
[EINTR]	A signal was caught during the *utime* system call.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

SEE ALSO

stat(2).

DIAGNOSTICS

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

NAME

wait, wait2 – wait for child process to stop or terminate

SYNOPSIS

```
int wait (stat_loc)
int *stat_loc;

#include <sys/wait.h>
int wait2 (stat_loc, options)
int *stat_loc, options;
```

DESCRIPTION

wait suspends the calling process until one of the immediate children terminates or until a child that is being traced stops, because it has hit a break point. The *wait* system call will return prematurely if a signal is received and if a child process stopped or terminated prior to the call on *wait*, return is immediate. *wait2* is similar to *wait*, except that it can be given options to effect its behavior.

If *stat_loc* is non-zero, 16 bits of information called status are stored in the low order 16 bits of the location pointed to by *stat_loc*. *status* can be used to differentiate between stopped and terminated child processes and if the child process terminated, status identifies the cause of termination and passes useful information to the parent. This is accomplished in the following manner:

If the child process stopped, the high order 8 bits of status will contain the number of the signal that caused the process to stop and the low order 8 bits will be set equal to 0177.

If the child process terminated due to an *exit* call, the low order 8 bits of status will be zero and the high order 8 bits will contain the low order 8 bits of the argument that the child process passed to *exit* [see *exit(2)*].

If the child process terminated due to a signal, the high order 8 bits of status will be zero and the low order 8 bits will contain the number of the signal that caused the termination. In addition, if the low order seventh bit (i.e., bit 200) is set, a "core image" will have been produced [see *signal(2)*].

The *options* argument is a flag which may have bits set to change the behavior of *wait2*. Setting the WNOHANG bit causes *wait2* to return immediately, even if no children are ready to be waited for. At the current time, only WNOHANG is implemented.

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 [see *intro(2)*].

ERRORS

wait will fail and return immediately if the following is true:

[ECHILD] The calling process has no existing unwaited-for child processes.

SEE ALSO

exec(2), exit(2), fork(2), intro(2), pause(2), ptrace(2), signal(2).

WARNING

wait fails and its actions are undefined if *stat_loc* points to an invalid address.

See NOTES in *signal(2)* and WARNING in *sigset(2)*.

DIAGNOSTICS

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 process, the process ID of the child is returned to the calling process. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

write – write on a file

SYNOPSIS

int write (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;

DESCRIPTION

fildes is a file descriptor obtained from a *creat*(2), *open*(2), *dup*(2), *fcntl*(2), or *pipe*(2) system call.

write attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the *fildes*.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from *write*, the file pointer is incremented by the number of bytes actually written.

On devices incapable of seeking, writing always takes place starting at the current position. The value of a file pointer associated with such a device is undefined.

If the O_APPEND flag of the file status flags is set, the file pointer will be set to the end of the file prior to each write.

For regular files, if the O_SYNC flag of the file status flags is set, the write will not return until both the file data and file status have been physically updated. This function is for special applications that require extra reliability at the cost of performance. For block special files, if O_SYNC is set, the write will not return until the data has been physically updated.

A write to a regular file will be blocked if mandatory file/record locking is set [see *chmod*(2)], and there is a record lock owned by another process on the segment of the file to be written. If O_NDELAY is not set, the write will sleep until the blocking record lock is removed.

For STREAMS [see *intro*(2)] files, the operation of *write* is determined by the values of the minimum and maximum *nbyte* range ("packet size") accepted by the *stream*. These values are contained in the topmost *stream* module. Unless the user pushes [see I_PUSH in *streamio*(7)] the topmost module, these values can not be set or tested from user level. If *nbyte* falls within the packet size range, *nbyte* bytes will be written. If *nbyte* does not fall within the range and the minimum packet size value is zero, *write* will break the buffer into maximum packet size segments prior to sending the data downstream (the last segment may contain less than the maximum packet size). If *nbyte* does not fall within the range and the minimum value is non-zero, *write* will fail with *errno* set to ERANGE. Writing a zero-length buffer (*nbyte* is zero) sends zero bytes with zero returned.

For STREAMS files, if O_NDELAY is not set and the *stream* can not accept data (the *stream* write queue is full due to internal flow control conditions), *write* will block until data can be accepted. O_NDELAY will prevent a process from blocking due to flow control conditions. If O_NDELAY is set and the *stream* can not accept data, *write* will fail. If O_NDELAY is set and part of the buffer has been written when a condition in which the *stream* can not accept additional data occurs, *write* will terminate and return the number of bytes written.

ERRORS

write will fail and the file pointer will remain unchanged if one or more of the following are true:

[EAGAIN] Mandatory file/record locking was set, O_NDELAY was set, and there was a blocking record lock.

[EAGAIN] Total amount of system memory available when reading via raw IO is

temporarily insufficient.

[EAGAIN] Attempt to write to a *stream* that can not accept data with the O_NDELAY flag set.

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

[EDEADLK] The write was going to go to sleep and cause a deadlock situation to occur.

[EFAULT] *buf* points outside the process's allocated address space.

[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* system call.

[EINVAL] Attempt to write to a *stream* linked below a multiplexor.

[ENOLCK] The system record lock table was full, so the write could not go to sleep until the blocking record lock was removed.

[ENOLINK] *fildes* is on a remote machine and the link to that machine is no longer active.

[ENOSPC] During a *write* to an ordinary file, there is no free space left on the device.

[ENXIO] A hangup occurred on the *stream* being written to.

[EPIPE and SIGPIPE signal] An attempt is made to write to a pipe that is not open for reading by any process.

[ERANGE] Attempt to write to a *stream* with *nbyte* outside specified minimum and maximum write range, and the minimum value is non-zero.

If a *write* requests that more bytes be written than there is room for (e.g., the *ulimit* [see *ulimit*(2)] or the physical end of a medium), only as many bytes as there is room for will be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512-bytes will return 20. The next write of a non-zero number of bytes will give a failure return (except as noted below).

If the file being written is a pipe (or FIFO) and the O_NDELAY flag of the file flag word is set, then write to a full pipe (or FIFO) will return a count of 0. Otherwise (O_NDELAY clear), writes to a full pipe (or FIFO) will block until space becomes available.

A write to a STREAMS file can fail if an error message has been received at the stream head. In this case, *errno* is set to the value included in the error message.

SEE ALSO

creat(2), dup(2), fcntl(2), intro(2), lseek(2), open(2), pipe(2), ulimit(2).

DIAGNOSTICS

Upon successful completion the number of bytes actually written is returned. Otherwise, −1 is returned and *errno* is set to indicate the error.

NAME

a64l, l64a – convert between long integer and base-64 ASCII string

SYNOPSIS

long a64l (s)
char *s;

char *l64a (l)
long 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.

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* will use the first six.

a64l scans the character string from left to right, decoding each character as a 6 bit Radix 64 number.

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.

CAVEAT

The value returned by *l64a* is a pointer into a static buffer, the contents of which are overwritten by each call.

NAME

abort – generate an IOT fault

SYNOPSIS

int abort ( )

DESCRIPTION

abort does the work of *exit*(2), but instead of just exiting, *abort* causes **SIGABRT** to be sent to the calling process. If **SIGABRT** is neither caught nor ignored, all *stdio*(3S) streams are flushed prior to the signal being sent, and a core dump results.

abort returns the value of the *kill*(2) system call.

SEE ALSO

exit(2), kill(2), signal(2).

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.

NAME

abort – terminate Fortran program

SYNOPSIS

call abort ( )

DESCRIPTION

abort terminates the program that calls it, closing all open files truncated to the current position of the file pointer. The abort usually results in a core cump.

DIAGNOSTICS

When invoked, *abort* prints "Fortran abort routine called" on the standard error output. The shell prints the message "abort - core dumped" if a core dump results.

SEE ALSO

abort(3C)
sh(1) in the *User's Reference Manual.*

NAME

abs – return integer absolute value

SYNOPSIS

int abs (i)
int i;

DESCRIPTION

abs returns the absolute value of its integer operand.

SEE ALSO

floor(3M).

CAVEAT

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.

NAME

abs, iabs, dabs, cabs, zabs – Fortran absolute value

SYNOPSIS

```
integer i1, i2
real r1, r2
double precision dp1, dp2
complex cx1, cx2
double complex dx1, dx2
r2 = abs(r1)
i2 = iabs(i1)
i2 = abs(i1)
dp2 = dabs(dp1)
dp2 = abs(dp1)
cx2 = cabs(cx1)
cx2 = abs(cx1)
dx2 = zabs(dx1)
dx2 = abs(dx1)
```

DESCRIPTION

abs is the family of absolute value functions. *iabs* returns the integer absolute value of its integer argument. *dabs* returns the double-precision absolute value of its double-precision argument. *cabs* returns the complex absolute value of its complex argument. *zabs* returns the double-complex absolute value of its double-complex argument. The generic form *abs* returns the type of its argument.

SEE ALSO

floor(3M).

NAME

access – determine accessibility of a file

SYNOPSIS

integer function access (name, mode)
character*(*) name, mode

DESCRIPTION

Access checks the given file, *name,* for accessibility with respect to the caller according to *mode. Mode* may include in any order and in any combination one or more of:

r	test for read permission
w	test for write permission
x	test for execute permission
(blank)	test for existence

An error code is returned if either argument is illegal, or if the file cannot be accessed in all of the specified modes. 0 is returned if the specified access would be successful.

FILES

/usr/lib/libU77.a

SEE ALSO

access(2), perror(3F)

BUGS

Pathnames can be no longer than MAXPATHLEN as defined in *<sys/param.h>*.

NAME

accept – accept a connection on a socket

SYNOPSIS

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

ns = accept(s, addr, addrlen)
int ns, s;
struct sockaddr *addr;
int *addrlen;
```

DESCRIPTION

The argument *s* is a socket that has been created with *socket*(2), bound to an address with *bind*(2), and is listening for connections after a *listen*(2). *accept* extracts the first connection on the queue of pending connections, creates a new socket with the same properties of *s* and allocates a new file descriptor, *ns*, for the socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, *accept* blocks the caller until a connection is present. 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*, may not be used to accept more connections. The original socket *s* remains open.

The argument *addr* is a result parameter that is filled in with the address of the connecting entity, as known to the communications layer. The exact format of the *addr* parameter is determined by the domain in which the communication is occurring. The *addrlen* is a value-result parameter; it should initially contain the amount of space pointed to by *addr*; on return it will contain the actual length (in bytes) of the address returned. This call is used with connection-based socket types, currently with SOCK_STREAM.

It is possible to *select*(2) a socket for the purposes of doing an *accept* by selecting it for read.

RETURN VALUE

The call returns −1 on error. If it succeeds, it returns a non-negative integer that is a descriptor for the accepted socket.

ERRORS

The *accept* will fail if:

[EBADF]	The descriptor is invalid.
[ENOTSOCK]	The descriptor references a file, not a socket.
[EOPNOTSUPP]	The referenced socket is not of type SOCK_STREAM.
[EFAULT]	The *addr* parameter is not in a writable part of the user address space.
[EWOULDBLOCK]	The socket is marked non-blocking and no connections are present to be accepted.

SEE ALSO

bind(2), connect(2), listen(2), select(2), socket(2)

NOTE

The primitives documented on this manual page are system calls, but unlike most system calls they are not resolved by libc.

ORIGIN

4.3 BSD

NAME

acos, dacos – Fortran arccosine intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
r2 = acos(r1)
dp2 = dacos(dp1)
dp2 = acos(dp1)
```

DESCRIPTION

acos returns the real arccosine of its real argument. *dacos* returns the double-precision arccosine of its double-precision argument. The generic form *acos* may be used with impunity as its argument will determine the type of the returned value.

NAME

aimag, dimag – Fortran imaginary part of complex argument

SYNOPSIS

```
real r
complex cxr
double precision dp
double complex cxd
r = aimag(cxr)
dp = dimag(cxd)
```

DESCRIPTION

aimag returns the imaginary part of its single-precision complex argument. *dimag* returns the double-precision imaginary part of its double-complex argument.

NAME

aint, dint – Fortran integer part intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
r2 = aint(r1)
dp2 = dint(dp1)
dp2 = aint(dp1)
```

DESCRIPTION

aint returns the truncated value of its real argument in a real. *dint* returns the truncated value of its double-precision argument as a double-precision value. *aint* may be used as a generic function name, returning either a real or double-precision value depending on the type of its argument.

NAME

alarm – execute a subroutine after a specified time

SYNOPSIS

integer function alarm (time, proc)
integer time
external proc

DESCRIPTION

This routine arranges for subroutine *proc* to be called after *time* seconds. If *time* is "0", the alarm is turned off and no routine will be called. The returned value will be the time remaining on the last alarm.

FILES

/usr/lib/libU77.a

SEE ALSO

alarm(3C), sleep(3F), signal(3F)

BUGS

Alarm and *sleep* interact. If *sleep* is called after *alarm*, the *alarm* process will never be called. SIGALRM will occur at the lesser of the remaining *alarm* time or the *sleep* time.

NAME

asin, dasin – Fortran arcsine intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
r2 = asin(r1)
dp2 = dasin(dp1)
dp2 = asin(dp1)
```

DESCRIPTION

asin returns the real arcsine of its real argument. *dasin* returns the double-precision arcsine of its double-precision argument. The generic form *asin* may be used with impunity as it derives its type from that of its argument.

NAME

asinh, acosh, atanh – inverse hyperbolic functions

SYNOPSIS

```
#include <math.h>

double asinh(x)
double x;

double acosh(x)
double x;

double atanh(x)
double x;
```

DESCRIPTION

These functions compute the designated inverse hyperbolic functions for real arguments.

ERROR (due to Roundoff etc.)

These functions inherit much of their error from log1p described in exp(3M).

DIAGNOSTICS

Acosh returns the default quiet *NaN* if the argument is less than 1.

Atanh returns the default quiet *NaN* if the argument has absolute value bigger than or equal to 1.

SEE ALSO

math(3M), exp(3M)

AUTHOR

W. Kahan, Kwok–Choi Ng

NAME

assert – verify program assertion

SYNOPSIS

#include <assert.h>

assert (expression)
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, will stop assertions from being compiled into the program.

SEE ALSO

cpp(1), abort(3C).

CAVEAT

Since *assert* is implemented as a macro, the *expression* may not contain any string literals.

NAME

atan, datan – Fortran arctangent intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
r2 = atan(r1)
dp2 = datan(dp1)
dp2 = atan(dp1)
```

DESCRIPTION

atan returns the real arctangent of its real argument. *datan* returns the double-precision arctangent of its double-precision argument. The generic form *atan* may be used with a double-precision argument returning a double-precision value.

NAME

atan2, datan2 – Fortran arctangent intrinsic function

SYNOPSIS

real r1, r2, r3
double precision dp1, dp2, dp3
r3 = atan2(r1, r2)
dp3 = datan2(dp1, dp2)
dp3 = atan2(dp1, dp2)

DESCRIPTION

atan2 returns the arctangent of *arg1/arg2* as a real value. *datan2* returns the double-precision arctangent of its double-precision arguments. The generic form *atan2* may be used with impunity with double-precision arguments.

NAME

bool: and, or, xor, not, lshift, rshift – Fortran Bitwise Boolean functions

SYNOPSIS

```
integer i, j, k
real a, b, c

k = and(i, j)
c = or(a, b)
j = xor(i, a)
j = not(i)
k = lshift(i, j)
k = rshift(i, j)
```

DESCRIPTION

The generic intrinsic Boolean functions *and*, *or* and *xor* return the value of the binary operations on their arguments. *not* is a unary operator returning the one's complement of its argument. *lshift* and *rshift* return the value of the first argument shifted left or right, respectively, the number of times specified by the second (integer) argument. While it is recommended that Boolean functions be used only on integer data, these functions are generic; that is, they are defined for all data types as arguments and return values. Where required, the compiler generates appropriate type conversions. However, when the functions are not used with integer data, the results are unpredictable.

ERRORS

The implementation of the shift functions may cause large shift values to deliver weird results.

SEE ALSO

mil(3F).

NAME

bsearch – binary search a sorted table

SYNOPSIS

```
#include <search.h>

char *bsearch ((char *) key, (char *) base, nel, sizeof (*key), compar)
unsigned nel;
int (*compar)( );
```

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. *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 as accordingly the first argument is to be considered less than, equal to, or greater than the second.

EXAMPLE

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>
#include <search.h>

#defineTABSIZE      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((char *)(&node),
                        (char *)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)
        char *node1, *node2;
        {
                return (strcmp(
                        ((struct node *)node1)->string,
                        ((struct node *)node2)->string));
        }
```

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 *bsearch* is declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

SEE ALSO

hsearch(3C), lsearch(3C), qsort(3C), tsearch(3C).

DIAGNOSTICS

A NULL pointer is returned if the key cannot be found in the table.

NAME

bcopy, bcmp, bzero, ffs – bit and byte string operations

SYNOPSIS

```
bcopy(src, dst, length)
char *src, *dst;
int length;

bcmp(b1, b2, length)
char *b1, *b2;
int length;

bzero(b, length)
char *b;
int length;

ffs(i)
int i;
```

DESCRIPTION

The functions *bcopy*, *bcmp*, and *bzero* operate on variable length strings of bytes. They do not check for null bytes as the routines in *string*(3) do.

bcopy copies *length* bytes from string *src* to the string *dst*.

bcmp compares byte string *b1* against byte string *b2*, returning zero if they are identical, non-zero otherwise. Both strings are assumed to be *length* bytes long.

bzero places *length* 0 bytes in the string *b1*.

ffs find the first bit set in the argument passed it and returns the index of that bit. Bits are numbered starting at 1. A return value of 0 indicates the value passed is zero.

ERRORS

The *bcopy* routine take parameters backwards from *strcpy*.

ORIGINS

BSD4.3

NAME

htonl, htons, ntohl, ntohs – convert values between host and network byte order

SYNOPSIS

```
#include <sys/types.h>
#include </bsd/netinet/in.h>

netlong = htonl(hostlong);
u_long netlong, hostlong;

netshort = htons(hostshort);
u_short netshort, hostshort;

hostlong = ntohl(netlong);
u_long hostlong, netlong;

hostshort = ntohs(netshort);
u_short hostshort, netshort;
```

DESCRIPTION

These routines convert 16 and 32 bit quantities between network byte order and host byte order. 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 *gethostbyname*(3N) and *getservent*(3N).

SEE ALSO

gethostbyname(3N), getservent(3N)

BUGS

This is not expected to be fixed in the near future.

NAME

chdir – change default directory

SYNOPSIS

integer function chdir (dirname)
character*(*) dirname

DESCRIPTION

The default directory for creating and locating files will be changed to *dirname*. Zero is returned if successful; an error code otherwise.

FILES

/usr/lib/libU77.a

SEE ALSO

chdir(2), cd(1), perror(3F)

BUGS

Pathnames can be no longer than MAXPATHLEN as defined in <*sys/param.h*>.

Use of this function may cause **inquire** by unit to fail.

NAME

chmod – change mode of a file

SYNOPSIS

integer function chmod (name, mode)
character*(*) name, mode

DESCRIPTION

This function changes the filesystem *mode* of file *name*. *Mode* can be any specification recognized by *chmod*(1). *Name* must be a single pathname.

The normal returned value is 0. Any other value will be a system error number.

FILES

/usr/lib/libU77.a
/bin/chmod exec'ed to change the mode.

SEE ALSO

chmod(1)

BUGS

Pathnames can be no longer than MAXPATHLEN as defined in <*sys/param.h*>.

NAME

clock – report CPU time used

SYNOPSIS

long clock ( )

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*(2), or *system*(3S).

SEE ALSO

times(2), wait(2), popen(3S), system(3S).

ERRORS

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 will wrap around after accumulating only 2147 seconds of CPU time (about 36 minutes).

NAME

conjg, dconjg – Fortran complex conjugate intrinsic function

SYNOPSIS

complex cx1, cx2
double complex dx1, dx2
cx2 = conjg(cx1)
dx2 = dconjg(dx1)

DESCRIPTION

conjg returns the complex conjugate of its complex argument. *dconjg* returns the double-complex conjugate of its double-complex argument.

NAME

cos, dcos, ccos – Fortran cosine intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
complex cx1, cx2
r2 = cos(r1)
dp2 = dcos(dp1)
dp2 = cos(dp1)
cx2 = ccos(cx1)
cx2 = cos(cx1)
```

DESCRIPTION

cos returns the real cosine of its real argument. *dcos* returns the double-precision cosine of its double-precision argument. *ccos* returns the complex cosine of its complex argument. The generic form *cos* may be used with impunity as its returned type is determined by that of its argument.

NAME

cosh, dcosh – Fortran hyperbolic cosine intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
r2 = cosh(r1)
dp2 = dcosh(dp1)
dp2 = cosh(dp1)
```

DESCRIPTION

cosh returns the real hyperbolic cosine of its real argument. *dcosh* returns the double-precision hyperbolic cosine of its double-precision argument. The generic form *cosh* may be used to return the hyperbolic cosine in the type of its argument.

SEE ALSO

sinh(3M).

NAME

crypt, setkey, encrypt – generate hashing encryption

SYNOPSIS

```
char *crypt (key, salt)
char *key, *salt;

void setkey (key)
char *key;

void encrypt (block, ignored)
char *block;
int ignored;
```

DESCRIPTION

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.

The *setkey* and *encrypt* entries provide (rather primitive) access to the actual hashing algorithm. The argument of *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 will be used with the hashing algorithm to encrypt the string *block* with the function *encrypt*.

The argument to the *encrypt* entry 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 set by *setkey*. *ignored* is unused by *encrypt* but it must be present.

SEE ALSO

getpass(3C), passwd(4).
login(1), passwd(1) in the *User's Reference Manual.*

CAVEAT

The return value points to static data that are overwritten by each call.

NAME

crypt – password and file encryption functions

SYNOPSIS

cc [flag ...] file ... −lcrypt

```
char *crypt (key, salt)
char *key, *salt;

void setkey (key)
char *key;

void encrypt (block, flag)
char *block;
int flag;

char *des_crypt (key, salt)
char *key, *salt;

void des_setkey (key)
char *key;

void des_encrypt (block, flag)
char *block;
int flag;

int run_setkey (p, key)
int p[2];
char *key;

int run_crypt (offset, buffer, count, p)
long offset;
char *buffer;
unsigned int count;
int p[2];

int crypt_close(p)
int p[2];
```

DESCRIPTION

des_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.

The *des_setkey* and *des_encrypt* entries provide (rather primitive) access to the actual hashing algorithm. The argument of *des_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 will be used with the hashing algorithm to encrypt the string *block* with the function *des_encrypt*.

The argument to the *des_encrypt* entry 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 set by *des_setkey*. If *edflag* is zero, the argument is encrypted; if non-zero, it is decrypted.

Note that decryption is not provided in the international version of *crypt*(3X). The international version is part of the *C Programming Language Utilities*, and the domestic version is part of the *Security Administration Utilities*. If decryption is attempted with the international version of *des_encrypt*, an error message is printed.

crypt, *setkey*, and *encrypt* are front-end routines that invoke *des_crypt*, *des_setkey*, and *des_encrypt* respectively.

The routines *run_setkey* and *run_crypt* are designed for use by applications that need cryptographic capabilities [such as *ed*(1) and *vi*(1)] that must be compatible with the *crypt*(1) user-level utility. *eun_setkey* establishes a two-way pipe connection with *crypt*(1), using *key* as the password argument. *run_crypt* takes a block of characters and transforms the cleartext or ciphertext into their ciphertext or cleartext using *crypt*(1). *offset* is the relative byte position from the beginning of the file that the block of text provided in *block* is coming from. *count* is the number of characters in *block*, and *connection* is an array containing indices to a table of input and output file streams. When encryption is finished, *crypt_close* is used to terminate the connection with *crypt*(1).

run_setkey returns -1 if a connection with *crypt*(1) cannot be established. This will occur on international versions of UNIX where *crypt*(1) is not available. If a null key is passed to *run_setkey*, 0 is returned. Otherwise, 1 is returned. *run_crypt* returns -1 if it cannot write output or read input from the pipe attached to *crypt*. Otherwise it returns 0.

DIAGNOSTICS

In the international version of *crypt*(3X), a flag argument of 1 to *des_encrypt* is not accepted, and an error message is printed.

SEE ALSO

getpass(3C), passwd(4).
crypt(1), login(1), passwd(1) in the *User's Reference Manual*.

CAVEAT

The return value in *crypt* points to static data that are overwritten by each call.

NAME

ctermid – generate file name for terminal

SYNOPSIS

#include <stdio.h>
char *ctermid (s)
char *s;

DESCRIPTION

ctermid generates the path name of the controlling terminal for the current process, and stores it in a string.

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.

NOTES

The difference between *ctermid* and *ttyname*(3C) 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 will refer to the terminal if used as a file name. Thus *ttyname* is useful only if the process already has at least one file open to a terminal.

SEE ALSO

ttyname(3C).

NAME

ctime, localtime, gmtime, asctime, tzset – convert date and time to string

SYNOPSIS

```
#include <sys/types.h>
#include <time.h>

char *ctime (clock)
time_t *clock;

struct tm *localtime (clock)
time_t *clock;

struct tm *gmtime (clock)
time_t *clock;

char *asctime (tm)
struct tm *tm;

extern long timezone;

extern int daylight;

extern char *tzname[2];

void tzset ( )
```

DESCRIPTION

ctime converts a long integer, pointed to by *clock*, representing the time in seconds since 00:00:00 GMT, January 1, 1970, and returns a pointer to a 26-character string in the following form. All the fields have constant width.

```
Sun Sep 16 01:03:52 1985\n\0
```

localtime and *gmtime* return pointers to "tm" structures, described below. *localtime* corrects for the time zone and possible Daylight Savings Time; *gmtime* converts directly to Greenwich Mean Time (GMT), which is the time the UNIX system uses.

asctime converts a "tm" structure to a 26-character string, as shown in the above example, and returns a pointer to the string.

Declarations of all the functions and externals, and the "tm" structure, are in the *<time.h>* header file. The structure declaration is:

```
struct tm {
        int tm_sec;     /* seconds (0 - 59) */
        int tm_min;     /* minutes (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;    /* year - 1900 */
        int tm_wday;    /* day of week (Sunday = 0) */
        int tm_yday;    /* day of year (0 - 365) */
        int tm_isdst;
};
```

tm_isdst is non-zero if Daylight Savings Time is in effect.

The external **long** variable *timezone* contains the difference, in seconds, between GMT and local standard time (in EST, *timezone* is 5*60*60); the external variable *daylight* is non-zero if and only if the standard U.S.A. Daylight Savings Time conversion should be applied. The program knows about the peculiarities of this conversion in 1974 and 1975; if necessary, a table for these years can be extended.

If an environment variable named **TZ** is present, *asctime* uses the contents of the variable to override the default time zone. The value of **TZ** must be a three-letter time zone name, followed by a number representing the difference between local time and Greenwich Mean Time in hours, followed by an optional three-letter name for a daylight time zone. For example, the setting for New Jersey would be **EST5EDT**. The effects of setting **TZ** are thus to change the values of the external variables *timezone* and *daylight*; in addition, the time zone names contained in the external variable

```
char *tzname[2] = { "EST", "EDT" };
```

are set from the environment variable **TZ**. The function *tzset* sets these external variables from **TZ**; *tzset* is called by *asctime* and may also be called explicitly by the user.

Note that in most installations, **TZ** is set by default when the user logs on, to a value in the local /etc/profile file [see *profile*(4)].

SEE ALSO

time(2), getenv(3C), profile(4), environ(5).

CAVEAT

The return values point to static data whose content is overwritten by each call.

NAME

ctype: isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii – classify characters

SYNOPSIS

#include <ctype.h>

int isalpha (c)
int c;

. . .

DESCRIPTION

These macros classify character-coded integer values by table lookup. Each is a predicate returning nonzero for true, zero for false. *isascii* is defined on all integer values; the rest are defined only where *isascii* is true and on the single non-ASCII value **EOF** [−1; see *stdio*(3S)].

isalpha	*c* is a letter.
isupper	*c* is an upper-case letter.
islower	*c* is a lower-case letter.
isdigit	*c* is a digit [0-9].
isxdigit	*c* is a hexadecimal digit [0-9], [A-F] or [a-f].
isalnum	*c* is an alphanumeric (letter or digit).
isspace	*c* is a space, tab, carriage return, newline, vertical tab, or form-feed.
ispunct	*c* is a punctuation character (neither control nor alphanumeric).
isprint	*c* is a printing character, code 040 (space) through 0176 (tilde).
isgraph	*c* is a printing character, like *isprint* except false for space.
iscntrl	*c* is a delete character (0177) or an ordinary control character (less than 040).
isascii	*c* is an ASCII character, code less than 0200.

SEE ALSO

stdio(3S), ascii(5).

DIAGNOSTICS

If the argument to any of these macros is not in the domain of the function, the result is undefined.

NAME

curses – terminal screen handling and optimization package

NOTE:

The **curses** manual page is organized as follows:

In SYNOPSIS:
- compiling information
- summary of parameters used by **curses** routines
- alphabetical list of curses routines, showing their parameters

In DESCRIPTION:
- An overview of how **curses** routines should be used

In ROUTINES, descriptions of each **curses** routines, are grouped under the appropriate topics:
- Overall Screen Manipulation
- Window and Pad Manipulation
- Output
- Input
- Output Options Setting
- Input Options Setting
- Environment Queries
- Soft Labels
- Low-level Curses Access
- Terminfo-Level Manipulations
- Termcap Emulation
- Miscellaneous
- Use of **curscr**

Followed by sections on:
- ATTRIBUTES
- FUNCTION CALLS
- LINE GRAPHICS

SYNOPSIS

cc [flag . . .] file . . . **−lcurses** [library . . .]

#include <curses.h> (automatically includes **<stdio.h>**, **<termio.h>**, and **<unctrl.h>**).

The parameters in the following list are not global variables, but rather this is a summary of the parameters used by the **curses** library routines. All routines return the **int** values **ERR** or **OK** unless otherwise noted. Routines that return pointers always return **NULL** on error. (**ERR**, **OK**, and **NULL** are all defined in **<curses.h>**.) Routines that return integers are not listed in the parameter list below.

bool bf

char **area,*boolnames[ ], *boolcodes[ ], *boolfnames[ ], *bp
char *cap, *capname, codename[2], erasechar, *filename, *fmt
char *keyname, killchar, *label, *longname
char *name, *numnames[ ], *numcodes[ ], *numfnames[ ]
char *slk_label, *str, *strnames[ ], *strcodes[ ], *strfnames[ ]

char *term, *tgetstr, *tigetstr, *tgoto, *tparm, *type

chtype attrs, ch, horch, vertch

FILE *infd, *outfd

int begin_x, begin_y, begline, bot, c, col, count
int dmaxcol, dmaxrow, dmincol, dminrow, *errret, fildes
int (*init( )), labfmt, labnum, line
int ms, ncols, new, newcol, newrow, nlines, numlines
int oldcol, oldrow, overlay
int p1, p2, p9, pmincol, pminrow, (*putc( )), row
int smaxcol, smaxrow, smincol, sminrow, start
int tenths, top, visibility, x, y

SCREEN *new, *newterm, *set_term

TERMINAL *cur_term, *nterm, *oterm

va_list varglist

WINDOW *curscr, *dstwin, *initscr, *newpad, *newwin, *orig
WINDOW *pad, *srcwin, *stdscr, *subpad, *subwin, *win

addch(ch)
addstr(str)
attroff(attrs)
attron(attrs)
attrset(attrs)
baudrate( )
beep( )
box(win, vertch, horch)
cbreak( )
clear( )
clearok(win, bf)
clrtobot( )
clrtoeol( )
copywin(srcwin, dstwin, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol, overlay)
curs_set(visibility)
def_prog_mode( )
def_shell_mode( )
del_curterm(oterm)
delay_output(ms)
delch( )
deleteln( )
delwin(win)
doupdate( )
draino(ms)
echo( )
echochar(ch)
endwin( )
erase( )
erasechar( )
filter( )
flash( )
flushinp( )
garbagedlines(win, begline, numlines)
getbegyx(win, y, x)

getch( )
getmaxyx(win, y, x)
getstr(str)
getsyx(y, x)
getyx(win, y, x)
halfdelay(tenths)
has_ic( )
has_il( )
idlok(win, bf)
inch( )
initscr( )
insch(ch)
insertln( )
intrflush(win, bf)
isendwin( )
keyname(c)
keypad(win, bf)
killchar( )
leaveok(win, bf)
longname( )
meta(win, bf)
move(y, x)
mvaddch(y, x, ch)
mvaddstr(y, x, str)
mvcur(oldrow, oldcol, newrow, newcol)
mvdelch(y, x)
mvgetch(y, x)
mvgetstr(y, x, str)
mvinch(y, x)
mvinsch(y, x, ch)
mvprintw(y, x, fmt [, arg . . .])
mvscanw(y, x, fmt [, arg . . .])
mvwaddch(win, y, x, ch)
mvwaddstr(win, y, x, str)
mvwdelch(win, y, x)
mvwgetch(win, y, x)
mvwgetstr(win, y, x, str)
mvwin(win, y, x)
mvwinch(win, y, x)
mvwinsch(win, y, x, ch)
mvwprintw(win, y, x, fmt [, arg . . .])
mvwscanw(win, y, x, fmt [, arg . . .])
napms(ms)
newpad(nlines, ncols)
newterm(type, outfd, infd)
newwin(nlines, ncols, begin_y, begin_x)
nl( )
nocbreak( )
nodelay(win, bf)
noecho( )
nonl( )
noraw( )

notimeout(win, bf)
overlay(srcwin, dstwin)
overwrite(srcwin, dstwin)
pechochar(pad, ch)
pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)
prefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)
printw(fmt [, arg . . .])
putp(str)
raw( )
refresh( )
reset_prog_mode( )
reset_shell_mode( )
resetty( )
restartterm(term, fildes, errret)
ripoffline(line, init)
savetty( )
scanw(fmt [, arg . . .])
scr_dump(filename)
scr_init(filename)
scr_restore(filename)
scroll(win)
scrollok(win, bf)
set_curterm(nterm)
set_term(new)
setscrreg(top, bot)
setsyx(y, x)
setupterm(term, fildes, errret)
slk_clear( )
slk_init(fmt)
slk_label(labnum)
slk_noutrefresh( )
slk_refresh( )
slk_restore( )
slk_set(labnum, label, fmt)
slk_touch( )
standend( )
standout( )
subpad(orig, nlines, ncols, begin_y, begin_x)
subwin(orig, nlines, ncols, begin_y, begin_x)
tgetent(bp, name)
tgetflag(codename)
tgetnum(codename)
tgetstr(codename, area)
tgoto(cap, col, row)
tigetflag(capname)
tigetnum(capname)
tigetstr(capname)
touchline(win, start, count)
touchwin(win)
tparm(str, p1, p2, . . ., p9)
tputs(str, count, putc)
traceoff( )

traceon( )
typeahead(fildes)
unctrl(c)
ungetch(c)
vidattr(attrs)
vidputs(attrs, putc)
vwprintw(win, fmt, varglist)
vwscanw(win, fmt, varglist)
waddch(win, ch)
waddstr(win, str)
wattroff(win, attrs)
wattron(win, attrs)
wattrset(win, attrs)
wclear(win)
wclrtobot(win)
wclrtoeol(win)
wdelch(win)
wdeleteln(win)
wechochar(win, ch)
werase(win)
wgetch(win)
wgetstr(win, str)
winch(win)
winsch(win, ch)
winsertln(win)
wmove(win, y, x)
wnoutrefresh(win)
wprintw(win, fmt [, arg . . .])
wrefresh(win)
wscanw(win, fmt [, arg . . .])
wsetscrreg(win, top, bot)
wstandend(win)
wstandout(win)

DESCRIPTION

The **curses** routines give the user a terminal-independent method of updating screens with reasonable optimization.

In order to initialize the routines, the routine **initscr**( ) or **newterm**( ) must be called before any of the other routines that deal with windows and screens are used. (Three exceptions are noted where they apply.) The routine **endwin**( ) must be called before exiting. To get character-at-a-time input without echoing, (most interactive, screen oriented programs want this) after calling **initscr**( ) you should call **"cbreak**( ); **noecho**( );" Most programs would additionally call **"nonl**( ); **intrflush (stdscr, FALSE); keypad(stdscr, TRUE);"**. Before a **curses** program is run, a terminal's tab stops should be set and its initialization strings, if defined, must be output. This can be done by executing the **tput init** command after the shell environment variable **TERM** has been exported. For further details, see *profile(4)*, *tput(1)*, and the "Tabs and Initialization" subsection of *terminfo(4)*.

The **curses** library contains routines that manipulate data structures called *windows* that can be thought of as two-dimensional arrays of characters representing all or part of a terminal screen. A default window called **stdscr** is supplied, which is the size of the terminal screen. Others may be created with **newwin**( ). Windows are referred to by variables declared as **WINDOW** *; the type **WINDOW** is defined in **<curses.h>** to be a C structure. These data

structures are manipulated with routines described below, among which the most basic are **move**( ) and **addch**( ). (More general versions of these routines are included with names beginning with **w**, allowing you to specify a window. The routines not beginning with **w** usually affect **stdscr**.) Then **refresh**( ) is called, telling the routines to make the user's terminal screen look like **stdscr**. The characters in a window are actually of type **chtype**, so that other information about the character may also be stored with each character.

Special windows called *pads* may also be manipulated. These are windows which are not constrained to the size of the screen and whose contents need not be displayed completely. See the description of **newpad**( ) under "Window and Pad Manipulation" for more information.

In addition to drawing characters on the screen, video attributes may be included which cause the characters to show up in modes such as underlined or in reverse video on terminals that support such display enhancements. Line drawing characters may be specified to be output. On input, **curses** is also able to translate arrow and function keys that transmit escape sequences into single values. The video attributes, line drawing characters, and input values use names, defined in **<curses.h>**, such as **A_REVERSE, ACS_HLINE,** and **KEY_LEFT. Curses** also defines the **WINDOW** * variable, **curscr**, which is used only for certain low-level operations like clearing and redrawing a garbaged screen. **curscr** can be used in only a few routines. If the window argument to **clearok**( ) is **curscr**, the next call to **wrefresh**( ) with any window will cause the screen to be cleared and repainted from scratch. If the window argument to **wrefresh**( ) is **curscr**, the screen in immediately cleared and repainted from scratch. This is how most programs would implement a "repaint-screen" function. More information on using **curscr** is provided where its use is appropriate.

The environment variables **LINES** and **COLUMNS** may be set to override **terminfo**'s idea of how large a screen is. These may be used in an AT&T Teletype 5620 layer, for example, where the size of a screen is changeable. If the environment variable **TERMINFO** is defined, any program using **curses** will check for a local terminal definition before checking in the standard place. For example, if the environment variable **TERM** is set to **att4425**, then the compiled terminal definition is found in */usr/lib/terminfo/a/att4425*. (The **a** is copied from the first letter of **att4425** to avoid creation of huge directories.) However, if **TERMINFO** is set to *$HOME/myterms*, **curses** will first check *$HOME/myterms/a/att4425*, and, if that fails, will then check */usr/lib/terminfo/a/att4425*. This is useful for developing experimental definitions or when write permission on */usr/lib/terminfo* is not available.

The integer variables **LINES** and **COLS** are defined in **<curses.h>**, and will be filled in by **initscr**( ) with the size of the screen. (For more information, see the subsection "Terminfo-Level Manipulations".) The constants **TRUE** and **FALSE** have the values **1** and **0**, respectively. The constants **ERR** and **OK** are returned by routines to indicate whether the routine successfully completed. These constants are also defined in **<curses.h>**.

ROUTINES

Many of the following routines have two or more versions. The routines prefixed with **w** require a *window* argument. The routines prefixed with **p** require a *pad* argument. Those without a prefix generally use **stdscr**.

The routines prefixed with **mv** require *y* and *x* coordinates to move to before performing the appropriate action. The **mv**( ) routines imply a call to **move**( ) before the call to the other routine. The window argument is always specified before the coordinates. *y* always refers to the row (of the window), and *x* always refers to the column. The upper left corner is always **(0,0)**, not **(1,1)**. The routines prefixed with **mvw** take both a *window* argument and *y* and *x* coordinates.

In each case, *win* is the window affected and *pad* is the pad affected. (**Win** and **pad** are always of type **WINDOW** *.) Option-setting routines require a boolean flag *bf* with the value **TRUE** or **FALSE.** (*bf* is always of type **bool**.) The types **WINDOW**, **bool**, and **chtype** are defined

in <**curses.h**>. See the SYNOPSIS for a summary of what types all variables are.

All routines return either the integer **ERR** or the integer **OK**, unless otherwise noted. Routines that return pointers always return **NULL** on error.

Overall Screen Manipulation

WINDOW *initscr( ) The first routine called should almost always be **initscr( )**. (The exceptions are **slk_init( )**, **filter( )**, and **ripoffline( ).**) This will determine the terminal type and initialize all **curses** data structures. **initscr( )** also arranges that the first call to **refresh( )** will clear the screen. If errors occur, **initscr( )** will write an appropriate error message to standard error and exit; otherwise, a pointer to **stdscr** is returned. If the program wants an indication of error conditions, **newterm( )** should be used instead of **initscr( )**. **Initscr( )** should only be called once per application.

endwin( ) A program should always call **endwin( )** before exiting or escaping from **curses** mode temporarily, to do a shell escape or *system(3S)* call, for example. This routine will restore *tty(7)* modes, move the cursor to the lower left corner of the screen and reset the terminal into the proper non-visual mode. To resume after a temporary escape, call **wrefresh( )** or **doupdate( )**.

isendwin( ) Returns **TRUE** if **endwin( )** has been called without any subsequent calls to **wrefresh( )**.

SCREEN *newterm(type, outfd, infd)

A program that outputs to more than one terminal must use **newterm( )** for each terminal instead of **initscr( )**. A program that wants an indication of error conditions, so that it may continue to run in a line-oriented mode if the terminal cannot support a screen-oriented program, must also use this routine. **newterm( )** should be called once for each terminal. It returns a variable of type **SCREEN*** that should be saved as a reference to that terminal. The arguments are the *type* of the terminal to be used in place of the environment variable **TERM**; *outfd*, a *stdio*(3S) file pointer for output to the terminal; and *infd*, another file pointer for input from the terminal. When it is done running, the program must also call **endwin( )** for each terminal being used. If **newterm( )** is called more than once for the same terminal, the first terminal referred to must be the last one for which **endwin( )** is called.

SCREEN *set_term(new)

This routine is used to switch between different terminals. The screen reference *new* becomes the new current terminal. A pointer to the screen of the previous terminal is returned by the routine. This is the only routine which manipulates **SCREEN** pointers; all other routines affect only the current terminal.

Window and Pad Manipulation

refresh( )

wrefresh (win) These routines (or **prefresh( )**, **pnoutrefresh( )**, **wnoutrefresh( )**, or **doupdate( )**) must be called to write output to the terminal, as most other routines merely manipulate data structures. **wrefresh( )** copies the named window to the physical terminal screen, taking into account what is already there in order to minimize the amount of information that's sent to the terminal (called optimization). **refresh( )** does the same

thing, except it uses **stdscr** as a default window. Unless **leaveok**( ) has been enabled, the physical cursor of the terminal is left at the location of the window's cursor. The number of characters output to the terminal is returned.

Note that **refresh**( ) is a macro.

wnoutrefresh(win)
doupdate( )
These two routines allow multiple updates to the physical terminal screen with more efficiency than **wrefresh**( ) alone. How this is accomplished is described in the next paragraph.

curses keeps two data structures representing the terminal screen: a *physical* terminal screen, describing what is actually on the screen, and a *virtual* terminal screen, describing what the programmer wants to have on the screen. **wrefresh**( ) works by first calling **wnoutrefresh**( ), which copys the named window to the virtual screen, and then by calling **doupdate**( ), which compares the virtual screen to the physical screen and does the actual update. If the programmer wishes to output several windows at once, a series of calls to **wrefresh**( ) will result in alternating calls to **wnoutrefresh**( ) and **doupdate**( ), causing several bursts of output to the screen. By first calling **wnoutrefresh**( ) for each window, it is then possible to call **doupdate**( ) once, resulting in only one burst of output, with probably fewer total characters transmitted and certainly less processor time used.

WINDOW *newwin(nlines, ncols, begin_y, begin_x)
Create and return a pointer to a new window with the given number of lines (or rows), *nlines*, and columns, *ncols*. The upper left corner of the window is at line *begin_y*, column *begin_x*. If either *nlines* or *ncols* is **0**, they will be set to the value of **lines**−*begin_y* and **cols**−*begin_x*. A new full-screen window is created by calling **newwin(0,0,0,0)**.

mvwin(win, y, x)
Move the window so that the upper left corner will be at position (*y*, *x*). If the move would cause the window to be off the screen, it is an error and the window is not moved.

WINDOW *subwin(orig, nlines, ncols, begin_y, begin_x)
Create and return a pointer to a new window with the given number of lines (or rows), *nlines*, and columns, *ncols*. The window is at position (*begin_y, begin_x*) on the screen. (This position is relative to the screen, and not to the window *orig*.) The window is made in the middle of the window *orig*, so that changes made to one window will affect both windows. When using this routine, often it will be necessary to call **touchwin**( ) or **touchline**( ) on *orig* before calling **wrefresh**( ).

delwin(win)
Delete the named window, freeing up all memory associated with it. In the case of overlapping windows, subwindows should be deleted before the main window.

WINDOW *newpad(nlines, ncols)
Create and return a pointer to a new pad data structure with the given number of lines (or rows), *nlines*, and columns, *ncols*. A pad is a window that is not restricted by the screen size and is not necessarily associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window will be on the screen at one time. Automatic refreshes of pads (e.g. from scrolling or echoing

of input) do not occur. It is not legal to call **wrefresh**( ) with a pad as an argument; the routines **prefresh**( ) or **pnoutrefresh**( ) should be called instead. Note that these routines require additional parameters to specify the part of the pad to be displayed and the location on the screen to be used for display.

WINDOW *subpad(orig, nlines, ncols, begin_y, begin_x)

Create and return a pointer to a subwindow within a pad with the given number of lines (or rows), *nlines*, and columns, *ncols*. Unlike **subwin**( ), which uses screen coordinates, the window is at position (*begin_y, begin_x*) on the pad. The window is made in the middle of the window *orig*, so that changes made to one window will affect both windows. When using this routine, often it will be necessary to call **touchwin**( ) or **touchline**( ) on *orig* before calling **prefresh**( ).

prefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)
pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)

These routines are analogous to **wrefresh**( ) and **wnoutrefresh**( ) except that pads, instead of windows, are involved. The additional parameters are needed to indicate what part of the pad and screen are involved. *pminrow* and *pmincol* specify the upper left corner, in the pad, of the rectangle to be displayed. *sminrow*, *smincol*, *smaxrow*, and *smaxcol* specify the edges, on the screen, of the rectangle to be displayed in. The lower right corner in the pad of the rectangle to be displayed is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures. Negative values of *pminrow*, *pmincol*, *sminrow*, or *smincol* are treated as if they were zero.

Output

These routines are used to "draw" text on windows.

addch(ch)
waddch(win, ch)
mvaddch(y, x, ch)
mvwaddch(win, y, x, ch)

The character *ch* is put into the window at the current cursor position of the window and the position of the window cursor is advanced. Its function is similar to that of *putchar* (see *putc(3S)*). At the right margin, an automatic newline is performed. At the bottom of the scrolling region, if **scrollok**( ) is enabled, the scrolling region will be scrolled up one line.

If *ch* is a tab, newline, or backspace, the cursor will be moved appropriately within the window. A newline also does a **clrtoeol**( ) before moving. Tabs are considered to be at every eighth column. If *ch* is another control character, it will be drawn in the ^X notation. (Calling **winch**( ) after adding a control character will not return the control character, but instead will return the representation of the control character.)

Video attributes can be combined with a character by or-ing them into the parameter. This will result in these attributes also being set. (The intent here is that text, including attributes, can be copied from one place to another using **inch**( ) and **addch**( ).) See **standout**( ), below.

Note that *ch* is actually of type **chtype**, not a character.

Note that **addch**( ), **mvaddch**( ), and **mvwaddch**( ) are macros.

echochar(ch)
wechochar(win, ch)
pechochar(pad, ch)

These routines are functionally equivalent to a call to **addch**(ch) followed by a call to **refresh**( ), a call to **waddch**(win, ch) followed by a call to **wrefresh**(win), or a call to **waddch**(pad, ch) followed by a call to **prefresh**(pad). The knowledge that only a single character is being output is taken into consideration and, for non-control characters, a considerable performance gain can be seen by using these routines instead of their equivalents. In the case of **pechochar**( ), the last location of the pad on the screen is reused for the arguments to **prefresh**( ).

Note that *ch* is actually of type **chtype**, not a character.

Note that **echochar**( ) is a macro.

addstr(str)
waddstr(win, str)
mvwaddstr(win, y, x, str)
mvaddstr(y, x, str)

These routines write all the characters of the null-terminated character string *str* on the given window. This is equivalent to calling **waddch**( ) once for each character in the string.

Note that **addstr**( ), **mvaddstr**( ), and **mvwaddstr**( ) are macros.

attroff(attrs)
wattroff(win, attrs)
attron(attrs)
wattron(win, attrs)
attrset(attrs)
wattrset(win, attrs)
standend( )
wstandend(win)
standout( )
wstandout(win)

These routines manipulate the current attributes of the named window. These attributes can be any combination of **A_STANDOUT**, **A_REVERSE**, **A_BOLD**, **A_DIM**, **A_BLINK**, **A_UNDERLINE**, and **A_ALTCHARSET**. These constants are defined in <**curses.h**> and can be combined with the C logical OR (|) operator.

The current attributes of a window are applied to all characters that are written into the window with **waddch**( ). Attributes are a property of the character, and move with the character through any scrolling and insert/delete line/character operations. To the extent possible on the particular terminal, they will be displayed as the graphic rendition of the characters put on the screen.

attrset(attrs) sets the current attributes of the given window to *attrs*. **attroff**(attrs) turns off the named attributes without turning on or off any other attributes. **attron**(attrs) turns on the named attributes without affecting any others. **standout**( ) is the same as **attron**(**A_STANDOUT**). **standend**( ) is the same as **attrset (0)**, that is, it turns off all attributes.

Note that *attrs* is actually of type **chtype**, not a character.

Note that **attroff**( ), **attron**( ), **attrset**( ), **standend**( ), and **standout**( ) are

macros.

beep( )
flash( )
These routines are used to signal the terminal user. **beep( )** will sound the audible alarm on the terminal, if possible, and if not, will flash the screen (visible bell), if that is possible. **flash( )** will flash the screen, and if that is not possible, will sound the audible signal. If neither signal is possible, nothing will happen. Nearly all terminals have an audible signal (bell or beep) but only some can flash the screen.

box(win, vertch, horch)
A box is drawn around the edge of the window, *win*. *vertch* and *horch* are the characters the box is to be drawn with. If *vertch* and *horch* are **0**, then appropriate default characters, **ACS_VLINE** and **ACS_HLINE**, will be used.

Note that *vertch* and *horch* are actually of type **chtype**, not characters.

erase( )
werase(win)
These routines copy blanks to every position in the window.

Note that **erase( )** is a macro.

clear( )
wclear(win)
These routines are like **erase( )** and **werase( )**, but they also call **clearok( )**, arranging that the screen will be cleared completely on the next call to **wrefresh( )** for that window, and repainted from scratch.

Note that **clear( )** is a macro.

clrtobot( )
wclrtobot(win)
All lines below the cursor in this window are erased. Also, the current line to the right of the cursor, inclusive, is erased.

Note that **clrtobot( )** is a macro.

clrtoeol( )
wclrtoeol(win)
The current line to the right of the cursor, inclusive, is erased.

Note that **clrtoeol( )** is a macro.

delay_output(ms)
Insert a *ms* millisecond pause in the output. It is not recommended that this routine be used extensively, because padding characters are used rather than a processor pause.

delch( )
wdelch(win)
mvdelch(y, x)
mvwdelch(win, y, x)
The character under the cursor in the window is deleted. All characters to the right on the same line are moved to the left one position and the last character on the line is filled with a blank. The cursor position does not change (after moving to (*y*, *x*), if specified). (This does not imply use of the hardware "delete-character" feature.)

Note that **delch( )**, **mvdelch( )**, and **mvwdelch( )** are macros.

deleteln( )
wdeleteln(win)
The line under the cursor in the window is deleted. All lines below the current line are moved up one line. The bottom line of the window is cleared. The cursor position does not change. (This does not imply use

of the hardware "delete-line" feature.)

Note that **deleteln**( ) is a macro.

getyx(win, y, x)
: The cursor position of the window is placed in the two integer variables *y* and *x*. This is implemented as a macro, so no "**&**" is necessary before the variables.

getbegyx(win, y, x)
getmaxyx(win, y, x)
: Like **getyx**( ), these routines store the current beginning coordinates and size of the specified window.

Note that **getbegyx**( ) and **getmaxyx**( ) are macros.

insch(ch)
winsch(win, ch)
mvwinsch(win, y, x, ch)
mvinsch(y, x, ch)
: The character *ch* is inserted before the character under the cursor. All characters to the right are moved one space to the right, possibly losing the rightmost character of the line. The cursor position does not change (after moving to (*y*, *x*), if specified). (This does not imply use of the hardware "insert-character" feature.)

Note that *ch* is actually of type **chtype**, not a character.

Note that **insch**( ), **mvinsch**( ), and **mvwinsch**( ) are macros.

insertln( )
winsertln(win)
: A blank line is inserted above the current line and the bottom line is lost. (This does not imply use of the hardware "insert-line" feature.)

Note that **insertln**( ) is a macro.

move(y, x)
wmove(win, y, x)
: The cursor associated with the window is moved to line (row) *y*, column *x*. This does not move the physical cursor of the terminal until **refresh**( ) is called. The position specified is relative to the upper left corner of the window, which is (**0**, **0**).

Note that **move**( ) is a macro.

overlay(srcwin, dstwin)
overwrite(srcwin, dstwin)
: These routines overlay *srcwin* on top of *dstwin*; that is, all text in *srcwin* is copied into *dstwin*. *srcwin* and *dstwin* need not be the same size; only text where the two windows overlap is copied. The difference is that **overlay**( ) is non-destructive (blanks are not copied), while **overwrite**( ) is destructive.

copywin(srcwin, dstwin, sminrow, smincol, dminrow, dmincol, dmaxrow, dmaxcol, overlay)
: This routine provides a finer grain of control over the **overlay**( ) and **overwrite**( ) routines. Like in the **prefresh**( ) routine, a rectangle is specified in the destination window, (*dminrow*, *dmincol*) and (*dmaxrow*, *dmaxcol*), and the upper-left-corner coordinates of the source window, (*sminrow*, *smincol*). If the argument *overlay* is true, then copying is non-destructive, as in **overlay**( ).

printw(fmt [, arg . . .])
wprintw(win, fmt [, arg . . .])

mvprintw(y, x, fmt [, arg . . .])
mvwprintw(win, y, x, fmt [, arg . . .])
These routines are analogous to *printf(3S)*. The string which would be output by *printf(3S)* is instead output using **waddstr**( ) on the given window.

vwprintw(win, fmt, varglist)
This routine corresponds to *vfprintf(3S)*. It performs a **wprintw**( ) using a variable argument list. The third argument is a *va_list*, a pointer to a list of arguments, as defined in <**varargs.h**>. See the *vprintf(3S)* and *varargs(5)* manual pages for a detailed description on how to use variable argument lists.

scroll(win) The window is scrolled up one line. This involves moving the lines in the window data structure. As an optimization, if the window is **stdscr** and the scrolling region is the entire window, the physical screen will be scrolled at the same time.

touchwin(win)
touchline(win, start, count)
Throw away all optimization information about which parts of the window have been touched, by pretending that the entire window has been drawn on. This is sometimes necessary when using overlapping windows, since a change to one window will affect the other window, but the records of which lines have been changed in the other window will not reflect the change. **touchline**( ) only pretends that *count* lines have been changed, beginning with line *start*.

Input
getch( )
wgetch(win)
mvgetch(y, x)
mvwgetch(win, y, x) A character is read from the terminal associated with the window. In NODELAY mode, if there is no input waiting, the value **ERR** is returned. In DELAY mode, the program will hang until the system passes text through to the program. Depending on the setting of **cbreak**( ), this will be after one character (CBREAK mode), or after the first newline (NOCBREAK mode). In HALF-DELAY mode, the program will hang until a character is typed or the specified timeout has been reached. Unless **noecho**( ) has been set, the character will also be echoed into the designated window. No **refresh**( ) will occur between the **move**( ) and the **getch**( ) done within the routines **mvgetch**( ) and **mvwgetch**( ).

When using **getch**( ), **wgetch**( ), **mvgetch**( ), or **mvwgetch**( ), do not set both NOCBREAK mode (**nocbreak**( )) and ECHO mode (**echo**( )) at the same time. Depending on the state of the *tty*(7) driver when each character is typed, the program may produce undesirable results.

If **keypad**(win, **TRUE**) has been called, and a function key is pressed, the token for that function key will be returned instead of the raw characters. (See **keypad**( ) under "Input Options Setting.") Possible function keys are defined in <**curses.h**> with integers beginning with **0401**, whose names begin with **KEY_**. If a character is received that could be the beginning of a function key (such as escape), **curses** will set a timer. If the remainder of the sequence is not received within the designated time, the character will be passed through, otherwise the function key

value will be returned. For this reason, on many terminals, there will be a delay after a user presses the escape key before the escape is returned to the program. (Use by a programmer of the escape key for a single character routine is discouraged. Also see **notimeout**( ) below.)

Note that **getch**( ), **mvgetch**( ), and **mvwgetch**( ) are macros.

getstr(str)
wgetstr(win, str)
mvgetstr(y, x, str)
mvwgetstr(win, y, x, str)

A series of calls to **getch**( ) is made, until a newline, carriage return, or enter key is received. The resulting value is placed in the area pointed at by the character pointer *str*. The user's erase and kill characters are interpreted. As in **mvgetch**( ), no **refresh**( ) is done between the **move**( ) and **getstr**( ) within the routines **mvgetstr**( ) and **mvwgetstr**( ).

Note that **getstr**( ), **mvgetstr**( ), and **mvwgetstr**( ) are macros.

flushinp( ) Throws away any typeahead that has been typed by the user and has not yet been read by the program.

ungetch(c) Place *c* back onto the input queue to be returned by the next call to **wgetch**( ).

inch( )
winch(win)
mvinch(y, x)
mvwinch(win, y, x)

The character, of type **chtype**, at the current position in the named window is returned. If any attributes are set for that position, their values will be OR'ed into the value returned. The predefined constants **A_CHARTEXT** and **A_ATTRIBUTES**, defined in <**curses.h**>, can be used with the C logical AND (&) operator to extract the character or attributes alone.

Note that **inch**( ), **winch**( ), **mvinch**( ), and **mvwinch**( ) are macros.

scanw(fmt [, arg . . .])
wscanw(win, fmt [, arg . . .])
mvscanw(y, x, fmt [, arg . . .])
mvwscanw(win, y, x, fmt [, arg . . .])

These routines correspond to *scanf(3S)*, as do their arguments and return values. **wgetstr**( ) is called on the window, and the resulting line is used as input for the scan.

vwscanw(win, fmt, ap)

This routine is similar to **vwprintw**( ) above in that performs a **wscanw**( ) using a variable argument list. The third argument is a *va_list*, a pointer to a list of arguments, as defined in <**varargs.h**>. See the *vprintf(3S)* and *varargs(5)* manual pages for a detailed description on how to use variable argument lists.

Output Options Setting

These routines set options within **curses** that deal with output. All options are initially FALSE, unless otherwise stated. It is not necessary to turn these options off before calling **endwin**( ).

clearok(win, bf) If enabled (*bf* is **TRUE**), the next call to **wrefresh**( ) with this window will clear the screen completely and redraw the entire screen from scratch.

This is useful when the contents of the screen are uncertain, or in some cases for a more pleasing visual effect.

idlok(win, bf)
If enabled (*bf* is **TRUE**), **curses** will consider using the hardware "insert/delete-line" feature of terminals so equipped. If disabled (*bf* is FALSE), **curses** will very seldom use this feature. (The "insert/delete-character" feature is always considered.) This option should be enabled only if your application needs "insert/delete-line", for example, for a screen editor. It is disabled by default because "insert/delete-line" tends to be visually annoying when used in applications where it isn't really needed. If "insert/delete-line" cannot be used, **curses** will redraw the changed portions of all lines.

leaveok(win, bf)
Normally, the hardware cursor is left at the location of the window cursor being refreshed. This option allows the cursor to be left wherever the update happens to leave it. It is useful for applications where the cursor is not used, since it reduces the need for cursor motions. If possible, the cursor is made invisible when this option is enabled.

setscrreg(top, bot)
wsetscrreg(win, top, bot)
These routines allow the user to set a software scrolling region in a window. *top* and *bot* are the line numbers of the top and bottom margin of the scrolling region. (Line 0 is the top line of the window.) If this option and **scrollok**( ) are enabled, an attempt to move off the bottom margin line will cause all lines in the scrolling region to scroll up one line. (Note that this has nothing to do with use of a physical scrolling region capability in the terminal, like that in the DEC VT100. Only the text of the window is scrolled; if **idlok**( ) is enabled and the terminal has either a scrolling region or "insert/delete-line" capability, they will probably be used by the output routines.)

Note that **setscrreg**( ) and **wsetscrreg**( ) are macros.

scrollok(win, bf)
This option controls what happens when the cursor of a window is moved off the edge of the window or scrolling region, either from a newline on the bottom line, or typing the last character of the last line. If disabled (*bf* is **FALSE**), the cursor is left on the bottom line at the location where the offending character was entered. If enabled (*bf* is **TRUE**), **wrefresh**( ) is called on the window, and then the physical terminal and window are scrolled up one line. (Note that in order to get the physical scrolling effect on the terminal, it is also necessary to call **idlok**( ).)

nl( )
nonl( )
These routines control whether newline is translated into carriage return and linefeed on output, and whether return is translated into newline on input. Initially, the translations do occur. By disabling these translations using **nonl**( ), **curses** is able to make better use of the linefeed capability, resulting in faster cursor motion.

Input Options Setting
These routines set options within **curses** that deal with input. The options involve using *ioctl(2)* and therefore interact with **curses** routines. It is not necessary to turn these options off before calling **endwin**( ).

For more information on these options, see Chapter 10 of the *Programmer's Guide*.

cbreak( )

nocbreak( )

These two routines put the terminal into and out of CBREAK mode, respectively. In CBREAK mode, characters typed by the user are immediately available to the program and erase/kill character processing is not performed. When in NOCBREAK mode, the tty driver will buffer characters typed until a newline or carriage return is typed. Interrupt and flow-control characters are unaffected by this mode (see *termio(7)*). Initially the terminal may or may not be in CBREAK mode, as it is inherited, therefore, a program should call **cbreak**( ) or **nocbreak**( ) explicitly. Most interactive programs using **curses** will set CBREAK mode.

Note that **cbreak**( ) overrides **raw**( ). See **getch**( ) under "Input" for a discussion of how these routines interact with **echo**( ) and **noecho**( ).

echo( )
noecho( )

These routines control whether characters typed by the user are echoed by **getch**( ) as they are typed. Echoing by the tty driver is always disabled, but initially **getch**( ) is in ECHO mode, so characters typed are echoed. Authors of most interactive programs prefer to do their own echoing in a controlled area of the screen, or not to echo at all, so they disable echoing by calling **noecho**( ). See **getch**( ) under "Input" for a discussion of how these routines interact with **cbreak**( ) and **nocbreak**( ).

halfdelay(tenths)

Half-delay mode is similar to CBREAK mode in that characters typed by the user are immediately available to the program. However, after blocking for *tenths* tenths of seconds, ERR will be returned if nothing has been typed. *tenths* must be a number between 1 and 255. Use **nocbreak**( ) to leave half-delay mode.

intrflush(win, bf)

If this option is enabled, when an interrupt key is pressed on the keyboard (interrupt, break, quit) all output in the tty driver queue will be flushed, giving the effect of faster response to the interrupt, but causing **curses** to have the wrong idea of what is on the screen. Disabling the option prevents the flush. The default for the option is inherited from the tty driver settings. The window argument is ignored.

keypad(win, bf)

This option enables the keypad of the user's terminal. If enabled, the user can press a function key (such as an arrow key) and **wgetch**( ) will return a single value representing the function key, as in **KEY_LEFT**. If disabled, **curses** will not treat function keys specially and the program would have to interpret the escape sequences itself. If the keypad in the terminal can be turned on (made to transmit) and off (made to work locally), turning on this option will cause the terminal keypad to be turned on when **wgetch**( ) is called.

meta(win, bf)

If enabled, characters returned by **wgetch**( ) are transmitted with all 8 bits, instead of with the highest bit stripped. In order for **meta**( ) to work correctly, the **km** (has_meta_key) capability has to be specified in the terminal's *terminfo(4)* entry.

nodelay(win, bf)

This option causes **wgetch**( ) to be a non-blocking call. If no input is ready, **wgetch**( ) will return ERR. If disabled, **wgetch**( ) will hang until a key is pressed.

notimeout(win, bf)

While interpreting an input escape sequence, **wgetch**( ) will set a timer while waiting for the next character. If **notimeout**(win, **TRUE**) is called, then **wgetch**( ) will not set a timer. The purpose of the timeout is to

differentiate between sequences received from a function key and those typed by a user.

raw( )
noraw( ) The terminal is placed into or out of raw mode. RAW mode is similar to CBREAK mode, in that characters typed are immediately passed through to the user program. The differences are that in RAW mode, the interrupt, quit, suspend, and flow control characters are passed through uninterpreted, instead of generating a signal. RAW mode also causes 8-bit input and output. The behavior of the BREAK key depends on other bits in the *tty(7)* driver that are not set by **curses**.

typeahead(fildes) **curses** does "line-breakout optimization" by looking for typeahead periodically while updating the screen. If input is found, and it is coming from a tty, the current update will be postponed until **refresh( )** or **doupdate( )** is called again. This allows faster response to commands typed in advance. Normally, the file descriptor for the input FILE pointer passed to **newterm( )**, or **stdin** in the case that **initscr( )** was used, will be used to do this typeahead checking. The **typeahead( )** routine specifies that the file descriptor *fildes* is to be used to check for typeahead instead. If *fildes* is **−1**, then no typeahead checking will be done.

Note that *fildes* is a file descriptor, not a **<stdio.h>** FILE pointer.

Environment Queries

baudrate( ) Returns the output speed of the terminal. The number returned is in bits per second, for example, 9600, and is an integer.

char erasechar( ) The user's current erase character is returned.

has_ic( ) True if the terminal has insert- and delete-character capabilities.

has_il( ) True if the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions. This might be used to check to see if it would be appropriate to turn on physical scrolling using **scrollok( )**.

char killchar( ) The user's current line-kill character is returned.

char *longname( ) This routine returns a pointer to a static area containing a verbose description of the current terminal. The maximum length of a verbose description is 128 characters. It is defined only after the call to **initscr( )** or **newterm( )**. The area is overwritten by each call to **newterm( )** and is not restored by **set_term( )**, so the value should be saved between calls to **newterm( )** if **longname( )** is going to be used with multiple terminals.

Soft Labels

If desired, **curses** will manipulate the set of soft function-key labels that exist on many terminals. For those terminals that do not have soft labels, if you want to simulate them, **curses** will take over the bottom line of **stdscr**, reducing the size of **stdscr** and the variable **LINES**. **Curses** standardizes on 8 labels of 8 characters each.

slk_init(labfmt) In order to use soft labels, this routine must be called before **initscr( )** or **newterm( )** is called. If **initscr( )** winds up using a line from **stdscr** to emulate the soft labels, then *labfmt* determines how the labels are arranged on the screen. Setting *labfmt* to **0** indicates that the labels are to be arranged in a 3-2-3 arrangement; **1** asks for a 4-4 arrangement.

slk_set(labnum, label, labfmt)

Labnum is the label number, from 1 to 8. *Label* is the string to be put on the label, up to 8 characters in length. A NULL string or a NULL pointer will put up a blank label. *Labfmt* is one of **0**, **1** or **2**, to indicate whether the label is to be left-justified, centered, or right-justified within the label.

slk_refresh( )
slk_noutrefresh( ) These routines correspond to the routines **wrefresh( )** and **wnoutrefresh( )**. Most applications would use **slk_noutrefresh( )** because a **wrefresh( )** will most likely soon follow.

char *slk_label(labnum)
The current label for label number *labnum*, with leading and trailing blanks stripped, is returned.

slk_clear( ) The soft labels are cleared from the screen.

slk_restore( ) The soft labels are restored to the screen after a **slk_clear( )**.

slk_touch( ) All of the soft labels are forced to be output the next time a **slk_noutrefresh( )** is performed.

Low-Level Curses Access

The following routines give low-level access to various **curses** functionality. These routines typically would be used inside of library routines.

def_prog_mode( )
def_shell_mode( ) Save the current terminal modes as the "program" (in **curses**) or "shell" (not in **curses**) state for use by the **reset_prog_mode( )** and **reset_shell_mode( )** routines. This is done automatically by **initscr( )**.

reset_prog_mode( )
reset_shell_mode( ) Restore the terminal to "program" (in **curses**) or "shell" (out of **curses**) state. These are done automatically by **endwin( )** and **doupdate( )** after an **endwin( )**, so they normally would not be called.

resetty( )
savetty( ) These routines save and restore the state of the terminal modes. **savetty( )** saves the current state of the terminal in a buffer and **resetty( )** restores the state to what it was at the last call to **savetty( )**.

getsyx(y, x) The current coordinates of the virtual screen cursor are returned in *y* and *x*. Like **getyx( )**, the variables *y* and *x* do not take an "**&**" before them. If **leaveok( )** is currently **TRUE**, then **−1,−1** will be returned. If lines may have been removed from the top of the screen using **ripoffline( )** and the values are to be used beyond just passing them on to **setsyx( )**, the value ***y*+stdscr−>_yoffset** should be used for those other uses.

Note that **getsyx( )** is a macro.

setsyx(y, x) The virtual screen cursor is set to *y*, *x*. If *y* and *x* are both **−1**, then **leaveok( )** will be set. The two routines **getsyx( )** and **setsyx( )** are designed to be used by a library routine which manipulates curses windows but does not want to mess up the current position of the program's cursor. The library routine would call **getsyx( )** at the beginning, do its manipulation of its own windows, do a **wnoutrefresh( )** on its windows, call **setsyx( )**, and then call **doupdate( )**.

ripoffline(line, init) This routine provides access to the same facility that **slk_init( )** uses to

reduce the size of the screen. **Ripoffline**( ) must be called before **initscr**( ) or **newterm**( ) is called. If *line* is positive, a line will be removed from the top of **stdscr**; if negative, a line will be removed from the bottom. When this is done inside **initscr**( ), the routine *init*( ) is called with two arguments: a window pointer to the 1-line window that has been allocated and an integer with the number of columns in the window. Inside this initialization routine, the integer variables **LINES** and **COLS** (defined in **<curses.h>**) are not guaranteed to be accurate and **wrefresh**( ) or **doupdate**( ) must not be called. It is allowable to call **wnoutrefresh**( ) during the initialization routine.

ripoffline( ) can be called up to five times before calling **initscr**( ) or **newterm**( ).

scr_dump(filename) The current contents of the virtual screen are written to the file *filename*.

scr_restore(filename)
The virtual screen is set to the contents of *filename*, which must have been written using **scr_dump**( ). The next call to **doupdate**( ) will restore the screen to what it looked like in the dump file.

scr_init(filename) The contents of *filename* are read in and used to initialize the **curses** data structures about what the terminal currently has on its screen. If the data is determined to be valid, **curses** will base its next update of the screen on this information rather than clearing the screen and starting from scratch. **scr_init**( ) would be used after **initscr**( ) or a *system(3S)* call to share the screen with another process which has done a **scr_dump**( ) after its **endwin**( ) call. The data will be declared invalid if the time-stamp of the tty is old or the *terminfo(4)* capability **nrrmc** is true.

curs_set(visibility) The cursor is set to invisible, normal, or very visible for *visibility* equal to **0**, **1** or **2**.

draino(ms) Wait until the output has drained enough that it will only take *ms* more milliseconds to drain completely.

garbagedlines(win, begline, numlines)
This routine indicates to **curses** that a screen line is garbaged and should be thrown away before having anything written over the top of it. It could be used for programs such as editors which want a command to redraw just a single line. Such a command could be used in cases where there is a noisy communications line and redrawing the entire screen would be subject to even more communication noise. Just redrawing the single line gives some semblance of hope that it would show up unblemished. The current location of the window is used to determine which lines are to be redrawn.

napms(ms) Sleep for *ms* milliseconds.

Terminfo-Level Manipulations

These low-level routines must be called by programs that need to deal directly with the *terminfo(4)* database to handle certain terminal capabilities, such as programming function keys. For all other functionality, **curses** routines are more suitable and their use is recommended. Initially, **setupterm**( ) should be called. (Note that **setupterm**( ) is automatically called by **initscr**( ) and **newterm**( ).) This will define the set of terminal-dependent variables defined in the *terminfo(4)* database. The *terminfo(4)* variables **lines** and **columns** (see *terminfo(4)*) are

initialized by **setupterm**( ) as follows: if the environment variables **LINES** and **COLUMNS** exist, their values are used. If the above environment variables do not exist and the program is running in a layer, the size of the current layer is used. Otherwise, the values for **lines** and **columns** specified in the *terminfo(4)* database are used. The header files **<curses.h>** and **<term.h>** should be included, in this order, 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, **reset_shell_mode**( ) should be called to restore the tty modes. Programs which use cursor addressing should output **enter_ca_mode** upon startup and should output **exit_ca_mode** before exiting (see *terminfo(4)*). (Programs desiring shell escapes should call **reset_shell_mode**( ) and output **exit_ca_mode** before the shell is called and should output **enter_ca_mode** and call **reset_prog_mode**( ) after returning from the shell. Note that this is different from the **curses** routines (see **endwin**( )).

setupterm(term, fildes, errret)

Reads in the *terminfo(4)* database, initializing the *terminfo(4)* structures, but does not set up the output virtualization structures used by **curses**. The terminal type is in the character string *term*; if *term* is **NULL**, the environment variable **TERM** will be used. All output is to the file descriptor *fildes*. If *errret* is not **NULL**, then **setupterm**( ) will return **OK** or **ERR** and store a status value in the integer pointed to by *errret*. A status of **1** in *errret* is normal, **0** means that the terminal could not be found, and **−1** means that the *terminfo(4)* database could not be found. If *errret* is **NULL**, **setupterm**( ) will print an error message upon finding an error and exit. Thus, the simplest call is **setupterm ((char *)0, 1, (int *)0)**, which uses all the defaults.

The *terminfo(4)* boolean, numeric and string variables are stored in a structure of type **TERMINAL**. After **setupterm**( ) returns successfully, the variable **cur_term** (of type **TERMINAL ***) is initialized with all of the information that the *terminfo(4)* boolean, numeric and string variables refer to. The pointer may be saved before calling **setupterm**( ) again. Further calls to **setupterm**( ) will allocate new space rather than reuse the space pointed to by **cur_term**.

set_curterm(nterm)

Nterm is of type **TERMINAL ***. **Set_curterm**( ) sets the variable **cur_term** to *nterm*, and makes all of the *terminfo(4)* boolean, numeric and string variables use the values from *nterm*.

del_curterm(oterm)

Oterm is of type **TERMINAL ***. **Del_curterm**( ) frees the space pointed to by *oterm* and makes it available for further use. If *oterm* is the same as **cur_term**, then references to any of the *terminfo(4)* boolean, numeric and string variables thereafter may refer to invalid memory locations until another **setupterm**( ) has been called.

restartterm(term, fildes, errret)

Like **setupterm**( ) after a memory restore.

char *tparm(str, p_1, p_2, ..., p_9)

Instantiate the string *str* with parms p_i. A pointer is returned to the result of *str* with the parameters applied.

tputs(str, count, putc)

Apply padding to the string *str* and output it. *Str* must be a *terminfo(4)* string variable or the return value from **tparm**( ), **tgetstr**( ), **tigetstr**( ) or **tgoto**( ). *Count* is the number of lines affected, or **1** if not applicable. *Putc(3S)* is a *putchar*-like routine to which the characters are passed, one

at a time.

putp(str) A routine that calls **tputs** (*str*, **1**, **putchar**( )).

vidputs(attrs, putc) Output a string that puts the terminal in the video attribute mode *attrs*, which is any combination of the attributes listed below. The characters are passed to the *putchar*-like routine *putc(3S)*.

vidattr(attrs) Like **vidputs**( ), except that it outputs through *putchar* (see *putc(3S)*).

mvcur(oldrow, oldcol, newrow, newcol)
Low-level cursor motion.

The following routines return the value of the capability corresponding to the *terminfo(4) capname* passed to them, such as **xenl**.

tigetflag(capname) The value **−1** is returned if *capname* is not a boolean capability.

tigetnum(capname) The value **−2** is returned if *capname* is not a numeric capability.

tigetstr(capname) The value (char *) **−1** is returned if *capname* is not a string capability.

char *boolnames[ ], ***boolcodes**[ ], ***boolfnames**[ ]
char *numnames[ ], ***numcodes**[ ], ***numfnames**[ ]
char *strnames[ ], ***strcodes**[ ], ***strfnames**[ ]
These null-terminated arrays contain the *capnames*, the *termcap* codes, and the full C names, for each of the *terminfo(4)* variables.

Termcap Emulation

These routines are included as a conversion aid for programs that use the *termcap* library. Their parameters are the same and the routines are emulated using the *terminfo(4)* database.

tgetent(bp, name) Look up *termcap* entry for *name*. The emulation ignores the buffer pointer *bp*.

tgetflag(codename) Get the boolean entry for *codename*.

tgetnum(codes) Get numeric entry for *codename*.

char *tgetstr(codename, area)
Return the string entry for *codename*. If *area* is not **NULL**, then also store it in the buffer pointed to by *area* and advance *area*. **tputs**( ) should be used to output the returned string.

char *tgoto(cap, col, row)
Instantiate the parameters into the given capability. The output from this routine is to be passed to **tputs**( ).

tputs(str, affcnt, putc)
See **tputs**( ) above, under "Terminfo-Level Manipulations".

Miscellaneous

traceoff( )
traceon( ) Turn off and on debugging trace output when using the debug version of the **curses** library, */usr/lib/libdcurses.a*. This facility is available only to customers with a source license.

unctrl(c) This macro expands to a character string which is a printable representation of the character *c*. Control characters are displayed in the ^X notation. Printing characters are displayed as is.

Unctrl( ) is a macro, defined in <**unctrl.h**>, which is automatically included by <**curses.h**>.

char *keyname(c) A character string corresponding to the key *c* is returned.

filter( ) This routine is one of the few that is to be called before **initscr**( ) or **newterm**( ) is called. It arranges things so that **curses** thinks that there is a 1-line screen. **curses** will not use any terminal capabilities that assume that they know what line on the screen the cursor is on.

Use of curscr

The special window **curscr** can be used in only a few routines. If the window argument to **clearok**( ) is **curscr**, the next call to **wrefresh**( ) with any window will cause the screen to be cleared and repainted from scratch. If the window argument to **wrefresh**( ) is **curscr**, the screen is immediately cleared and repainted from scratch. (This is how most programs would implement a "repaint-screen" routine.) The source window argument to **overlay**( ), **overwrite**( ), and **copywin**( ) may be **curscr**, in which case the current contents of the virtual terminal screen will be accessed.

Obsolete Calls

Various routines are provided to maintain compatibility in programs written for older versions of the curses library. These routines are all emulated as indicated below.

crmode( ) Replaced by **cbreak**( ).

fixterm( ) Replaced by **reset_prog_mode**( ).

gettmode( ) A no-op.

nocrmode( ) Replaced by **nocbreak**( ).

resetterm( ) Replaced by **reset_shell_mode**( ).

saveterm( ) Replaced by **def_prog_mode**( ).

setterm( ) Replaced by **setupterm**( ).

ATTRIBUTES

The following video attributes, defined in **<curses.h>**, can be passed to the routines **attron**( ), **attroff**( ), and **attrset**( ), or OR'ed with the characters passed to **addch**( ).

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_ALTCHARSET	Alternate character set
A_CHARTEXT	Bit-mask to extract character (described under **winch**( ))
A_ATTRIBUTES	Bit-mask to extract attributes (described under **winch**( ))
A_NORMAL	Bit mask to reset all attributes off (for example: **attrset (A_NORMAL)**

FUNCTION-KEYS

The following function keys, defined in **<curses.h>**, might be returned by **getch**( ) if **keypad**( ) has been enabled. Note that not all of these may be supported on a particular terminal if the terminal does not transmit a unique code when the key is pressed or the definition for the key is not present in the *terminfo(4)* database.

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 for 64 keys is reserved.
KEY_F(n)	(KEY_F0+(n))	Formula for f_n.
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
KEY_SRESET	0530	soft (partial) reset
KEY_RESET	0531	reset or hard reset
KEY_PRINT	0532	print or copy
KEY_LL	0533	home down or bottom (lower left) keypad is arranged like this: A1 up A3 left B2 right C1 down C3
KEY_A1	0534	Upper left of keypad
KEY_A3	0535	Upper right of keypad
KEY_B2	0536	Center of keypad
KEY_C1	0537	Lower left of keypad
KEY_C3	0540	Lower right of keypad
KEY_BTAB	0541	Back tab key
KEY_BEG	0542	beg(inning) key
KEY_CANCEL	0543	cancel key
KEY_CLOSE	0544	close key
KEY_COMMAND	0545	cmd (command) key
KEY_COPY	0546	copy key
KEY_CREATE	0547	create key
KEY_END	0550	end key
KEY_EXIT	0551	exit key
KEY_FIND	0552	find key
KEY_HELP	0553	help key
KEY_MARK	0554	mark key

KEY_MESSAGE	0555	message key
KEY_MOVE	0556	move key
KEY_NEXT	0557	next object key
KEY_OPEN	0560	open key
KEY_OPTIONS	0561	options key
KEY_PREVIOUS	0562	previous object key
KEY_REDO	0563	redo key
KEY_REFERENCE	0564	ref(erence) key
KEY_REFRESH	0565	refresh key
KEY_REPLACE	0566	replace key
KEY_RESTART	0567	restart key
KEY_RESUME	0570	resume key
KEY_SAVE	0571	save key
KEY_SBEG	0572	shifted beginning key
KEY_SCANCEL	0573	shifted cancel key
KEY_SCOMMAND	0574	shifted command key
KEY_SCOPY	0575	shifted copy key
KEY_SCREATE	0576	shifted create key
KEY_SDC	0577	shifted delete char key
KEY_SDL	0600	shifted delete line key
KEY_SELECT	0601	select key
KEY_SEND	0602	shifted end key
KEY_SEOL	0603	shifted clear line key
KEY_SEXIT	0604	shifted exit key
KEY_SFIND	0605	shifted find key
KEY_SHELP	0606	shifted help key
KEY_SHOME	0607	shifted home key
KEY_SIC	0610	shifted input key
KEY_SLEFT	0611	shifted left arrow key
KEY_SMESSAGE	0612	shifted message key
KEY_SMOVE	0613	shifted move key
KEY_SNEXT	0614	shifted next key
KEY_SOPTIONS	0615	shifted options key
KEY_SPREVIOUS	0616	shifted prev key
KEY_SPRINT	0617	shifted print key
KEY_SREDO	0620	shifted redo key
KEY_SREPLACE	0621	shifted replace key
KEY_SRIGHT	0622	shifted right arrow
KEY_SRSUME	0623	shifted resume key
KEY_SSAVE	0624	shifted save key
KEY_SSUSPEND	0625	shifted suspend key
KEY_SUNDO	0626	shifted undo key
KEY_SUSPEND	0627	suspend key
KEY_UNDO	0630	undo key

LINE GRAPHICS

The following variables may be used to add line-drawing characters to the screen with **waddch**( ). When defined for the terminal, the variable will have the **A_ALTCHARSET** bit turned on. Otherwise, the default charcter listed below will be stored in the variable. The names were chosen to be consistent with the DEC VT100 nomenclature.

Name	*Default*	*Glyph Description*
ACS_ULCORNER	+	upper left corner
ACS_LLCORNER	+	lower left corner
ACS_URCORNER	+	upper right corner
ACS_LRCORNER	+	lower right corner
ACS_RTEE	+	right tee (┤)
ACS_LTEE	+	left tee (├)
ACS_BTEE	+	bottom tee (┴)
ACS_TTEE	+	top tee (┬)
ACS_HLINE	−	horizontal line
ACS_VLINE	|	vertical line
ACS_PLUS	+	plus
ACS_S1	−	scan line 1
ACS_S9	_	scan line 9
ACS_DIAMOND	+	diamond
ACS_CKBOARD	:	checker board (stipple)
ACS_DEGREE	'	degree symbol
ACS_PLMINUS	#	plus/minus
ACS_BULLET	o	bullet
ACS_LARROW	<	arrow pointing left
ACS_RARROW	>	arrow pointing right
ACS_DARROW	v	arrow pointing down
ACS_UARROW	^	arrow pointing up
ACS_BOARD	#	board of squares
ACS_LANTERN	#	lantern symbol
ACS_BLOCK	#	solid square block

RETURN VALUES

All routines return the integer **OK** upon successful completion and the integer **ERR** upon failure, unless otherwise noted in the preceding routine descriptions.

All macros return the value of their **w** version, except **setscrreg**(), **wsetscrreg**(), **getsyx**(), **getyx**(), **getbegy**(), **getmaxyx**(). For these macros, no useful value is returned. Routines that return pointers always return **(type *) NULL** on error.

ERRORS

Currently typeahead checking is done using a nodelay read followed by an **ungetch**() of any character that may have been read. Typeahead checking is done only if **wgetch**() has been called at least once. This will be changed when proper kernel support is available. Programs which use a mixture of their own input routines with **curses** input routines may wish to call **typeahead(−1)** to turn off typeahead checking. The argument to **napms**() is currently rounded up to the nearest second. **Draino**(ms) only works for *ms* equal to **0**.

WARNINGS

To use the new **curses** features, use the Release 3.0 version of **curses** on UNIX System Release 3.0. All programs that ran with System V Release 2 **curses** will run with System V Release 3.0. You may link applications with object files based on the Release 2 *curses/terminfo* with the Release 3.0 *libcurses.a* library. You may link applications with object files based on the Release 3.0 *curses/terminfo* with the Release 2 *libcurses.a* library, so long as the application does not use the new features in the Release 3.0 *curses/terminfo*.

Between the time a call to **initscr**() and **endwin**() has been issued, use only the routines in the **curses** library to generate output. Using system calls or the "standard I/O package" (see *stdio(3S)*) for output during that time can cause unpredictable results.

SEE ALSO

cc(1), ld(1) in the *User's Reference Manual.*
ioctl(2), printf(3S), putc(3S), scanf(3S), stdio(3S), system(3S), vprintf(3S), profile(4), term(4), terminfo(4), varargs(5).
termio(7), tty(7) in the *System Administrator's Reference Manual.*
Chapter 12, "**curses/terminfo**", in the *Programmer's Guide.*

NAME

cuserid – get character login name of the user

SYNOPSIS

#include <stdio.h>

char *cuserid (s)
char *s;

DESCRIPTION

cuserid generates a character-string representation of the login name that the owner of the current process is logged in under. 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)** will be placed at *s[0]*.

SEE ALSO

getlogin(3C), getpwent(3C).

NAME

dial – establish an out-going terminal line connection

SYNOPSIS

#include <dial.h>

int dial (call)
CALL call;

void undial (fd)
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;
```

The CALL element *speed* is 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 will transmit 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 secound only. The CALL element *baud* is for the 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). 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 *L-devices* file. In this case, the value of the *baud* element need not be specified as it will be determined from the *L-devices* file. The *telno* element is for a pointer to a character string representing the telephone number to be dialed. The termination symbol will be supplied by the *dial* function, and should not be included in the *telno* string passed to *dial* in the CALL structure. The CALL element *modem* is used to specify modem control for direct lines. This element should be non-zero if modem control is required. The CALL element *attr* is a pointer to a *termio* structure, as defined in the *termio.h* header file. A NULL value for this pointer element may be passed to the *dial* function, but if such a structure is included, the elements specified in it will be set for the outgoing terminal line before the connection is established. This is often important for certain attributes such as parity and baud-rate.

The CALL element *device* is used to hold the device name (cul..) that establishes the connection.

The CALL element *dev_len* is the length of the device name that is copied into the array device.

FILES

/usr/lib/uucp/L-devices
/usr/spool/uucp/LCK.. tty-device

SEE ALSO

alarm(2), read(2), write(2).
termio(7) in the *System Administrator's Reference Manual*.

DIAGNOSTICS

On failure, a negative value indicating the reason for the failure will be 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    /* 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

The *dial(3C)* library function is not compatible with Basic Networking Utilities on UNIX System V Release 2.0.

Including the <**dial.h**> header file automatically includes the <**termio.h**> header file.

The above routine uses <**stdio.h**>, which causes it to increase the size of programs, not otherwise using standard I/O, more than might be expected.

ERRORS

An *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(2)* or *write(2)* system call, causing an apparent error return. If the user program expects to be around for an hour or more, error returns from *read*s should be checked for (**errno==EINTR**), and the *read* possibly reissued.

NAME

dim, ddim, idim – positive difference intrinsic functions

SYNOPSIS

```
integer a1, a2, a3
a3 = idim(a1, a2)

real a1, a2, a3
a3 = dim(a1, a2)

double precision a1, a2, a3
a3 = ddim(a1, a2)
```

DESCRIPTION

These functions return:

```
a1−a2   if a1 > a2
0       if a1 <= a2
```

NAME

disassembler – disassemble a MIPS instruction and print the results

SYNOPSIS

```
int disassembler (iadr, regstyle, get_symname, get_regvalue, get_bytes, print_header)
unsigned        iadr;
int             regstyle;
char            *(*get_symname)();
int             (*get_regvalue)();
long            (*get_bytes)();
void            (*print_header)();
```

DESCRIPTION

Disassembler disassembles and prints a MIPS machine instruction on *stdout.*

Iadr is the instruction address to be disassembled. *Regstyle* specifies how registers are named in the disassembly; if the value is 0, compiler names are used; otherwise, hardware names are used.

The next four arguments are function pointers, most of which give the caller some flexibility in the appearance of the disassembly. The only function that **MUST** be provided is *get_bytes.* All other functions are optional. *Get_bytes* is called with no arguments and returns the next byte(s) to disassemble.

Get_symname is passed an address, which is the target of a *jal* instruction. If **NULL** is returned or if *get_symname* is **NULL**, the *disassembler* prints the address; otherwise, the string name is printed as returned from *get_symname.* If *get_regvalue* is not **NULL**, it is passed a register number and returns the current contents of the specified register. *Disassembler* prints this information along with the instruction disassembly. If *print_header* is not **NULL**, it is passed the instruction address *iadr* and the current instruction to be disassembled, which is the return value from *get_bytes. Print_header* can use these parameters to print any desired information before the actual instruction disassembly is printed.

If *get_bytes* is **NULL,** the *disassembler* returns -1 and errno is set to **EINVAL;** otherwise, the number of bytes that were disassembled is returned. If the disassembled word is a jump or branch instruction, the instruction in the delay slot is also disassembled.

The program must be loaded with the object file access routine library **libmld.a**.

SEE ALSO

ldfcn(4).

NAME

directory: opendir, readdir, telldir, seekdir, rewinddir, closedir – directory operations

SYNOPSIS

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir (filename)
char *filename;

struct dirent *readdir (dirp)
DIR *dirp;

long telldir (dirp)
DIR *dirp;

void seekdir (dirp, loc)
DIR *dirp;
long loc;

void rewinddir (dirp)
DIR *dirp;

void closedir(dirp)
DIR *dirp;
```

DESCRIPTION

opendir opens the directory named by *filename* and associates a *directory stream* with it. *opendir* returns a pointer to be used to identify the *directory stream* in subsequent operations. The pointer NULL is returned if *filename* cannot be accessed or is not a directory, or if it cannot *malloc*(3X) enough memory to hold a DIR structure or a buffer for the directory entries.

readdir returns a pointer to the next active directory entry. No inactive entries are returned. It returns NULL upon reaching the end of the directory or upon detecting an invalid location in the directory.

telldir returns the current location associated with the named *directory stream.*

seekdir sets the position of the next *readdir* operation on the *directory stream.* The new position reverts to the one associated with the *directory stream* when the *telldir* operation from which *loc* was obtained was performed. Values returned by *telldir* are good only if the directory has not changed due to compaction or expansion. This is not a problem with System V, but it may be with some file system types.

rewinddir resets the position of the named *directory stream* to the beginning of the directory.

closedir closes the named *directory stream* and frees the DIR structure. The following errors can occur as a result of these operations.

opendir:

[ENOTDIR]	A component of *filename* is not a directory.
[EACCES]	A component of *filename* denies search permission.
[EMFILE]	The maximum number of file descriptors are currently open.
[EFAULT]	*filename* points outside the allocated address space.

readdir:

[ENOENT]	The current file pointer for the directory is not located at a valid entry.
[EBADF]	The file descriptor determined by the DIR stream is no longer valid. This results if the DIR stream has been closed.

telldir, seekdir, and *closedir:*

[EBADF] The file descriptor determined by the DIR stream is no longer valid. This results if the DIR stream has been closed.

EXAMPLE

Sample code which searches a directory for entry *name*:

```
dirp = opendir( "." );
while ( (dp = readdir( dirp )) != NULL )
        if ( strcmp( dp->d_name, name ) == 0 )
                {
                closedir( dirp );
                return FOUND;
                }
closedir( dirp );
return NOT_FOUND;
```

SEE ALSO

getdents(2), dirent(4).

WARNINGS

rewinddir is implemented as a macro, so its function address cannot be taken.

NAME

dprod – double precision product intrinsic function

SYNOPSIS

```
real a1, a2

double precision a3

a3 = dprod(a1, a2)
```

DESCRIPTION

Dprod returns the double precision product of its real arguments.

NAME

drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48, lcong48 – generate uniformly distributed pseudo-random numbers

SYNOPSIS

double drand48 ( )

double erand48 (xsubi)
unsigned short xsubi[3];

long lrand48 ( )

long nrand48 (xsubi)
unsigned short xsubi[3];

long mrand48 ( )

long jrand48 (xsubi)
unsigned short xsubi[3];

void srand48 (seedval)
long seedval;

unsigned short *seed48 (seed16v)
unsigned short seed16v[3];

void lcong48 (param)
unsigned short param[7];

DESCRIPTION

This family of functions generates pseudo-random numbers using the well-known linear congruential algorithm and 48-bit integer arithmetic.

Functions *drand48* and *erand48* return non-negative double-precision floating-point values uniformly distributed over the interval (0.0, 1.0).

Functions *lrand48* and **nrand48** return non-negative long integers uniformly distributed over the interval $(0, 2^{31})$.

Functions *mrand48* and *jrand48* return signed long integers uniformly distributed over the interval $(-2^{31}, 2^{31})$.

Functions *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 will be supplied automatically if *drand48, lrand48* or *mrand48* is called without a prior call to an initialization entry point.) Functions *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

$$X_{n+1} = (aX_n + c)_{\text{mod } m} \qquad n \geq 0.$$

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, and 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. These routines do not have to be initialized; the calling program must place the desired initial value of X_i into the array and pass it as an argument. By using different arguments, functions *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 will *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* will restore the "standard" multiplier and addend values, a and c, specified on the previous page.

NOTES

The source code for the portable version can be used on computers which do not have floating-point arithmetic. In such a situation, functions *drand48* and *erand48* are replaced by the two new functions below.

long irand48 (m)
unsigned short m;

long krand48 (xsubi, m)
unsigned short xsubi[3], m;

Functions *irand48* and *krand48* return non-negative long integers uniformly distributed over the interval (0, $m-1$).

SEE ALSO

rand(3C). f'

NAME

dup2 – duplicate an open file descriptor

SYNOPSIS

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

DESCRIPTION

fildes is a file descriptor referring to an open file, and *fildes2* is a non-negative integer less than NOFILES. *dup2* causes *fildes2* to refer to the same file as *fildes*. If *fildes2* already referred to an open file, it is closed first.

dup2 will fail if one or more of the following are true:

[EBADF]	*fildes* is not a valid open file descriptor.
[EMFILE]	NOFILES file descriptors are currently open.

SEE ALSO

creat(2), close(2), exec(2), fcntl(2), open(2), pipe(2), lockf(3C).

DIAGNOSTICS

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.

NAME

ecvt, fcvt, gcvt – convert floating-point number to string

SYNOPSIS

```
char *ecvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *gcvt (value, ndigit, buf)
double value;
int ndigit;
char *buf;
```

DESCRIPTION

ecvt converts *value* to a null-terminated string of *ndigit* digits and returns a pointer thereto. The high-order digit is non-zero, unless the value is zero. The low-order digit is rounded. The position of the decimal point relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). The decimal point 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.

fcvt is 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 attempts to produce *ndigit* significant digits in FORTRAN F-format if possible, otherwise E-format, ready for printing. A minus sign, if there is one, or a decimal point will be included as part of the returned string. Trailing zeros are suppressed.

SEE ALSO

printf(3S).

BUGS

The values returned by *ecvt* and *fcvt* point to a single static data array whose content is overwritten by each call.

NAME

emulate_branch – MIPS branch emulation

SYNOPSIS

#include <signal.h>

emulate_branch(scp, branch_instruction)
struct sigcontext *scp;
unsigned long branch_instruction;

DESCRIPTION

Emulate_branch is passed a signal context structure and a branch instruction. It emulates the branch based on the register values in the signal context structure. It modifies the value of the program counter in the signal context structure (*sc_pc*) to the target of the branch instruction. The program counter must initially be pointing at the branch and the register values must be those at the time of the branch. If the branch is not taken the program counter is advanced to point to the instruction after the delay slot (*sc_pc* += 8).

In the case the branch instruction is a *branch on coprocessor 2 or 3* instruction *emulate_branch* can't emulate or execute the branch currently.

RETURN VALUE

Emulate_branch returns a 0 if the branch was emulated successfully. An non-zero value indicates the value passed as a branch instruction was not a branch instruction.

ALSO SEE

signal(2), sigset(2)

NAME

end, etext, edata – last locations in program
eprol, _ftext, _fdata, _fbss – first locations in program
_procedure_table, _procedure_table_size, _procedure_string_table – runtime procedure table

SYNOPSIS

```
#include <syms.h>
extern _END;
extern _ETEXT;
extern _EDATA;
extern eprol;
extern _FTEXT;
extern _FDATA;
extern _FBSS;
extern _PROCEDURE_TABLE;
extern _PROCEDURE_TABLE_SIZE;
extern _PROCEDURE_STRING_TABLE;
```

DESCRIPTION

These names refer neither to routines nor to locations with interesting contents except for _PROCEDURE_TABLE and _PROCEDURE_STRING_TABLE. Except for *eprol* these are all names of loader defined symbols. The address of _ETEXT is the first address above the program text, _EDATA is above the initialized data region, _END is above the uninitialized data region, and *eprol* is the first instruction of the user's program that follows the runtime startup routine.

When execution begins, the program break coincides with _END, but it is reset by the routines *brk*(2), *malloc*(3), standard input/output (*stdio*(3)), the profile (**-p**) option of *cc*(1), etc. The current value of the program break is reliably returned by 'sbrk(0)', see *brk*(2).

The loader defined symbols _PROCEDURE_TABLE, _PROCEDURE_TABLE_SIZE and _PROCEDURE_STRING_TABLE refer to the data structures of the runtime procedure table. Since these are loader defined symbols the data structures are build by *ld*(1) only if they are referenced. See the include file *<sym.h>* for the definition of the runtime procedure table and see the include file *<exception.h>* for its uses.

SEE ALSO

brk(2), malloc(3)

NAME

erf, erfc – error functions

SYNOPSIS

```
#include <math.h>

double erf(x)
double x;

double erfc(x)
double x;
```

DESCRIPTION

Erf (x) returns the error function of x; where erf (x) := $(2/\sqrt{\pi}) \int_0^x \exp(-t^2)\, dt$.

Erfc (x) returns 1.0–erf (x).

The entry for erfc is provided because of the extreme loss of relative accuracy if erf (x) is called for large x and the result subtracted from 1. (e.g. for x = 10, 12 places are lost).

SEE ALSO

math(3M)

NAME

ethers, ether_ntoa, ether_aton, ether_ntohost, ether_hostton, ether_line – Ethernet address mapping operations

SYNOPSIS

```
#include <sys/types.h>
#include <bsd/sys/socket.h>
#include <bsd/net/if.h>
#include <bsd/netinet/in.h>
#include <bsd/netinet/if_ether.h>

char *
ether_ntoa(e)
        struct ether_addr *e;

struct ether_addr *
ether_aton(s)
        char *s;

ether_ntohost(hostname, e)
        char *hostname;
        struct ether_addr *e;

ether_hostton(hostname, e)
        char *hostname;
        struct ether_addr *e;

ether_line(l, e, hostname)
        char *l;
        struct ether_addr *e;
        char *hostname;
```

DESCRIPTION

These routines are useful for mapping 48 bit Ethernet numbers to their ASCII representations or their corresponding host names, and vice versa.

The function *ether_ntoa* converts a 48 bit Ethernet number pointed to by *e* to its standard ASCII representation; it returns a pointer to the ASCII string. The representation is of the form: "x:x:x:x:x:x" where *x* is a hexadecimal number between 0 and ff. The function *ether_aton* converts an ASCII string in the standard representation back to a 48 bit Ethernet number; the function returns NULL if the string cannot be scanned successfully.

The function *ether_ntohost* maps an Ethernet number (pointed to by *e*) to its associated host-name. The string pointed to by *hostname* must be long enough to hold the hostname and a null character. The function returns zero upon success and non-zero upon failure. Inversely, the function *ether_hostton* maps a hostname string to its corresponding Ethernet number; the function modifies the Ethernet number pointed to by *e*. The function also returns zero upon success and non-zero upon failure.

The function *ether_line* scans a line (pointed to by *l*) and sets the hostname and the Ethernet number (pointed to by *e*). The string pointed to by *hostname* must be long enough to hold the hostname and a null character. The function returns zero upon success and non-zero upon failure. The format of the scanned line is described by *ethers*(4).

FILES

/etc/ethers (or the yellowpages' maps ethers.byaddr and ethers.byname)

SEE ALSO

ethers(4)

NAME

ether_ntoa, ether_aton, ether_ntohost, ether_hostton, ether_line – ethernet address mapping operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <net/if.h>
#include <netinet/in.h>
#include <netinet/if_ether.h>

char *
ether_ntoa(e)
        struct ether_addr *e;

struct ether_addr *
ether_aton(s)
        char *s;

ether_ntohost(hostname, e)
        char *hostname;
        struct ether_addr *e;

ether_hostton(hostname, e)
        char *hostname;
        struct ether_addr *e;

ether_line(l, e, hostname)
        char *l;
        struct ether_addr *e;
        char *hostname;
```

DESCRIPTION

These routines are useful for mapping 48 bit ethernet numbers to their ASCII representations or their corresponding host names, and vice versa.

The function *ether_ntoa* converts a 48 bit ethernet number pointed to by *e* to its standard ASCII representation; it returns a pointer to the ASCII string. The representation is of the form: "x:x:x:x:x:x" where *x* is a hexadecimal number between 0 and ff. The function *ether_aton* converts an ASCII string in the standard representation back to a 48 bit ethernet number; the function returns NULL if the string cannot be scanned successfully.

The function *ether_ntohost* maps an ethernet number (pointed to by *e*) to its associated hostname. The string pointed to by *hostname* must be long enough to hold the hostname and a null character. The function returns zero upon success and non-zero upon failure. Inversely, the function *ether_hostton* maps a hostname string to its corresponding ethernet number; the function modifies the ethernet number pointed to by *e*. The function also returns zero upon success and non-zero upon failure.

The function *ether_line* scans a line (pointed to by *l*) and sets the hostname and the ethernet number (pointed to by *e*). The string pointed to by *hostname* must be long enough to hold the hostname and a null character. The function returns zero upon success and non-zero upon failure.

SEE ALSO

ethers(4)

ORIGIN

Sun Microsystems

NAME

etime, dtime – return elapsed execution time

SYNOPSIS

function etime (tarray)
real tarray(2)

function dtime (tarray)
real tarray(2)

DESCRIPTION

These two routines return elapsed runtime in seconds for the calling process. *Dtime* returns the elapsed time since the last call to *dtime,* or the start of execution on the first call.

The argument array returns user time in the first element and system time in the second element. The function value is the sum of user and system time.

The resolution of all timing is 1/HZ sec. where HZ is currently 60.

FILES

/usr/lib/libU77.a

SEE ALSO

times(2)

NAME

examples – library of sample programs

SYNOPSIS

examples

DESCRIPTION

examples is a library containing sample programs to illustrate Ada language use and demonstrate the capabilities of the language, including those provided by the packages in the *standard*, *verdixlib*, and *publiclib* libraries.

Note: programs in the **examples** are neither supported nor warranted by MIPS.

The directory contains the program files listed below.

arguments.a	uses package COMMAND_LINE from *verdixlib* to print program arguments and environment variables.
date	uses package CALENDAR from *standard* to print current date and time.
hanoi.a, *termbody.a*, *termspec.a*	demonstrates solution to "Towers of Hanoi" problem.
hello	a typical first program, which uses package TEXT_IO from *standard* to print the message "hello, world".
mortgage.a	uses package MATH from *verdixlib* to calculate mortgage payments.
queens.a	provides a solution of the "8 Queens" chess problem gerneralized for any board with sides of 4-12 squares.
random.a	uses packages CALENDAR from *standard* to create pseudo-random numbers.
slideshow.a	uses the package CURSES in *publiclib* and illustrates background tasks.
sort_file	sorts lines in a file within specifies columns.
sort_integer.a	uses packages ORDERING form *verdixlib* to sort input of IO integer in ascending and descending order.
uc.p, *uctran.a*	uses package CALENDAR from *standard* to maintain a calendar file; these illustrate the translation of a program from Pascal to Ada. *uc.p* is in Pascal, and *uctran.a* is a close translation of UC.PAS to Ada.

FILES

*/usr/vads5/examples/**

SEE ALSO

publiclib, *standard*, *verdixlib*

NAME

exp, dexp, cexp – Fortran exponential intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
complex cx1, cx2
r2 = exp(r1)
dp2 = dexp(dp1)
dp2 = exp(dp1)
cx2 = cexp(cx1)
cx2 = exp(cx1)
```

DESCRIPTION

exp returns the real exponential function e^x of its real argument. *dexp* returns the double-precision exponential function of its double-precision argument. *cexp* returns the complex exponential function of its complex argument. The generic function *exp* becomes a call to *dexp* or *cexp* as required, depending on the type of its argument.

SEE ALSO

exp(3M).

NAME

exp, expm1, log, log10, log1p, pow – exponential, logarithm, power

SYNOPSIS

```
#include <math.h>

double exp(x)
double x;

float fexp(float x)
float x;

double expm1(x)
double x;

float fexpm1(float x)
float x;

double log(x)
double x;

float flog(float x)
float x;

double log10(x)
double x;

float flog10(float x)
float x;

double log1p(x)
double x;

float flog1p(float x)
float x;

double pow(x,y)
double x,y;
```

DESCRIPTION

Exp and fexp returns the exponential function of x for double and float data types respectively.

Expm1 and fexpm1 returns exp(x)−1 accurately even for tiny x for double and float data types respectively.

Log and flog returns the natural logarithm of x for double and float data types respectively.

Log10 and flog10 returns the logarithm of x to base 10 for double and float data types respectively.

Log1p and flog1p returns log(1+x) accurately even for tiny x for double and float data types respectively.

Pow(x,y) returns x^y.

ERROR (due to Roundoff etc.)

exp(x), log(x), expm1(x) and log1p(x) are accurate to within an *ulp*, and log10(x) to within about 2 *ulps*; an *ulp* is one *U*nit in the *L*ast *P*lace. The error in pow(x,y) is below about 2 *ulps* when its magnitude is moderate, but increases as pow(x,y) approaches the over/underflow thresholds until almost as many bits could be lost as are occupied by the floating-point format's exponent field; 11 bits for IEEE 754 Double. No such drastic loss has been exposed by testing; the worst errors observed have been below 300 *ulps* for IEEE 754 Double. Moderate values of pow are accurate enough that pow(integer,integer) is exact until it is bigger

than 2**53 for IEEE 754 Double.

DIAGNOSTICS

exp returns ∞ when the correct value would overflow, or the smallest non-zero value when the correct value would underflow.

Log and *log10* returns the default quiet *NaN* when *x* is less than zero indicating the invalid operation. *Log* and *log10* returns $-\infty$ when *x* is zero.

Pow returns ∞ when *x* is 0 and *y* is non-positive. *Pow* returns *NaN* when *x* is negative and *y* is not an integer indicating the invalid operation. When the correct value for *pow* would overflow or underflow, *pow* returns $\pm\infty$ or 0 respectively.

NOTES

Pow(x,0) returns x**0 = 1 for all x including x = 0, ∞ , and *NaN*. Previous implementations of pow may have defined x**0 to be undefined in some or all of these cases. Here are reasons for returning x**0 = 1 always:

(1) Any program that already tests whether x is zero (or infinite or *NaN*) before computing x**0 cannot care whether 0**0 = 1 or not. Any program that depends upon 0**0 to be invalid is dubious anyway since that expression's meaning and, if invalid, its consequences vary from one computer system to another.

(2) Some Algebra texts (e.g. Sigler's) define x**0 = 1 for all x, including x = 0. This is compatible with the convention that accepts a[0] as the value of polynomial

$$p(x) = a[0]*x**0 + a[1]*x**1 + a[2]*x**2 +...+ a[n]*x**n$$

at x = 0 rather than reject a[0]*0**0 as invalid.

(3) Analysts will accept 0**0 = 1 despite that x**y can approach anything or nothing as x and y approach 0 independently. The reason for setting 0**0 = 1 anyway is this:

If x(z) and y(z) are *any* functions analytic (expandable in power series) in z around z = 0, and if there x(0) = y(0) = 0, then x(z)**y(z) $\rightarrow$ 1 as z $\rightarrow$ 0.

(4) If 0**0 = 1, then ∞**0 = 1/0**0 = 1 too; and then *NaN***0 = 1 too because x**0 = 1 for all finite and infinite x, i.e., independently of x.

SEE ALSO

math(3M)

AUTHOR

Kwok-Choi Ng, W. Kahan

NAME

fclose, fflush – close or flush a stream

SYNOPSIS

#include <stdio.h>

int fclose (stream)
FILE *stream;

int fflush (stream)
FILE *stream;

DESCRIPTION

fclose causes any buffered data for the named *stream* to be written out, and the *stream* to be closed.

fclose is performed automatically for all open files upon calling *exit*(2).

fflush causes any buffered data for the named *stream* to be written to that file. The *stream* remains open.

SEE ALSO

close(2), exit(2), fopen(3S), setbuf(3S), stdio(3S).

DIAGNOSTICS

These functions return 0 for success, and **EOF** if any error (such as trying to write to a file that has not been opened for writing) was detected.

NAME

fdate – return date and time in an ASCII string

SYNOPSIS

subroutine fdate (string)
character*(*) string

character*(*) function fdate()

DESCRIPTION

Fdate returns the current date and time as a 24 character string in the format described under *ctime*(3). Neither 'newline' nor NULL will be included.

Fdate can be called either as a function or as a subroutine. If called as a function, the calling routine must define its type and length. For example:

```
character*24   fdate
external       fdate

write(*,*) fdate()
```

FILES

/usr/lib/libU77.a

SEE ALSO

ctime(3), time(3F), itime(3F), idate(3F), ltime(3F)

NAME

ferror, feof, clearerr, fileno – stream status inquiries

SYNOPSIS

```
#include <stdio.h>

int ferror (stream)
FILE *stream;

int feof (stream)
FILE *stream;

void clearerr (stream)
FILE *stream;

int fileno (stream)
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.

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 to zero on the named *stream*.

fileno returns the integer file descriptor associated with the named *stream*; see *open*(2).

NOTES

All these functions are implemented as macros; they cannot be declared or redeclared.

SEE ALSO

open(2), fopen(3S), stdio(3S).

NAME

fabs, floor, ceil, rint – absolute value, floor, ceiling, and round-to-nearest functions

SYNOPSIS

```
#include <math.h>

double floor(x)
double x;

float ffloor(float x)
float x;

double ceil(x)
double x;

float fceil(float x)
float x;

double trunc(x)
double x;

float ftrunc(float x)
float x;

double fabs(x)
double x;

double rint(x)
double x;

double fmod (x, y)
double x, y;
```

DESCRIPTION

Floor and ffloor returns the largest integer no greater than x for double and float data types respectively.

Ceil and fceil returns the smallest integer no less than x for double and float data types respectively.

Trunc and ftrunc returns the integer (represented as a floating-point number) of x with the fractional bits truncated for double and float data types respectively.

Fabs returns the absolute value |x|.

Rint returns the integer (represented as a double precision number) nearest x in the direction of the prevailing rounding mode.

Fmod returns the floating-point remainder of the division of x by y: zero if y is zero or if x/y would overflow; otherwise the number f with the same sign as x, such that $x = iy + f$ for some integer i, and $|f| < |y|$.

NOTES

In the default rounding mode, to nearest, 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 towards zero.

Another way to obtain an integer near x is to declare (in C)

```
double x;    int k;    k = x;
```

The MIPS C compilers rounds x towards 0 to get the integer k. Also 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.

The routine fabs is in libc.a rather than libm.a.

SEE ALSO

abs(3), ieee(3M), math(3M)

NAME

flush – flush output to a logical unit

SYNOPSIS

subroutine flush (lunit)

DESCRIPTION

Flush causes the contents of the buffer for logical unit *lunit* to be flushed to the associated file. This is most useful for logical units 0 and 6 when they are both associated with the control terminal.

FILES

/usr/lib/libI77.a

SEE ALSO

fclose(3S)

NAME

fopen, freopen, fdopen – open a stream

SYNOPSIS

```
#include <stdio.h>

FILE *fopen (filename, type)
char *filename, *type;

FILE *freopen (filename, type, stream)
char *filename, *type;
FILE *stream;

FILE *fdopen (fildes, type)
int fildes;
char *type;
```

DESCRIPTION

fopen opens the file named by *filename* and associates a *stream* with it. *fopen* returns a pointer to the FILE structure associated with the *stream*.

filename points to a character string that contains the name of the file to be opened.

type is a character string having one of the following values:

"r"	open for reading
"w"	truncate or create for writing
"a"	append; open for writing at end of file, or create for writing
"r+"	open for update (reading and writing)
"w+"	truncate or create for update
"a+"	append; open or create for update at end-of-file

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*.

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*(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.

When a file is opened for update, both input and output may be done on the resulting *stream*. However, output may not be directly followed by input without an intervening *fseek* or *rewind*, and input may not be directly followed by output without an intervening *fseek*, *rewind*, or an input operation which 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. *fseek* may be used to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is disregarded. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output. If two separate processes open the same file for append, each process may write freely to the file without fear of destroying output being written by the other. The output from the two processes will be intermixed in the file in the order in which it is written.

SEE ALSO

creat(2), dup(2), open(2), pipe(2), fclose(3S), fseek(3S), stdio(3S).

DIAGNOSTICS

fopen, *fdopen*, and *freopen* return a NULL pointer on failure.

NAME

fork – create a copy of this process

SYNOPSIS

integer function fork()

DESCRIPTION

Fork creates a copy of the calling process. The only distinction between the 2 processes is that the value returned to one of them (referred to as the 'parent' process) will be the process id of the copy. The copy is usually referred to as the 'child' process. The value returned to the 'child' process will be zero.

All logical units open for writing are flushed before the fork to avoid duplication of the contents of I/O buffers in the external file(s).

If the returned value is negative, it indicates an error and will be the negation of the system error code. See perror(3F).

A corresponding *exec* routine has not been provided because there is no satisfactory way to retain open logical units across the exec. However, the usual function of *fork/exec* can be performed using *system*(3F).

FILES

/usr/lib/libU77.a

SEE ALSO

fork(2), wait(3F), kill(3F), system(3F), perror(3F)

NAME

fp_class – classes of IEEE floating-point values

SYNOPSIS

#include <fp_class.h>

int fp_class_d(double x);

int fp_class_f(float x);

DESCRIPTION

These routines are used to determine the class of IEEE floating-point values. They return one of the constants in the file *<fp_class.h>* and never cause an exception even for signaling NaN's. These routines are to implement the recommended function class(x) in the appendix of the IEEE 754-1985 standard for binary floating-point arithmetic.

The constants in *<fp_class.h>* refer to the following classes of values:

Constant	Class
FP_SNAN	Signaling NaN (Not-a-Number)
FP_QNAN	Quiet NaN (Not-a-Number)
FP_POS_INF	+∞ (positive infinity)
FP_NEG_INF	-∞ (negative infinity)
FP_POS_NORM	positive normalized non-zero
FP_NEG_NORM	negative normalized non-zero
FP_POS_DENORM	positive denormalized
FP_NEG_DENORM	negative denormalized
FP_POS_ZERO	+0.0 (positive zero)
FP_NEG_ZERO	-0.0 (negative zero)

ALSO SEE

ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic

NAME

fpc – floating-point control registers

SYNOPSIS

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

int get_fpc_csr()

int set_fpc_csr(csr)
int csr;

int get_fpc_irr()

int get_fpc_eir()

void set_fpc_led(value)
int value;

int swapRM(x)
int x;

int swapINX(x)
int x;
```

DESCRIPTION

These routines are to get and set the floating-point control registers of MIPS floating-point units. All of these routines take and or return their values as 32 bit integers.

The file *<sys/fpu.h>* contains unions for each of the control registers. Each union contains a structure that breaks out the bit fields into the logical parts for each control register. This file also contains constants for fields of the control registers.

All implementations of MIPS floating-point have a *control and status* register and a *implementation revsion* register. The *control and status* register is returned by *get_fpc_csr*. The routine *set_fpc_csr* sets the *control and status* register and returns the old value. The *implementation revsion* register is read-only and is returned by the routine *get_fpc_irr*.

The R2360 floating-point units (floating-point boards) have two addtitonal control registers. The *exception instruction* register is a read-only register and is returned by the routine *get_fpc_eir*. The other floating-point control register on the R2360 is the *leds* register. The low 8 bits corresponds to the leds where a one is off and a zero is on. The *leds* register is a write-only register and is set with the routine *set_fpc_leds*.

The routine *swapRN* sets only the rounding mode and returns the old rounding mode. The routine *swapINX* sets only the sticky inexact bit and returns the old one. The bits in the arguments and return values to *swapRN* and *swapINX* are right justified.

ALSO SEE

R2010 Floating Point Coprocessor Architecture
R2360 Floating Point Board Product Description

NAME

fpgetround, fpsetround, fpgetmask, fpsetmask, fpgetsticky, fpsetsticky – IEEE floating point environment control

SYNOPSIS

```
#include <ieeefp.h>

typedef enum {
        FP_RN=0,    /* round to nearest */
        FP_RP,      /* round to plus */
        FP_RM,      /* round to minus */
        FP_RZ,      /* round to zero (truncate) */
        } fp_rnd;

        fp_rnd fpgetround();

        fp_rnd fpsetround(rnd_dir)
        fp_rnd rnd_dir;

        #define         fp_except       int
        #define FP_X_INV        0x10    /* invalid operation exception*/
        #define FP_X_OFL        0x08    /* overflow exception*/
        #define FP_X_UFL        0x04    /* underflow exception*/
        #define FP_X_DZ         0x02    /* divide-by-zero exception*/
        #define FP_X_IMP        0x01    /* imprecise (loss of precision)*/

        fp_except fpgetmask();

        fp_except fpsetmask(mask);
        fp_except mask;

        fp_except fpgetsticky();

        fp_except fpsetsticky(sticky);
        fp_except sticky;
```

DESCRIPTION

There are five floating point exceptions: divide-by-zero, overflow, underflow, imprecise (inexact) result, and invalid operation. When a floating point exception occurs, the corresponding sticky bit is set (1), and if the mask bit is enabled (1), the trap takes place. These routines let the user change the behavior on occurrence of any of these exceptions, as well as change the rounding mode for floating point operations.

fpgetround() returns the current rounding mode.

fpsetround() sets the rounding mode and returns the previous rounding mode.

fpgetmask() returns the current exception masks.

fpsetmask() sets the exception masks and returns the previous setting.

fpgetsticky() returns the current exception sticky flags.

fpsetsticky() sets (clears) the exception sticky flags and returns the previous setting.

The default environment on the 3B computer family is:

Rounding mode set to nearest(FP_RN),
Divide-by-zero,
Floating point overflow, and
Invalid operation traps enabled.

SEE ALSO

isnan(3C).

WARNINGS

fpsetsticky() modifies all sticky flags. *fpsetmask*() changes all mask bits.

Both C and F77 require truncation (round to zero) for floating point to integral conversions. The current rounding mode has no effect on these conversions.

CAVEATS

One must clear the sticky bit to recover from the trap and to proceed. If the sticky bit is not cleared before the next trap occurs, a wrong exception type may be signaled. For the same reason, when calling *fpsetmask*() the user should make sure that the sticky bit corresponding to the exception being enabled is cleared.

NAME

fpi – floating-point interrupt analysis

SYNOPSIS

#include <fpi.h>

void fpi()

void print_fpicounts()

int fpi_counts[];

char *fpi_list[];

DESCRIPTION

MIPS floating-point units generate floating-point interrupts for some classes of operations that occur with low frequency. In these cases the system software then emulates the operation in software. As a program takes floating-point interrupts its performance degrades since the operations are emulated in software. The routines and counters described here are used to analyze the causes of floating-point interrupts.

The routine *fpi* makes a *sysmips*(2) [MIPS_FPSIGINT] system call to causes floating-point interrupts to generate a SIGFPE. It also sets up a special signal handler for SIGFPE's. On a floating-point interrupt that signal handler determines the precise cause of the interrupt and increments the appropriate counter in *fpi_counts[]*.

The routine *print_fpicounts* prints out the value of the counters and their description on *stderr* as in the following example:

```
source signaling NaN = 0
source quiet NaN = 10
source denormalized value = 23
move of zero = 83
negate of zero = 84
implemented only in software = 5
invalid operation = 96
divide by zero = 3837
destination overflow = 398
destination underflow = 489
```

The constants in the file *<fpi.h>* along the counters, *fpi_counts[]*, and the descriptive strings, *fpi_list[]*, can also be used to format messages.

LIMITATIONS

Fpi can't be used with programs that normally generate SIGFPE's.

ALSO SEE

R2010 Floating Point Coprocessor Architecture
R2360 Floating Point Board Product Description
sysmips(2) [MIPS_FPSIGINTR].

NAME

fread, fwrite – binary input/output

SYNOPSIS

```
#include <stdio.h>
#include <sys/types.h>

int fread (ptr, size, nitems, stream)
char *ptr;
int nitems;
size_t size;
FILE *stream;

int fwrite (ptr, size, nitems, stream)
char *ptr;
int nitems;
size_t size;
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), stdio(3S).

DIAGNOSTICS

fread and *fwrite* return the number of items read or written. If *nitems* is non-positive, no characters are read or written and 0 is returned by both *fread* and *fwrite*.

NAME

frexp, ldexp, modf – manipulate parts of floating-point numbers

SYNOPSIS

```
double frexp (value, eptr)
double value;
int *eptr;

double ldexp (value, exp)
double value;
int exp;

double modf (value, iptr)
double value, *iptr;
```

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** (defined in **<math.h>**) 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**.

NAME

fseek, ftell – reposition a file on a logical unit

SYNOPSIS

integer function fseek (lunit, offset, from)
integer offset, from

integer function ftell (lunit)

DESCRIPTION

lunit must refer to an open logical unit. *offset* is an offset in bytes relative to the position specified by *from*. Valid values for *from* are:

0 meaning 'beginning of the file'
1 meaning 'the current position'
2 meaning 'the end of the file'

The value returned by *fseek* will be 0 if successful, a system error code otherwise. (See perror(3F))

Ftell returns the current position of the file associated with the specified logical unit. The value is an offset, in bytes, from the beginning of the file. If the value returned is negative, it indicates an error and will be the negation of the system error code. (See perror(3F))

FILES

/usr/lib/libU77.a

SEE ALSO

fseek(3S), perror(3F)

NAME

fseek, rewind, ftell – reposition a file pointer in a stream

SYNOPSIS

#include <stdio.h>

int fseek (stream, offset, ptrname)
FILE *stream;
long offset;
int ptrname;

void rewind (stream)
FILE *stream;

long ftell (stream)
FILE *stream;

DESCRIPTION

fseek sets the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the beginning, from the current position, or from the end of the file, according as *ptrname* has the value 0, 1, or 2.

rewind(*stream*) is equivalent to *fseek*(*stream*, 0L, 0), 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 may be either input or output.

ftell returns the offset of the current byte relative to the beginning of the file associated with the named *stream*.

SEE ALSO

lseek(2), fopen(3S), popen(3S), stdio(3S), ungetc(3S).

DIAGNOSTICS

fseek returns non-zero for improper seeks, otherwise zero. An improper seek can be, for example, an *fseek* done on a file that has not been opened via *fopen*; in particular, *fseek* may not be used on a terminal, or on a file opened via *popen*(3S).

WARNING

Although on the UNIX system an offset returned by *ftell* is measured in bytes, and it is permissible to seek to positions relative to that offset, portability to non-UNIX systems requires that an offset be used by *fseek* directly. Arithmetic may not meaningfully be performed on such an offset, which is not necessarily measured in bytes.

NAME

ftw – walk a file tree

SYNOPSIS

```
#include <ftw.h>

int ftw (path, fn, depth)
char *path;
int (*fn) ( );
int depth;
```

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 [see *stat*(2)] containing information about the object, and an integer. Possible values of the integer, defined in the <ftw.h> header file, are FTW_F for a file, FTW_D for a directory, FTW_DNR for a directory that cannot be read, and FTW_NS for an object for which *lstat* could not successfully be executed. If the integer is FTW_DNR, descendants of that directory will not be processed. If the integer is FTW_NS, the **stat** structure will contain garbage. An example of an object that would cause FTW_NS to be passed to *fn* would be a file in a directory with read but without execute (search) permission.

ftw visits a directory before visiting any of its descendants.

The 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 nonzero 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 uses one file descriptor for each level in the tree. The *depth* argument limits the number of file descriptors so used. If *depth* is zero 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. *ftw* will run more quickly if *depth* is at least as large as the number of levels in the tree.

SEE ALSO

stat(2), malloc(3C).

BUGS

Because *ftw* is recursive, it is possible for it to terminate with a memory fault when applied to very deep file structures.

CAVEATS

ftw uses *malloc*(3C) to allocate dynamic storage during its operation. If *ftw* is forcibly terminated, such as by *longjmp* being executed by *fn* or an interrupt routine, *ftw* will not have a chance to free that storage, so it will remain permanently allocated. 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.

ftw uses *lstat*(2) instead of *stat*(2) to avoid symbolic link loops and symbolic links to nonexistent files.

NAME

ftype: int, ifix, idint, real, float, sngl, dble, cmplx, dcmplx, ichar, char – explicit Fortran type conversion

SYNOPSIS

```
integer i, j
real r, s
double precision dp, dq
complex cx
double complex dcx
character*1 ch
i = int(r)
i = int(dp)
i = int(cx)
i = int(dcx)
i = ifix(r)
i = idint(dp)
r = real(i)
r = real(dp)
r = real(cx)
r = real(dcx)
r = float(i)
r = sngl(dp)
dp = dble(i)
dp = dble(r)
dp = dble(cx)
dp = dble(dcx)
cx = cmplx(i)
cx = cmplx(i, j)
cx = cmplx(r)
cx = cmplx(r, s)
cx = cmplx(dp)
cx = cmplx(dp, dq)
cx = cmplx(dcx)
dcx = dcmplx(i)
dcx = dcmplx(i, j)
dcx = dcmplx(r)
dcx = dcmplx(r, s)
dcx = dcmplx(dp)
dcx = dcmplx(dp, dq)
dcx = dcmplx(cx)
i = ichar(ch)
ch = char(i)
```

DESCRIPTION

These functions perform conversion from one data type to another. The function **int** converts to *integer* form its *real*, *double precision*, *complex*, or *double complex* argument. If the argument is *real* or *double precision*, **int** returns the integer whose magnitude is the largest integer that does not exceed the magnitude of the argument and whose sign is the same as the sign of the argument (i.e. truncation). For complex types, the above rule is applied to the real part. **ifix** and **idint** convert only *real* and *double precision* arguments respectively. The function **real** converts to *real* form an *integer*, *double precision*, *complex*, or *double complex* argument. If the argument is *double precision* or *double complex*, as much precision is kept as is possible. If the argument is one of the complex types, the real part is returned. **float** and **sngl** convert

only *integer* and *double precision* arguments respectively. The function **dble** converts any *integer*, *real*, *complex*, or *double complex* argument to *double precision* form. If the argument is of a complex type, the real part is returned. The function **cmplx** converts its *integer*, *real*, *double precision*, or *double complex* argument(s) to *complex* form. The function **dcmplx** converts to *double complex* form its *integer*, *real*, *double precision*, or *complex* argument(s). Either one or two arguments may be supplied to **cmplx** and **dcmplx** . If there is only one argument, it is taken as the real part of the complex type and an imaginary part of zero is supplied. If two arguments are supplied, the first is taken as the real part and the second as the imaginary part. The function **ichar** converts from a character to an integer depending on the character's position in the collating sequence. The function **char** returns the character in the *i*th position in the processor collating sequence where *i* is the supplied argument. For a processor capable of representing *n* characters,

ichar(char(i)) = i for $0 \leq i < n$, and

char(ichar(ch)) = ch for any representable character *ch*.

NAME

gamma – log gamma function

SYNOPSIS

#include <math.h>

double gamma (x)
double x;

extern int signgam;

DESCRIPTION

gamma returns $\ln(|\Gamma(x)|)$, 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 may not be a non-positive integer.

The following C program fragment might be used to calculate Γ:

```
if ((y = gamma(x)) > LN_MAXDOUBLE)
        error( );
y = signgam * exp(y);
```

where LN_MAXDOUBLE is the least value that causes *exp*(3F) to return a range error, and is defined in the <*values.h*> header file.

SEE ALSO

exp(3F), values(5).

DIAGNOSTICS

For non-negative integer arguments **HUGE** is returned, and *errno* is set to **EDOM**. A message indicating SING error is printed on the standard error output.

If the correct value would overflow, *gamma* returns **HUGE** and sets *errno* to **ERANGE**.

NAME

getarg, iargc – return command line arguments

SYNOPSIS

subroutine getarg (k, arg)
character*(*) arg

function iargc ()

DESCRIPTION

A call to *getarg* will return the **k***th* command line argument in character string *arg*. The 0*th* argument is the command name.

Iargc returns the index of the last command line argument.

FILES

/usr/lib/libU77.a

SEE ALSO

getenv(3F), execve(2)

NAME

getc, fgetc – get a character from a logical unit

SYNOPSIS

integer function getc (char)
character char

integer function fgetc (lunit, char)
character char

DESCRIPTION

These routines return the next character from a file associated with a fortran logical unit, bypassing normal fortran I/O. *Getc* reads from logical unit 5, normally connected to the control terminal input.

The value of each function is a system status code. Zero indicates no error occurred on the read; −1 indicates end of file was detected. A positive value will be either a UNIX system error code or an f77 I/O error code. See perror(3F).

FILES

/usr/lib/libU77.a

SEE ALSO

getc(3S), intro(2), perror(3F)

NAME

getc, getchar, fgetc, getw – get character or word from a stream

SYNOPSIS

```
#include <stdio.h>

int getc (stream)
FILE *stream;

int getchar ()

int fgetc (stream)
FILE *stream;

int getw (stream)
FILE *stream;
```

DESCRIPTION

getc returns the next character (i.e., byte) from the named input *stream*, as 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 *macros*.

fgetc behaves like *getc*, but is a function rather than a macro. *fgetc* runs more slowly 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., integer) 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.

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), fread(3S), gets(3S), putc(3S), scanf(3S), stdio(3S).

DIAGNOSTICS

These functions return the constant **EOF** at end-of-file or upon an error. Because **EOF** is a valid integer, *ferror*(3S) should be used to detect *getw* errors.

WARNING

If the integer value returned by *getc*, *getchar*, or *fgetc* is stored into a character variable and 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.

CAVEATS

Because it is implemented as a macro, *getc* evaluates a *stream* argument more than once. In particular, **getc(*f++)** does not work sensibly. *fgetc* 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 read using *getw* on a different processor.

NAME

getcwd – get path-name of current working directory

SYNOPSIS

char *getcwd (buf, size)
char *buf;
int size;

DESCRIPTION

getcwd returns a pointer to the current directory path name. The value of *size* must be at least two greater than the length of the path-name to be returned.

If *buf* is a NULL pointer, *getcwd* will obtain *size* bytes of space using *malloc*(3C). In this case, the pointer returned by *getcwd* may be used as the argument in a subsequent call to *free*.

The function is implemented by using *popen*(3S) to pipe the output of the *pwd*(1) command into the specified string space.

EXAMPLE

```
void exit(), perror();
.
.
.
if ((cwd = getcwd((char *)NULL, 64)) == NULL) {
        perror("pwd");
        exit(2);
}
printf("%s\n", cwd);
```

SEE ALSO

malloc(3C), popen(3S).
pwd(1) in the *User's Reference Manual*.

DIAGNOSTICS

Returns **NULL** with *errno* set if *size* is not large enough, or if an error occurs in a lower-level function.

NAME

getcwd – get pathname of current working directory

SYNOPSIS

integer function getcwd (dirname)
character*(*) dirname

DESCRIPTION

The pathname of the default directory for creating and locating files will be returned in *dirname*. The value of the function will be zero if successful; an error code otherwise.

FILES

/usr/lib/libU77.a

SEE ALSO

chdir(3F), perror(3F)

BUGS

Pathnames can be no longer than MAXPATHLEN as defined in *<sys/param.h>*.

NAME

getenv – return value for environment name

SYNOPSIS

char *getenv (name)
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.

SEE ALSO

exec(2), putenv(3C), environ(5).

NAME

getenv – get value of environment variables

SYNOPSIS

subroutine getenv (ename, evalue)
character*(*) ename, evalue

DESCRIPTION

Getenv searches the environment list (see *environ*(7)) for a string of the form *ename*=*value* and returns *value* in *evalue* if such a string is present, otherwise fills *evalue* with blanks.

FILES

/usr/lib/libU77.a

SEE ALSO

environ(7), execve(2)

NAME

getgrent, getgrgid, getgrnam, setgrent, endgrent, fgetgrent – get group file entry

SYNOPSIS

```
#include <grp.h>

struct group *getgrent ( )

struct group *getgrgid (gid)
int gid;

struct group *getgrnam (name)
char *name;

void setgrent ( )

void endgrent ( )

struct group *fgetgrent (f)
FILE *f;
```

DESCRIPTION

getgrent, *getgrgid* and *getgrnam* each return pointers to an object with the following structure containing the broken-out fields of a line in the **/etc/group** file. Each line contains a "group" structure, defined in the *<grp.h>* header file.

```
struct group {
        char    *gr_name;   /* the name of the group */
        char    *gr_passwd; /* the encrypted group password */
        int     gr_gid;     /* the numerical group ID */
        char    **gr_mem;   /* vector of pointers to member names */
};
```

getgrent when first called returns a pointer to the first group structure in the file; thereafter, it returns a pointer to the next group structure in the file; so, successive calls may be used to search the entire file. *getgrgid* searches from the beginning of the file until a numerical 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. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

A call to *setgrent* has the effect of rewinding the group file to allow repeated searches. *endgrent* may be called to close the group file when processing is complete.

fgetgrent returns a pointer to the next group structure in the stream *f*, which matches the format of **/etc/group**.

FILES

/etc/group

SEE ALSO

getlogin(3C), getpwent(3C), group(4).

DIAGNOSTICS

A **NULL** pointer is returned on **EOF** or error.

WARNING

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.

CAVEAT

All information is contained in a static area, so it must be copied if it is to be saved.

NAME

gethostbyname, gethostbyaddr, gethostent, sethostent, endhostent – get network host entry

SYNOPSIS

```
#include <bsd/netdb.h>

extern int h_errno;

struct hostent *gethostbyname(name)
char *name;

struct hostent *gethostbyaddr(addr, len, type)
char *addr; int len, type;

struct hostent *gethostent()

sethostent(stayopen)
int stayopen;

endhostent()
```

DESCRIPTION

gethostbyname and *gethostbyaddr* each return a pointer to an object with the following structure. This structure contains either the information obtained from the name server, *named*(8), or broken-out fields from a line in */etc/hosts*. If the local name server is not running these routines do a lookup in */etc/hosts*.

```
struct  hostent {
        char    *h_name;        /* official name of host */
        char    **h_aliases;    /* alias list */
        int     h_addrtype;     /* host address type */
        int     h_length;       /* length of address */
        char    **h_addr_list;  /* list of addresses from name server */
};
#define h_addr  h_addr_list[0]  /* address, for backward compatibility */
```

The members of this structure are:

h_name	Official name of the host.
h_aliases	A zero terminated array of alternate names for the host.
h_addrtype	The type of address being returned; currently always AF_INET.
h_length	The length, in bytes, of the address.
h_addr_list	A zero terminated array of network addresses for the host. Host addresses are returned in network byte order.
h_addr	The first address in h_addr_list; this is for backward compatiblity.

sethostent allows a request for the use of a connected socket using TCP for queries. If the *stayopen* flag is non-zero, this sets the option to send all queries to the name server using TCP and to retain the connection after each call to *gethostbyname* or *gethostbyaddr*.

endhostent closes the TCP connection.

DIAGNOSTICS

Error return status from *gethostbyname* and *gethostbyaddr* is indicated by return of a null pointer. The external integer *h_errno* may then be checked to see whether this is a temporary failure or an invalid or unknown host.

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	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.

FILES

/etc/hosts

SEE ALSO

hosts(4), resolver(5)

CAVEAT

gethostent is defined, and *sethostent* and *endhostent* are redefined, when *libc* is built to use only the routines to lookup in */etc/hosts* and not the name server.

gethostent reads the next line of */etc/hosts*, opening the file if necessary.

sethostent is redefined to open and rewind the file. If the *stayopen* argument is non-zero, the hosts data base will not be closed after each call to *gethostbyname* or *gethostbyaddr*. *endhostent* is redefined to close the file.

ERRORS

All information is contained in a static area so it must be copied if it is to be saved. Only the Internet address format is currently understood.

NAME

getlog – get user's login name

SYNOPSIS

subroutine getlog (name)
character*(*) name

character*(*) function getlog()

DESCRIPTION

Getlog will return the user's login name or all blanks if the process is running detached from a terminal.

FILES

/usr/lib/libU77.a

SEE ALSO

getlogin(3)

NAME

getlogin – get login name

SYNOPSIS

char *getlogin ( );

DESCRIPTION

getlogin returns a pointer to the login name as found in **/etc/utmp**. It may 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 correct procedure for determining the login name is to call *cuserid*, or to call *getlogin* and if it fails to call *getpwuid*.

FILES

/etc/utmp

SEE ALSO

cuserid(3S), getgrent(3C), getpwent(3C), utmp(4).

DIAGNOSTICS

Returns the **NULL** pointer if *name* is not found.

CAVEAT

The return values point to static data whose content is overwritten by each call.

NAME

setmntent, getmntent, addmntent, endmntent, hasmntopt – get file system descriptor file entry

SYNOPSIS

```
#include <stdio.h>
#include <mntent.h>

FILE *setmntent(filep, type)
char *filep;
char *type;

struct mntent *getmntent(filep)
FILE *filep;

int addmntent(filep, mnt)
FILE *filep;
struct mntent *mnt;

char *hasmntopt(mnt, opt)
struct mntent *mnt;
char *opt;

int endmntent(filep)
FILE *filep;
```

DESCRIPTION

These routines replace the *getfsent* routines for accessing the file system description file */etc/fstab*. They are also used to access the mounted file system description file */etc/mtab*.

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*(3S). *getmntent* reads the next line from *filep* and returns a pointer to an object with the following structure containing the broken-out fields of a line in the filesystem description file, *<mntent.h>*. The fields have meanings described in *fstab*(4).

```
struct mntent {
        char    *mnt_fsname;    /* file system name */
        char    *mnt_dir;       /* file system path prefix */
        char    *mnt_type;      /* 4.2, nfs, swap, or xx */
        char    *mnt_opts;      /* ro, quota, etc. */
        int     mnt_freq;       /* dump frequency, in days */
        int     mnt_passno;     /* pass number on parallel fsck */
};
```

Addmntent adds the *mntent* structure *mnt* to the end of the open file *filep*. Note that *filep* has to be opened for writing if this is to work. *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.

FILES

/etc/fstab
/etc/mtab

SEE ALSO

fstab(4)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

ERRORS

The returned *mntent* structure points to static information that is overwritten in each call.

ORIGIN

Sun Microsystems

NAME

getnetent, getnetbyaddr, getnetbyname, setnetent, endnetent – get network entry

SYNOPSIS

```
#include </bsd/netdb.h>

struct netent *getnetent()

struct netent *getnetbyname(name)
char *name;

struct netent *getnetbyaddr(net, type)
long net;
int type;

setnetent(stayopen)
int stayopen;

endnetent()
```

DESCRIPTION

getnetent, *getnetbyname*, and *getnetbyaddr* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network data base, */etc/networks*.

```
struct  netent {
        char            *n_name;        /* official name of net */
        char            **n_aliases;    /* alias list */
        int             n_addrtype;     /* net number type */
        unsigned long   n_net;          /* net number */
};
```

The members of this structure are:

n_name The official name of the network.

n_aliases A zero terminated list of alternate names for the network.

n_addrtype The type of the network number returned; currently only AF_INET.

n_net The network number. Network numbers are returned in machine byte order.

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 net data base will not be closed after each call to *getnetbyname* or *getnetbyaddr*.

endnetent closes the file.

getnetbyname and *getnetbyaddr* sequentially search from the beginning of the file until a matching net name or net address and type is found, or until EOF is encountered. Network numbers are supplied in host order.

FILES

/etc/networks

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

ERRORS

All information is contained in a static area so it must be copied if it is to be saved. Only Internet network numbers are currently understood. Expecting network numbers to fit in no more than 32 bits is probably naive.

NAME

getopt – get option letter from argument vector

SYNOPSIS

```
int getopt (argc, argv, optstring)
int argc;
char **argv, *opstring;

extern char *optarg;
extern int optind, opterr;
```

DESCRIPTION

getopt returns the next option letter in *argv* that matches a letter in *optstring*. It supports all the rules of the command syntax standard (see *intro*(1)). So all new commands will adhere to the command syntax standard, they should use *getopts*(1) or *getopt*(3C) to parse positional parameters and check for options that are legal for that command.

optstring must contain the option letters the command using *getopt* will recognize; if a letter is followed by a colon, the option is expected to have an argument, or group of arguments, which must be separated from it by white space.

optarg is set to point to the start of the option-argument on return from *getopt*.

getopt places in **optind** the *argv* index of the next argument to be processed. **optind** is external and is initialized to **1** before the first call to *getopt*.

When all options have been processed (i.e., up to the first non-option argument), *getopt* returns **−1**. The special option "--" may be used to delimit the end of the options; when it is encountered, **−1** will be returned, and "--" will be skipped.

DIAGNOSTICS

getopt prints an error message on standard error and returns a question mark (?) when it encounters an option letter not included in *optstring* or no option-argument after an option that expects one. This error message may be disabled by setting **opterr** to **0**.

EXAMPLE

The following code fragment shows how one might process the arguments for a command that can take the mutually exclusive options **a** and **b**, and the option **o**, which requires an option-argument:

```
main (argc, argv)
int argc;
char **argv;
{
        int c;
        extern char *optarg;
        extern int optind;
        .
        .
        .
        while ((c = getopt(argc, argv, "abo:")) != -1)
                switch (c) {
                case 'a':
                        if (bflg)
                                errflg++;
                        else
                                aflg++;
                        break;
                case 'b':
                        if (aflg)
```

```
                        errflg++;
                else
                        bproc( );
                break;
        case 'o':
                ofile = optarg;
                break;
        case '?':
                errflg++;
        }
    if (errflg) {
        (void)fprintf(stderr, "usage: . . . ");
        exit (2);
    }
    for ( ; optind < argc; optind++) {
        if (access(argv[optind], 4)) {
        .
        .
        .
}
```

WARNING

Although the following command syntax rule (see *intro*(1)) relaxations are permitted under the current implementation, they should not be used because they may not be supported in future releases of the system. As in the **EXAMPLE** section above, **a** and **b** are options, and the option **o** requires an option-argument:

```
cmd -aboxxx file   (Rule 5 violation:  options with
      option-arguments must not be grouped with other options)
cmd -ab -oxxx file  (Rule 6 violation:  there must be
      white space after an option that takes an option-argument)
```

SEE ALSO

getopts(1), intro(1) in the *User's Reference Manual*.

NOTES

Changing the value of the variable **optind**, or calling *getopt* with different values of *argv*, may lead to unexpected results.

NAME

getpass – read a password

SYNOPSIS

char *getpass (prompt)
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 will terminate input and send an interrupt signal to the calling program before returning.

FILES

/dev/tty

WARNING

The above routine uses **<stdio.h>**, which causes it to increase the size of programs not otherwise using standard I/O, more than might be expected.

CAVEAT

The return value points to static data whose content is overwritten by each call.

NAME

getpid – get process id

SYNOPSIS

integer function getpid()

DESCRIPTION

Getpid returns the process ID number of the current process.

FILES

/usr/lib/libU77.a

SEE ALSO

getpid(2)

NAME

getprotoent, getprotobynumber, getprotobyname, setprotoent, endprotoent – get protocol entry

SYNOPSIS

```
#include </bsd/netdb.h>

struct protoent *getprotoent()

struct protoent *getprotobyname(name)
char *name;

struct protoent *getprotobynumber(proto)
int proto;

setprotoent(stayopen)
int stayopen

endprotoent()
```

DESCRIPTION

getprotoent, *getprotobyname*, and *getprotobynumber* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network protocol data base, */etc/protocols*.

```
struct  protoent {
        char    *p_name;        /* official name of protocol */
        char    **p_aliases;    /* alias list */
        int     p_proto;        /* protocol number */
};
```

The members of this structure are:

p_name The official name of the protocol.

p_aliases A zero terminated list of alternate names for the protocol.

p_proto The protocol number.

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 net data base will not be closed after each call to *getprotobyname* or *getprotobynumber*.

endprotoent closes the file.

getprotobyname and *getprotobynumber* sequentially search from the beginning of the file until a matching protocol name or protocol number is found, or until EOF is encountered.

FILES

/etc/protocols

SEE ALSO

protocols(4)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

ERRORS

All information is contained in a static area so it must be copied if it is to be saved. Only the Internet protocols are currently understood.

NAME

getuid, getgid – get user or group ID of the caller

SYNOPSIS

integer function getuid()

integer function getgid()

DESCRIPTION

These functions return the real user or group ID of the user of the process.

FILES

/usr/lib/libU77.a

SEE ALSO

getuid(2)

NAME

getpw – get name from UID

SYNOPSIS

int getpw (uid, buf)
int 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.

This routine is included only for compatibility with prior systems and should not be used; see *getpwent*(3C) for routines to use instead.

FILES

/etc/passwd

SEE ALSO

getpwent(3C), passwd(4).

DIAGNOSTICS

getpw returns non-zero on error.

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.

NAME

getpwent, getpwuid, getpwnam, setpwent, endpwent, fgetpwent – get password file entry

SYNOPSIS

```
#include <pwd.h>

struct passwd *getpwent ( )

struct passwd *getpwuid (uid)
int uid;

struct passwd *getpwnam (name)
char *name;

void setpwent ( )

void endpwent ( )

struct passwd *fgetpwent (f)
FILE *f;
```

DESCRIPTION

getpwent, *getpwuid* and *getpwnam* each returns a pointer to an object with the following structure containing the broken-out fields of a line in the */etc/passwd* file. Each line in the file contains a "passwd" structure, declared in the <*pwd.h*> header file:

```
struct passwd {
        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;
};
```

This structure is declared in <*pwd.h*> so it is not necessary to redeclare it.

The fields have meanings described in *passwd(4)*.

getpwent when first called returns a pointer to the first passwd structure in the file; thereafter, it returns a pointer to the next passwd structure in the file; so successive calls can be used to search the entire file. *getpwuid* searches from the beginning of the file until a numerical 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. If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

A call to *setpwent* has the effect of rewinding the password file to allow repeated searches. *endpwent* may be called to close the password file when processing is complete.

fgetpwent returns a pointer to the next passwd structure in the stream *f*, which matches the format of */etc/passwd*.

FILES

/etc/passwd

SEE ALSO

getlogin(3C), getgrent(3C), passwd(4).

DIAGNOSTICS

A NULL pointer is returned on Eof or error.

WARNING

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.

CAVEAT

All information is contained in a static area, so it must be copied if it is to be saved. The structure contains pointers, and this data must also be saved.

NAME

getrpcent, getrpcbyname, getrpcbynumber – get rpc entry

SYNOPSIS

```
#include <netdb.h>

struct rpcent *getrpcent()

struct rpcent *getrpcbyname(name)
char *name;

struct rpcent *getrpcbynumber(number)
int number;

setrpcent(stayopen)
int stayopen

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;    /* alias list */
        long    r_number;       /* rpc program number */
};
```

The members of this structure are:

r_name The name of the server for this rpc program.

r_aliases A zero terminated list of alternate names for the rpc program.

r_number The rpc program number for this service.

getrpcent reads the next line of the file, opening the file if necessary.

setrpcent opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to *getrpcent* (either directly, or indirectly through one of the other "getrpc" calls).

endrpcent closes the file.

getrpcbyname and *getrpcbynumber sequentially search from the beginning* matching rpc program name or program number is found, or until EOF is encountered.

FILES

/etc/rpc

SEE ALSO

rpc(4), rpcinfo(1M)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

ERRORS

All information is contained in a static area so it must be copied if it is to be saved.

ORIGIN

Sun Microsystems

NAME

gets, fgets – get a string from a stream

SYNOPSIS

```
#include <stdio.h>

char *gets (s)
char *s;

char *fgets (s, n, stream)
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, or 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.

SEE ALSO

ferror(3S), fopen(3S), fread(3S), getc(3S), scanf(3S), stdio(3S).

DIAGNOSTICS

If end-of-file is encountered and no characters have been read, no characters are transferred to *s* and a NULL pointer is returned. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a NULL pointer is returned. Otherwise *s* is returned.

NAME

getservent, getservbyport, getservbyname, setservent, endservent – get service entry

SYNOPSIS

```
#include </bsd/netdb.h>

struct servent *getservent()

struct servent *getservbyname(name, proto)
char *name, *proto;

struct servent *getservbyport(port, proto)
int port; char *proto;

setservent(stayopen)
int stayopen

endservent()
```

DESCRIPTION

Getservent, *getservbyname*, and *getservbyport* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network services data base, */etc/services*.

```
struct servent {
        char    *s_name;        /* official name of service */
        char    **s_aliases;    /* alias list */
        int     s_port;         /* port service resides at */
        char    *s_proto;       /* protocol to use */
};
```

The members of this structure are:

s_name The official name of the service.

s_aliases A zero terminated list of alternate names for the service.

s_port The port number at which the service resides. Port numbers are returned in network byte order.

s_proto The name of the protocol to use when contacting the service.

etservent 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 net data base will not be closed after each call to *getservbyname* or *getservbyport*.

endservent closes the file.

etservbyname and *getservbyport* sequentially search from the beginning of the file until a matching protocol name or port number is found, or until EOF is encountered. If a protocol name is also supplied (non-NULL), searches must also match the protocol.

FILES

/etc/services

SEE ALSO

getprotoent(3N), services(4)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

ERRORS

All information is contained in a static area so it must be copied if it is to be saved. Expecting port numbers to fit in a 32 bit quantity is probably naive.

NAME

getut: getutent, getutid, getutline, pututline, setutent, endutent, utmpname – access utmp file entry

SYNOPSIS

```
#include <utmp.h>

struct utmp *getutent ( )

struct utmp *getutid (id)
struct utmp *id;

struct utmp *getutline (line)
struct utmp *line;

void pututline (utmp)
struct utmp *utmp;

void setutent ( )

void endutent ( )

void utmpname (file)
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) */
        short   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. */
        time_t  ut_time;            /* time entry was made */
};
```

getutent reads in the next entry from a *utmp*-like file. If the file is not already open, it opens it. If it reaches the end of the file, it 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, then *getutid* will return a pointer to the first entry whose type is one of these four and whose *ut_id* field matches *id->ut_id*. If the end of file is reached without a match, it fails.

getutline searches forward from the current point in the *utmp* file until it finds an entry of the type LOGIN_PROCESS or USER_PROCESS which also has a *ut_line* string matching the *line->ut_line* string. If the end of file is reached without a match, it fails.

Pututline writes out the supplied *utmp* structure into the *utmp* file. It uses *getutid* to search forward for the proper place if it finds that it is not already at the proper place. It is expected that normally the user of *pututline* will have searched for the proper entry using one of the *getut* routines. If so, *pututline* will not search. If *pututline* does not find a matching slot for the new entry, it will add a new entry to the end of the file.

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 examined, from **/etc/utmp** to any other file. It is most often expected that this other file will be **/etc/wtmp**. If the file does not exist, this will not be apparent until the first attempt to reference the file is made. *utmpname* does not open the file. It just closes the old file if it is currently open and saves the new file name.

FILES

/etc/utmp
/etc/wtmp

SEE ALSO

ttyslot(3C), utmp(4).

DIAGNOSTICS

A NULL pointer is returned upon failure to read, whether for permissions or having reached the end of file, or upon failure to write.

NOTES

The most current entry is saved in a static structure. Multiple accesses require that it be copied before further accesses are made. Each call to either *getutid* or *getutline* sees the routine examine the static structure before performing more I/O. If the contents of the static structure match what it is searching for, it looks no further. For this reason to use *getutline* to search for multiple occurrences, it would be necessary to zero out the static after each success, or *getutline* would just return the same pointer over and over again. There is one exception to the rule about removing the structure before further reads are done. The implicit read done by *pututline* (if it finds that it is not already at the correct place in the file) will not hurt the contents of the static structure returned by the *getutent*, *getutid* or *getutline* routines, if the user has just modified those contents and passed the pointer back to *pututline*.

These routines use buffered standard I/O for input, but *pututline* uses an unbuffered nonstandard write to avoid race conditions between processes trying to modify the *utmp* and *wtmp* files.

NAME

hsearch, hcreate, hdestroy – manage hash search tables

SYNOPSIS

#include <search.h>

ENTRY *hsearch (item, action)
ENTRY item;
ACTION action;

int hcreate (nel)
unsigned nel;

void hdestroy ( )

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: *item.key* points to the comparison key, and *item.data* 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 may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances. *hdestroy* destroys the search table, and may be followed by another call to *hcreate*.

NOTES

hsearch uses *open addressing* with a *multiplicative* hash function. However, its source code has many other options available which the user may select by compiling the *hsearch* source with the following symbols defined to the preprocessor:

DIV Use the *remainder modulo table size* as the hash function instead of the multiplicative algorithm.

USCR Use a User Supplied Comparison Routine for ascertaining table membership. The routine should be named *hcompar* and should behave in a mannner similar to *strcmp* [see *string*(3C)].

CHAINED Use a linked list to resolve collisions. If this option is selected, the following other options become available.

- **START** Place new entries at the beginning of the linked list (default is at the end).
- **SORTUP** Keep the linked list sorted by key in ascending order.
- **SORTDOWN** Keep the linked list sorted by key in descending order.

Additionally, there are preprocessor flags for obtaining debugging printout (**–DDEBUG**) and for including a test driver in the calling routine (**–DDRIVER**). The source code should be consulted for further details.

EXAMPLE

The following example will read in strings followed by two numbers and store them in a hash table, discarding duplicates. It will then read in strings and find the matching entry in the hash table and print 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)
            }
        }
}
```

SEE ALSO

bsearch(3C), lsearch(3C), malloc(3C), malloc(3X), string(3C), tsearch(3C).

DIAGNOSTICS

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.

WARNING

hsearch and *hcreate* use *malloc*(3C) to allocate space.

CAVEAT

Only one hash search table may be active at any given time.

NAME

hypot, cabs – Euclidean distance, complex absolute value

SYNOPSIS

```
#include <math.h>

double hypot(x,y)
double x,y;

float fhypot(float x, float y)
double x,y;

double cabs(z)
struct {double x,y;} z;

float fcabs(z)
struct {float x,y;} z;
```

DESCRIPTION

Hypot(x,y), fhypot(x,y), cabs(x,y) and fcabs(x,y) return sqrt(x*x+y*y) computed in such a way that underflow will not happen, and overflow occurs only if the final result deserves it.

Fhypot and fcabs are the same functions as hypot and cabs but for the float data type.

hypot(∞,v) = hypot(v,∞) = +∞ for all v, including *NaN*.

DIAGNOSTICS

When the correct value would overflow, *hypot* returns +∞.

ERROR (due to Roundoff, etc.)

Below 0.97 *ulps*. Consequently hypot(5.0,12.0) = 13.0 exactly; in general, hypot and cabs return an integer whenever an integer might be expected.

The same cannot be said for the shorter and faster version of hypot and cabs that is provided in the comments in cabs.c; its error can exceed 1.2 *ulps*.

NOTES

As might be expected, hypot(v,*NaN*) and hypot(*NaN*,v) are *NaN* for all *finite* v. Programmers might be surprised at first to discover that hypot(±∞,*NaN*) = +∞. This is intentional; it happens because hypot(∞,v) = +∞ for *all* v, finite or infinite. Hence hypot(∞,v) is independent of v. The IEEE *NaN* is designed to disappear when it turns out to be irrelevant, as it does in hypot(∞,*NaN*).

SEE ALSO

math(3M), sqrt(3M)

AUTHOR

W. Kahan

NAME

iargc – return the number of command line arguments

SYNOPSIS

integer i

i = iargc( )

DESCRIPTION

The *iargc* function returns the number of command line arguments passed to the program. Thus, if a program were invoked via

foo arg1 arg2 arg3

iargc( ) would return 3.

NAME

idate, itime – return date or time in numerical form

SYNOPSIS

subroutine idate (iarray)
integer iarray(3)

subroutine itime (iarray)
integer iarray(3)

DESCRIPTION

Idate returns the current date in *iarray*. The order is: day, mon, year. Month will be in the range 1-12. Year will be $\geq$ 1969.

Itime returns the current time in *iarray*. The order is: hour, minute, second.

FILES

/usr/lib/libU77.a

SEE ALSO

ctime(3F), fdate(3F)

NAME

copysign, drem, finite, logb, scalb – copysign, remainder, exponent manipulations

SYNOPSIS

```
#include <math.h>

double copysign(x,y)
double x,y;

double drem(x,y)
double x,y;

int finite(x)
double x;

double logb(x)
double x;

double scalb(x,n)
double x;
int n;
```

DESCRIPTION

These functions are required for, or recommended by the IEEE standard 754 for floating–point arithmetic.

Copysign(x,y) returns x with its sign changed to y's.

Drem(x,y) 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 DIAGNOSTICS.

Finite(x) = 1 just when $-\infty < x < +\infty$,
= 0 otherwise (when $|x| = \infty$ or x is *NaN*)

Logb(x) returns x's exponent n, a signed integer converted to double–precision floating–point and so chosen that $1 \leq |x|/2^{**}n < 2$ unless x = 0 or $|x| = \infty$ or x lies between 0 and the Underflow Threshold.

Scalb(x,n) = x*(2**n) computed, for integer n, without first computing 2**n.

DIAGNOSTICS

IEEE 754 defines drem(x,0) and drem(∞,y) to be invalid operations that produce a *NaN*.

IEEE 754 defines logb(±∞) = +∞ and logb(0) = −∞, and requires the latter to signal Division–by–Zero.

SEE ALSO

floor(3M), fp_class(3), math(3M)

AUTHOR

Kwok–Choi Ng

BUGS

IEEE 754 currently specifies that logb(denormalized no.) = logb(tiniest normalized no. > 0) but the consensus has changed to the specification in the new proposed IEEE standard p854, namely that logb(x) satisfy

$1 \leq$ scalb(|x|,−logb(x)) < Radix ... = 2 for IEEE 754

for every x except 0, ∞ and *NaN*. Almost every program that assumes 754's specification will work correctly if logb follows 854's specification instead.

IEEE 754 requires copysign(x,*NaN*) = ±x but says nothing else about the sign of a *NaN*.

NAME

index – return location of Fortran substring

SYNOPSIS

```
character*N1 ch1
character*N2 ch2
integer i
i = index(ch1, ch2)
```

DESCRIPTION

index returns the location of substring *ch2* in string *ch1*. The value returned is the position at which substring *ch2* starts, or 0 if it is not present in string *ch1*. If N2 is greater than N1, a zero is returned.

NAME

inet_addr, inet_network, inet_ntoa, inet_makeaddr, inet_lnaof, inet_netof – Internet address manipulation routines

SYNOPSIS

```
#include </bsd/sys/socket.h>
#include </bsd/netinet/in.h>
#include </bsd/arpa/inet.h>

unsigned long inet_addr(cp)
char *cp;

unsigned long inet_network(cp)
char *cp;

char *inet_ntoa(in)
struct in_addr in;

struct in_addr inet_makeaddr(net, lna)
int net, lna;

int inet_lnaof(in)
struct in_addr in;

int inet_netof(in)
struct in_addr in;
```

DESCRIPTION

The routines *inet_addr* and *inet_network* each interpret character strings representing numbers expressed in the Internet standard "." notation, returning numbers suitable for use as Internet addresses and Internet network numbers, respectively. The routine *inet_ntoa* takes an Internet address and returns an ASCII string representing the address in "." notation. The routine *inet_makeaddr* takes an Internet network number and a local network address and constructs an Internet address from it. The routines *inet_netof* and *inet_lnaof* break apart Internet host addresses, returning the network number and local network address part, respectively.

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

INTERNET ADDRESSES

Values specified using the "." 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 "128.net.host".

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

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

All numbers supplied as "parts" in a "." notation may be decimal, octal, or hexadecimal, as specified in the C language (i.e., a leading 0x or 0X implies hexadecimal; otherwise, a leading 0 implies octal; otherwise, the number is interpreted as decimal).

SEE ALSO

gethostbyname(3N), getnetent(3N), hosts(4), networks(4).

DIAGNOSTICS

The value -1 is returned by *inet_addr* and *inet_network* for malformed requests.

ERRORS

The problem of host byte ordering versus network byte ordering is confusing. A simple way to specify Class C network addresses in a manner similar to that for Class B and Class A is needed. The string returned by *inet_ntoa* resides in a static memory area.

Inet_addr should return a struct *in_addr.*

ORIGIN

4.3 BSD

NAME

intro – introduction to functions and libraries

DESCRIPTION

This section describes functions found in various libraries, other than those functions that directly invoke UNIX system primitives, which are described in Section 2 of this volume. Certain major collections are identified by a letter after the section number:

(3C) These functions, together with those of Section 2 and those marked (3S), constitute the Standard C Library *libc*, which is automatically loaded by the C compiler, *cc*(1). (For this reason the (3C) and (3S) sections together comprise one section of this manual.) The link editor *ld*(1) searches this library under the **–lc** option. Declarations for some of these functions may be obtained from **#include** files indicated on the appropriate pages.

(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 may be obtained from the **#include** file **<stdio.h>**.

(3M) These functions constitute the Math Library, *libm*. They are automatically loaded as needed by the FORTRAN compiler *f77*(1). They are not automatically loaded by the C compiler, *cc*(1); however, the link editor searches this library under the **–lm** option. Declarations for these functions may be obtained from the **#include** file **<math.h>**. Several generally useful mathematical constants are also defined there [see *math*(5)].

(3X) Various specialized libraries. The files in which these libraries are found are given on the appropriate pages.

(3F) These functions constitute the FORTRAN intrinsic function library, *libF77*. These functions are automatically available to the FORTRAN programmer and require no special invocation of the compiler.

(3B) Berkeley compatibility routines. This library provides compatible implementations of a *limited* subset of the functions provided by the Standard C Library in Berkeley 4.x Distribution of UNIX. Include files needed for routines in this library are in the tree */usr/include/bsd*. It is recommended that the **-I/usr/include/bsd** compiler control be supplied when compiling programs that call (3B) routines. This library is searched by the link editor when the **-lbsd** and **-lsum** flags are supplied.

DEFINITIONS

A *character* is any bit pattern able to fit into a byte on the machine. 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 this value will not match that of any legitimate pointer, so many functions that return pointers return it to indicate an error. **NULL** is defined as **0** in **<stdio.h>**; the user can include an appropriate definition if not using **<stdio.h>**.

Many groups of FORTRAN intrinsic functions have *generic* function names that do not require explicit or implicit type declaration. The type of the function will be determined by the type of its argument(s). For example, the generic function *max* will return 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*).

FILES

LIBDIR usually /lib
LIBDIR/libc.a
LIBDIR/libm.a
LIBDIR/lib77.a

SEE ALSO

ar(1), cc(1), ld(1), lint(1), nm(1), stdio(3S), math(5).
f77(1) in the *FORTRAN Programming Language Manual.*

DIAGNOSTICS

Function in the C and Math libraries (3C and 3M) may return $\pm\infty$ or *NaN* (Not-a-Number) when the function is undefined for the given arguments or when the value is not representable.

WARNING

Many of the functions in the libraries call and/or refer to other functions and external variables described in this section and in Section 2 (*System Calls*). If a program inadvertently defines a function or external variable with the same name, the presumed library version of the function or external variable may not be loaded. 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, **-l***m* includes definitions for Section 3M, the Math Library.) Use of *lint* is highly recommended.

NAME

intro – introduction to FORTRAN library functions

DESCRIPTION

This section describes functions that are in the Fortran runtime library.

The math intrinsics required by the 1977 Fortran standard are available, although not described here. In addition, the *abs*, *sqrt*, *exp*, *log*, *sin*, and *cos* intrinsics have been extended for double complex values. They can be referenced using the generic names listed above, or they can be referenced using their specific names that consist of the generic names preceded by either *cd* or *z*. For example, if *zz* is double complex, then *sqrt(zz)*, *zsqrt(zz)*, or *cdsqrt(zz)* compute the square root of *zz*. The *dcmplx* intrinsic forms a double complex value from two double precision variables or expressions, and the name of the specific function for the conjugate of a double complex value is *dconjg*.

Most of these functions are in libU77.a. Some are in libF77.a or libI77.a.

For efficiency, the SCCS ID strings are not normally included in the *a.out* file. To include them, simply declare

```
external f77lid
```

in any *f77* module.

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
abort	abort.3f	abnormal termination
access	access.3f	determine accessibility of a file
alarm	alarm.3f	execute a subroutine after a specified time
chdir	chdir.3f	change default directory
chmod	chmod.3f	change mode of a file
ctime	time.3f	return system time
dtime	etime.3f	return elapsed execution time
etime	etime.3f	return elapsed execution time
fdate	fdate.3f	return date and time in an ASCII string
fgetc	getc.3f	get a character from a logical unit
flush	flush.3f	flush output to a logical unit
fork	fork.3f	create a copy of this process
fputc	putc.3f	write a character to a fortran logical unit
fseek	fseek.3f	reposition a file on a logical unit
fstat	stat.3f	get file status
ftell	fseek.3f	reposition a file on a logical unit
gerror	perror.3f	get system error messages
getarg	getarg.3f	return command line arguments
getc	getc.3f	get a character from a logical unit
getcwd	getcwd.3f	get pathname of current working directory
getenv	getenv.3f	get value of environment variables
getgid	getuid.3f	get user or group ID of the caller
getlog	getlog.3f	get user's login name
getpid	getpid.3f	get process id
getuid	getuid.3f	get user or group ID of the caller
gmtime	time.3f	return system time
iargc	getarg.3f	return command line arguments
idate	idate.3f	return date or time in numerical form
ierrno	perror.3f	get system error messages

irand	rand.3f	return random values
isatty	ttynam.3f	find name of a terminal port
itime	idate.3f	return date or time in numerical form
kill	kill.3f	send a signal to a process
len	len.3f	tell about character objects
link	link.3f	make a link to an existing file
loc	loc.3f	return the address of an object
ltime	time.3f	return system time
perror	perror.3f	get system error messages
putc	putc.3f	write a character to a fortran logical unit
qsort	qsort.3f	quick sort
rand	rand.3f	return random values
signal	signal.3f	change the action for a signal
sleep	sleep.3f	suspend execution for an interval
stat	stat.3f	get file status
system	system.3f	execute a UNIX command
time	time.3f	return system time
ttynam	ttynam.3f	find name of a terminal port
unlink	unlink.3f	remove a directory entry
wait	wait.3f	wait for a process to terminate

NAME

isnan: isnand, isnanf – test for floating point NaN (Not-A-Number)

SYNOPSIS

```
#include <ieeefp.h>

int isnand (dsrc)
double dsrc;

int isnanf (fsrc)
float fsrc;
```

DESCRIPTION

isnand and isnanf return true (1) if the argument dsrc or fsrc is a NaN; otherwise they return false (0).

Neither routine generates any exception, even for signaling NaNs.

isnanf() is implemented as a macro included in <ieeefp.h>.

SEE ALSO

fpgetround(3C).

NAME

j0, j1, jn, y0, y1, yn – bessel functions

SYNOPSIS

```
#include <math.h>

double j0(x)
double x;

double j1(x)
double x;

double jn(n, x)
int n;
double x;

double y0(x)
double x;

double y1(x)
double x;

double yn(n, x)
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 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.

DIAGNOSTICS

Non-positive arguments cause *y0*, *y1* and *yn* to return a quiet NaN.

BUGS

Arguments too large in magnitude cause *j0*, *j1*, *y0* and *y1* to return zero with no indication of the total loss of precision.

SEE ALSO

math(3M)

NAME

kill – send a signal to a process

SYNOPSIS

function kill (pid, signum)
integer pid, signum

DESCRIPTION

Pid must be the process id of one of the user's processes. *Signum* must be a valid signal number (see sigvec(2)). The returned value will be 0 if successful; an error code otherwise.

FILES

/usr/lib/libU77.a

SEE ALSO

kill(2), sigvec(2), signal(3F), fork(3F), perror(3F)

NAME

l3tol, ltol3 – convert between 3-byte integers and long integers

SYNOPSIS

```
void l3tol (lp, cp, n)
long *lp;
char *cp;
int n;

void ltol3 (cp, lp, n)
char *cp;
long *lp;
int n;
```

DESCRIPTION

l3tol converts 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 performs 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).

CAVEAT

Because of possible differences in byte ordering, the numerical values of the long integers are machine-dependent.

NAME

ldahread – read the archive header of a member of an archive file

SYNOPSIS

```
#include <stdio.h>
#include <ar.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldahread (ldptr, arhead)
LDFILE *ldptr;
ARCHDR *arhead;
```

DESCRIPTION

If **TYPE**(*ldptr*) is the archive file magic number, *ldahread* reads the archive header of the common object file currently associated with *ldptr* into the area of memory beginning at *arhead.*

Ldahread returns **SUCCESS** or **FAILURE**. If **TYPE**(*ldptr*) does not represent an archive file or if it cannot read the archive header, *Ldahread* fails.

The program must be loaded with the object file access routine library **libmld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ar(4), ldfcn(4), and intro(4).

NAME

ldclose, ldaclose – close a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldclose (ldptr)
LDFILE *ldptr;

int ldaclose (ldptr)
LDFILE *ldptr;
```

DESCRIPTION

Ldopen(3X) and *ldclose* provide uniform access to simple object files and object files that are members of archive files. An archive of common object files can be processed as if it is a series of simple common object files.

If **TYPE**(*ldptr*) does not represent an archive file, *ldclose* closes the file and frees the memory allocated to the **LDFILE** structure associated with *ldptr*. If **TYPE**(*ldptr*) is the magic number for an archive file and if archive has more files, *ldclose* reinitializes **OFFSET**(*ldptr*) to the file address of the next archive member and returns **FAILURE**. The **LDFILE** structure is prepared for a later *ldopen*(3X). In all other cases, *ldclose* returns **SUCCESS**.

Ldaclose closes the file and frees the memory allocated to the **LDFILE** structure associated with *ldptr* regardless of the value of **TYPE**(*ldptr*). *Ldaclose* always returns **SUCCESS**. The function is often used with *ldaopen*.

The program must be loaded with the object file access routine library **libmld.a**.

SEE ALSO

fclose(3S), ldopen(3X), ldfcn(4).

NAME

ldfhread – read the file header of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldfhread (ldptr, filehead)
LDFILE *ldptr;
FILHDR *filehead;
```

DESCRIPTION

Ldfhread reads the file header of the common object file currently associated with *ldptr* . It reads the file header into the area of memory beginning at *filehead*.

Ldfhread returns **SUCCESS** or **FAILURE**. If *ldfhread* cannot read the file header, it fails.

Usually, *ldfhread* can be avoided by using the macro **HEADER**(*ldptr*) defined in **<ldfcn.h>** (see *ldfcn*(4)). Note that the information in HEADER is swapped, if necessary. The information in any field, *fieldname*, of the file header can be accessed using **HEADER**(*ldptr*)*.fieldname*.

The program must be loaded with the object file access routine library **libmld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldfcn(4).

NAME

ldgetaux – retrieve an auxiliary entry, given an index

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <sym.h>
#include <ldfcn.h>

pAUXU ldgetaux (ldptr, iaux)
LDFILE ldptr;
long iaux;
```

DESCRIPTION

Ldgetaux returns a pointer to an auxiliary table entry associated with *iaux*. The AUXU is contained in a static buffer. Because the buffer can be overwritten by later calls to *ldgetaux*, it must be copied by the caller if the aux is to be saved or changed.

Note that auxiliary entries are not swapped as this routine cannot detect what manifestation of the AUXU union is retrieved. If LDAUXSWAP(ldptr, ldf) is non-zero, a further call to *swap_aux* is required. Before calling the *swap_aux* routine, the caller should copy the aux.

If the auxiliary cannot be retrieved, *Ldgetaux* returns **NULL** (defined in **<stdio.h>**) for an object file. This occurs when:

- the auxiliary table cannot be found
- the iaux offset into the auxiliary table is beyond the end of the table

Typically, *ldgetaux* is called immediately after a successful call to *ldtbread* to retrieve the data type information associated with the symbol table entry filled by *ldtbread*. The index field of the symbol, pSYMR, is the *iaux* when data type information is required. If the data type information for a symbol is not present, the index field is *indexNil* and ldgetaux should not be called.

The program must be loaded with the object file access routine library *libmld.a*.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbseek(3X), ldtbread(3X), ldfcn(4).

NAME

ldgetname – retrieve symbol name for object file symbol table entry

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <sym.h>
#include <ldfcn.h>

char *ldgetname (ldptr, symbol)
LDFILE * ldptr ;
pSYMR * symbol ;
```

DESCRIPTION

Ldgetname returns a pointer to the name associated with *symbol* as a string. The string is contained in a static buffer. Because the buffer can be overwritten by later calls to *ldgetname*, the caller must copy the buffer if the name is to be saved.

If the name cannot be retrieved, *ldgetname* returns **NULL** (defined in **<stdio.h>**) for an object file. This occurs when:

- the string table cannot be found
- the name's offset into the string table is beyond the end of the string table

Typically, *ldgetname* is called immediately after a successful call to *ldtbread*. *Ldgetname* retrieves the name associated with the symbol table entry filled by *ldtbread*.

The program must be loaded with the object file access routine library **libmld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbseek(3X), ldtbread(3X), ldfcn(4).

NAME

ldgetpd – retrieve procedure descriptor given a procedure descriptor index

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <sym.h>
#include <ldfcn.h>

long ldgetpd (ldptr, ipd, ppd)
LDFILE ldptr;
long ipd;
pPDR ipd;
```

DESCRIPTION

Ldgetpd returns a SUCCESS or FAILURE depending on whether the procedure descriptor with index *ipd* can be accessed. If it can be accessed, the structure pointed to by *ppd* is filled with the contents of the corresponding procedure descriptor. The *isym, iline,* and *iopt* fields of the procedure descriptor are updated to be used in further LD routine calls. The *adr* field is updated from the symbol referenced by the *isym field.*

The PDR cannot be retrieved when:

- The procedure descriptor table cannot be found.
- The ipd offset into the procedure descriptor table is beyond the end of the table.
- The file descriptor that the ipd offset falls into cannot be found.

Typically, *ldgetpd* is called while traversing the table that runs from 0 to SYMHEADER(ldptr).ipdMax - 1.

The program must be loaded with the object file access routine library **libmld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbseek(3X), ldtbread(3X), ldfcn(4).

NAME

ldlread, ldlinit, ldlitem – manipulate line number entries of a common object file function

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldlread (ldptr, fcnindx, linenum, linent)
LDFILE *ldptr;
long fcnindx;
unsigned short linenum;
LINER linent;

int ldlinit (ldptr, fcnindx)
LDFILE *ldptr;
long fcnindx;

int ldlitem (ldptr, linenum, linent)
LDFILE *ldptr;
unsigned short linenum;
LINER linent;
```

DESCRIPTION

Ldlread searches the line number entries of the common object file currently associated with *ldptr*. *Ldlread* begins its search with the line number entry for the beginning of a function and confines its search to the line numbers associated with a single function. The function is identified by *fcnindx*, which is the index of its local symbols entry in the object file symbol table. *Ldlread* reads the entry with the smallest line number equal to or greater than *linenum* into *linent*.

Ldlinit and *ldlitem* together do exactly the same function as *ldlread*. After an initial call to *ldlread* or *ldlinit, ldlitem* can be used to retrieve a series of line number entries associated with a single function. *Ldlinit* simply finds the line number entries for the function identified by *fcnindx*. *Ldlitem* finds and reads the entry with the smallest line number equal to or greater than *linenum* into *linent*.

Ldlread, *ldlinit*, and *ldlitem* each return either **SUCCESS** or **FAILURE**. If no line number entries exist in the object file, if *fcnindx* does not index a function entry in the symbol table, or if it finds no line number equal to or greater than *linenum*, *ldlread* fails. If no line number entries exist in the object file or if *fcnindx* does not index a function entry in the symbol table, *ldlinit* fails. If it finds no line number equal to or greater than *linenum*, *ldlitem* fails.

The programs must be loaded with the object file access routine library **libmld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbindex(3X), ldfcn(4).

NAME

ldlseek, ldnlseek – seek to line number entries of a section of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldlseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnlseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

DESCRIPTION

Ldlseek seeks to the line number entries of the section specified by *sectindx* of the common object file currently associated with *ldptr*.

Ldnlseek seeks to the line number entries of the section specified by *sectname*.

Ldlseek and *ldnlseek* return **SUCCESS** or **FAILURE**. **NOTE:** Line numbers are not associated with sections in the MIPS symbol table; therefore, the second argument is ignored, but maintained for historical purposes.

If they cannot seek to the specified line number entries, both routines fail.

The program must be loaded with the object file access routine library **libmld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4).

NAME

ldohseek – seek to the optional file header of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldohseek (ldptr)
LDFILE *ldptr;
```

DESCRIPTION

Ldohseek seeks to the optional file header of the common object file currently associated with *ldptr*.

Ldohseek returns **SUCCESS** or **FAILURE**. If the object file has no optional header or if it cannot seek to the optional header, *ldohseek* fails.

The program must be loaded with the object file access routine library **libmld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldfhread(3X), ldfcn(4).

NAME

ldopen, ldaopen – open a common object file for reading

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

LDFILE *ldopen (filename, ldptr)
char *filename;
LDFILE *ldptr;

LDFILE *ldaopen (filename, oldptr)
char *filename;
LDFILE *oldptr;

ld readst (ldptr, flags)
LDFILE *ldptr;
intflags;
```

DESCRIPTION

Ldopen and *ldclose*(3X) provide uniform access to simple object files and to object files that are members of archive files. An archive of common object files can be processed as if it were a series of simple common object files.

If *ldptr* has the value **NULL**, *ldopen* opens *filename*, allocates and initializes the **LDFILE** structure, and returns a pointer to the structure to the calling program.

If *ldptr* is valid and **TYPE**(*ldptr*) is the archive magic number, *ldopen* reinitializes the **LDFILE** structure for the next archive member of *filename*.

Ldopen and *ldclose* work in concert. *Ldclose* returns **FAILURE** only when **TYPE**(*ldptr*) is the archive magic number and there is another file in the archive to be processed. Only then should *ldopen* be called with the current value of *ldptr*. In all other cases, and particularly when a new *filename* is opened, *ldopen* should be called with a **NULL** *ldptr* argument.

The following is a prototype for the use of *ldopen* and *ldclose:*

```
/* for each filename to be processed */
ldptr = NULL;
do
        if ( (ldptr = ldopen(filename, ldptr)) != NULL )

        {
                /* check magic number */
                /* process the file */
        }
} while (ldclose(ldptr) == FAILURE );
```

If the value of *oldptr* is not **NULL**, *ldaopen* opens *filename* anew and allocates and initializes a new **LDFILE** structure, copying the fields from *oldptr*. *Ldaopen* returns a pointer to the new **LDFILE** structure. This new pointer is independent of the old pointer, *oldptr*. The two pointers can be used concurrently to read separate parts of the object file. For example, one pointer can be used to step sequentially through the relocation information while the other is used to read indexed symbol table entries.

Ldopen and *ldaopen* open *filename* for reading. If *filename* cannot be opened or if memory for the **LDFILE** structure cannot be allocated, both functions return **NULL**. A successful open does not ensure that the given file is a common object file or an archived object file.

Ldopen causes the symbol table header and file descriptor table to be read. Further access, using *ldptr*, causes other appropriate sections of the symbol table to be read (for example, if you call *ldtbread, the symbols or externals are read). To force sections fot eh symbol table in memory, call ldreadst* with *ST_P** constants ORed together from *st_support.h*.

The program must be loaded with the object file access routine library **libmld.a**.

SEE ALSO

fopen(3S), ldclose(3X), ldfcn(4).

NAME

ldrseek, ldnrseek – seek to relocation entries of a section of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldrseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnrseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

DESCRIPTION

Ldrseek seeks to the relocation entries of the section specified by *sectindx* of the common object file currently associated with *ldptr*.

Ldnrseek seeks to the relocation entries of the section specified by *sectname*.

Ldrseek and *ldnrseek* return **SUCCESS** or **FAILURE**. If *sectindx* is greater than the number of sections in the object file, *ldrseek* fails; if there is no section name corresponding with *sectname, ldnrseek* fails. If the specified section has no relocation entries or if it cannot seek to the specified relocation entries, either function fails.

NOTE: The first section has an index of *one*.

The program must be loaded with the object file access routine library **libmld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4).

NAME

ldshread, ldnshread – read an indexed/named section header of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <scnhdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldshread (ldptr, sectindx, secthead)
LDFILE *ldptr;
unsigned short sectindx;
SCNHDR *secthead;

int ldnshread (ldptr, sectname, secthead)
LDFILE *ldptr;
char *sectname;
SCNHDR *secthead;
```

DESCRIPTION

Ldshread reads the section header specified by *sectindx* of the common object file currently associated with *ldptr* into the area of memory beginning at *secthead*.

Ldnshread reads the section header specified by *sectname* into the area of memory beginning at *secthead*.

Ldshread and *ldnshread* return **SUCCESS** or **FAILURE**. If *sectindx* is greater than the number of sections in the object file, *ldshread* fails; If there is no section name corresponding with *sectname, ldnshread* fails. If it cannot read the specified section header, either function fails.

NOTE: The first section header has an index of *one*.

The program must be loaded with the object file access routine library **libmld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldfcn(4).

NAME

ldsseek, ldnsseek – seek to an indexed/named section of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldsseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnsseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

DESCRIPTION

Ldsseek seeks to the section specified by *sectindx* of the common object file currently associated with *ldptr*.

Ldnsseek seeks to the section specified by *sectname*.

Ldsseek and *ldnsseek* return **SUCCESS** or **FAILURE**. If *sectindx* is greater than the number of sections in the object file, *ldsseek* fails; if there is no section name corresponding with *sectname, ldnsseek* fails. If there is no section data for the specified section or if it cannot seek to the specified section, either function fails.

NOTE: The first section has an index of *one*.

The program must be loaded with the object file access routine library **libmld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldshread(3X), ldfcn(4).

NAME

ldtbindex – compute the index of a symbol table entry of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

long ldtbindex (ldptr)
LDFILE *ldptr;
```

DESCRIPTION

ldtbindex returns the (**long**) index of the symbol table entry at the current position of the common object file associated with *ldptr*.

The index returned by *ldtbindex* can be used in later calls to *ldtbread*(3X). *ldtbindex* returns the index of the symbol table entry that begins at the current position of the object file; therefore, if *ldtbindex* is called immediately after a particular symbol table entry has been read, it returns the the index of the next entry.

If there are no symbols in the object file or if the object file is not positioned at the beginning of a symbol table entry, *ldtbindex* fails and returns BADINDEX (-1).

NOTE: The first symbol in the symbol table has an index of *zero*.

The program must be loaded with the object file access routine library **libmld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbread(3X), ldtbseek(3X), ldfcn(5).

NAME

ldtbread – read an indexed symbol table entry of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldtbread (ldptr, symindex, symbol)
LDFILE *ldptr;
long symindex;
pSYMR *symbol;
```

DESCRIPTION

Ldtbread reads the symbol table entry specified by *symindex* of the common object file currently associated with *ldptr* into the area of memory beginning at *symbol*.

Ldtbread returns **SUCCESS** or **FAILURE**. If *symindex* is greater than the number of symbols in the object file or if it cannot read the specified symbol table entry, *ldtbread* fails.

The local and external symbols are concatenated into a linear list. Symbols are accessible from symnum zero to *SYMHEADER(ldptr).isymMax+SYMHEADER(ldptr).iextMax*. The index and iss fields of the SYMR are made absolute (rather than file relative) so that routines ldgetname(3X), ldgetaux(3X), and ldtbread (this routine) proceed normally given those indices. Only the "sym" part of externals is returned.

NOTE: The first symbol in the symbol table has an index of *zero*.

The program must be loaded with the object file access routine library **libmld.a**.

SEE ALSO

ldclose(3X), ldgetname(3X), ldopen(3X), ldtbseek(3X), ldgetname(3X), ldfcn(4).

NAME

ldtbseek – seek to the symbol table of a common object file

SYNOPSIS

```
#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

int ldtbseek (ldptr)
LDFILE *ldptr;
```

DESCRIPTION

Ldtbseek seeks to the symbol table of the object file currently associated with *ldptr*.

Ldtbseek returns **SUCCESS** or **FAILURE**. If the symbol table has been stripped from the object file or if it cannot seek to the symbol table, *ldtbseek* fails.

The program must be loaded with the object file access routine library **libld.a**.

SEE ALSO

ldclose(3X), ldopen(3X), ldtbread(3X), ldfcn(4).

NAME

len – return length of Fortran string

SYNOPSIS

```
character*B ch
integer i
i = len(ch)
```

DESCRIPTION

len returns the length of string *ch*.

NAME

lgamma – log gamma function

SYNOPSIS

#include <math.h>

double lgamma(x)
double x;

DESCRIPTION

Lgamma returns $\ln |\Gamma(x)|$ where $\Gamma(x) = \int_0^\infty t^{x-1} e^{-t}\, dt$ for $x > 0$ and $\Gamma(x) = \pi/(\Gamma(1-x)\sin(\pi x))$ for $x < 1$.

The external integer signgam returns the sign of $\Gamma(x)$.

IDIOSYNCRASIES

Do **not** use the expression signgam*exp(lgamma(x)) to compute g := $\Gamma(x)$. Instead use a program like this (in C):

```
lg = lgamma(x); g = signgam*exp(lg);
```

Only after lgamma has returned can signgam be correct. Note too that $\Gamma(x)$ must overflow when x is large enough, underflow when −x is large enough, and spawn a division by zero when x is a nonpositive integer.

The following C program fragment might be used to calculate Γ if the overflow needs to be detected:

```
if ((y = lgamma(x)) > LN_MAXDOUBLE)
error( );
y = signgam * exp(y);
```

where LN_MAXDOUBLE is the least value that causes *exp*(3M) to overflow, and is defined in the *<values.h>* header file.

Only in the UNIX math library for C was the name gamma ever attached to $\ln\Gamma$. Elsewhere, for instance in IBM's FORTRAN library, the name GAMMA belongs to Γ and the name ALGAMA to $\ln\Gamma$ in single precision; in double the names are DGAMMA and DLGAMA. Why should C be different?

Archaeological records suggest that C's gamma originally delivered $\ln(\Gamma(|x|))$. Later, the program gamma was changed to cope with negative arguments x in a more conventional way, but the documentation did not reflect that change correctly. The most recent change corrects inaccurate values when x is almost a negative integer, and lets $\Gamma(x)$ be computed without conditional expressions. Programmers should not assume that lgamma has settled down.

At some time in the future, the name *gamma* will be rehabilitated and used for the gamma function, just as is done in FORTRAN. The reason for this is not so much compatibility with FORTRAN as a desire to achieve greater speed for smaller values of $|x|$ and greater accuracy for larger values.

Meanwhile, programmers who have to use the name *gamma* in its former sense, for what is now *lgamma*, have two choices:

1) Change your source to use *lgamma* instead of *gamma*.

2) Add the following program to your others:

```
#include <math.h>
double gamma(x)
double x;
{
        return (lgamma(x));
}
```

DIAGNOSTICS

Γ returns $+\infty$ for negative integer arguments.

SEE ALSO

math(3M)

NAME

VADS libraries – overview of VADS libraries

DESCRIPTION

VADS includes libraries containing packages and functions that may be referenced by user applications and a directory of examples using them.

Libraries contained in the current release of the VADS are listed below. The exact contents varies with each implementation.

standard – predefined Ada packages and additional packages to implement them

verdixlib – Verdix-supplied packages

*publiclib** – public domain packages written in Ada

*examples** – sample Ada program files

**Note: publiclib* and *examples*
are neither supported nor warranted by VERDIX.

NAME

link – make a link to an existing file

SYNOPSIS

function link (name1, name2)
character*(*) name1, name2

integer function symlnk (name1, name2)
character*(*) name1, name2

DESCRIPTION

Name1 must be the pathname of an existing file. *Name2* is a pathname to be linked to file *name1*. *Name2* must not already exist. The returned value will be 0 if successful; a system error code otherwise.

Symlnk creates a symbolic link to *name1*.

FILES

/usr/lib/libU77.a

SEE ALSO

link(2), symlink(2), perror(3F), unlink(3F)

BUGS

Pathnames can be no longer than MAXPATHLEN as defined in *<sys/param.h>*.

NAME

loc – return the address of an object

SYNOPSIS

function loc (arg)

DESCRIPTION

The returned value will be the address of *arg*.

FILES

/usr/lib/libU77.a

NAME

lockf – record locking on files

SYNOPSIS

#include <unistd.h>

int lockf (fildes, function, size)
long size;
int fildes, function;

DESCRIPTION

The *lockf* command will allow sections of a file to be locked; advisory or mandatory write locks depending on the mode bits of the file [see *chmod*(2)]. Locking calls from other processes which attempt to lock the locked file section will either return an error value or be put to sleep until the resource becomes unlocked. All the locks for a process are removed when the process terminates. [See *fcntl*(2) for more information about record locking.]

fildes is an open file descriptor. The file descriptor must have O_WRONLY or O_RDWR permission in order to establish lock with this function call.

function is a control value which specifies the action to be taken. The permissible values for *function* are defined in **<unistd.h>** as follows:

```
#define   F_ULOCK   0   /* Unlock a previously locked section */
#define   F_LOCK    1   /* Lock a section for exclusive use */
#define   F_TLOCK   2   /* Test and lock a section for exclusive use */
#define   F_TEST    3   /* Test section for other processes locks */
```

All other values of *function* are reserved for future extensions and will result in an error return if not implemented.

F_TEST is used to detect if a lock by another process is present on the specified section. F_LOCK and F_TLOCK both lock a section of a file if the section is available. F_ULOCK removes locks from a section 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 section from the current offset through the largest file offset is locked (i.e., 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 as such locks may exist past the end-of-file.

The sections locked with F_LOCK or F_TLOCK may, in whole or in part, contain or be contained by a previously locked section for the same process. When this occurs, or if adjacent sections occur, the sections are combined into a single section. If the request requires that a new element be added to the table of active locks and this table is already full, an error is returned, and the new section is not locked.

F_LOCK and F_TLOCK requests differ only by the action taken if the resource is not available. F_LOCK will cause the calling process to sleep until the resource is available. F_TLOCK will cause the function to return a –1 and set *errno* to [EACCES] error if the section is already locked by another process.

F_ULOCK requests may, in whole or in part, release one or more locked sections controlled by the process. When sections are not fully released, the remaining sections are still locked by the process. Releasing the center section of a locked section requires an additional element in the table of active locks. If this table is full, an [EDEADLK] error is returned and the requested section is not released.

A potential for deadlock occurs if a process controlling a locked resource is put to sleep by accessing another process's locked resource. Thus calls to *lockf* or *fcntl* scan for a deadlock prior to sleeping on a locked resource. An error return is made if sleeping on the locked resource would cause a deadlock.

Sleeping on a resource is interrupted with any signal. The *alarm*(2) command may be used to provide a timeout facility in applications which require this facility.

The *lockf* utility will fail if one or more of the following are true:

[EBADF]	*fildes* is not a valid open descriptor.
[EACCES]	*cmd* is F_TLOCK or F_TEST and the section is already locked by another process.
[EDEADLK]	*cmd* is F_LOCK and a deadlock would occur. Also the *cmd* is either F_LOCK, F_TLOCK, or F_ULOCK and the number of entries in the lock table would exceed the number allocated on the system.
[ECOMM]	*fildes* is on a remote machine and the link to that machine is no longer active.

SEE ALSO

chmod(2), close(2), creat(2), fcntl(2), intro(2), open(2), read(2), write(2).

DIAGNOSTICS

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

Unexpected results may occur in processes that do buffering in the user address space. The process may later read/write data which is/was locked. The standard I/O package is the most common source of unexpected buffering.

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.

NAME

log, alog, dlog, clog – Fortran natural logarithm intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
complex cx1, cx2
r2 = alog(r1)
r2 = log(r1)
dp2 = dlog(dp1)
dp2 = log(dp1)
cx2 = clog(cx1)
cx2 = log(cx1)
```

DESCRIPTION

alog returns the real natural logarithm of its real argument. *dlog* returns the double-precision natural logarithm of its double-precision argument. *clog* returns the complex logarithm of its complex argument. The generic function *log* becomes a call to *alog*, *dlog*, or *clog* depending on the type of its argument.

SEE ALSO

exp(3M).

NAME

log10, alog10, dlog10 - Fortran common logarithm intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2
r2 = alog10(r1)
r2 = log10(r1)
dp2 = dlog10(dp1)
dp2 = log10(dp1)

DESCRIPTION

alog10 returns the real common logarithm of its real argument. *dlog10* returns the double-precision common logarithm of its double-precision argument. The generic function *log10* becomes a call to *alog10* or *dlog10* depending on the type of its argument.

SEE ALSO

exp(3M).

NAME

logname – return login name of user

SYNOPSIS

char *logname( )

DESCRIPTION

logname returns a pointer to the null-terminated login name; it extracts the **LOGNAME** environment variable from the user's environment.

This routine is kept in **/usr/lib/lib**PW.a.

FILES

/etc/profile

SEE ALSO

getenv(3C), profile(4), environ(5).
env(1), login(1) in the *User's Reference Manual.*

CAVEATS

The return values point to static data whose content is overwritten by each call. This method of determining a login name is subject to forgery.

NAME

lsearch, lfind – linear search and update

SYNOPSIS

```
#include <stdio.h>
#include <search.h>

char *lsearch ((char *)key, (char *)base, nelp, sizeof(*key), compar)
unsigned *nelp;
int (*compar)( );

char *lfind ((char *)key, (char *)base, nelp, sizeof(*key), compar)
unsigned *nelp;
int (*compar)( );
```

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** is the 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 is the 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.

EXAMPLE

This fragment will read in less than TABSIZE strings of length less than ELSIZE and store them in a table, eliminating duplicates.

```
#include <stdio.h>
#include <search.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), string(3C), tsearch(3C).

DIAGNOSTICS

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.

ERRORS

Undefined results can occur if there is not enough room in the table to add a new item.

NAME

machine_info – get information about the running system

SYNOPSIS

#include <machine_info.h>
int machine_info (command)
int command;

DESCRIPTION

machine_info is an interface for getting information about the current system. It is intended to be used by commands to determine what type or class of system is running, since it is possible to run the same object code on all types of Mips hardware.

The **command** argument specifies which information is requested, such as the type of the machine or the class of the machine. See the *machine_info.h* include file for the currently-supported set of commands.

The return value is the information requested. If the command is no supported, the return value is -1. In all other cases, the value will be as described by definitions found in the include file or be an integer value as appropriate (e.g., in the future, there may be a command to obtain the number of megabytes of system memory).

Information that has been requested is saved as static data so that multiple calls to *machine_info* simply return the data.

As the system evolves, with new system calls and other means of getting dynamic system information, *machine_info* will present a consistent interface for obtaining certain useful items, instead of requiring programmers to deal with more system calls with various interfaces.

SEE ALSO

uname(2).

NAME

malloc, free, realloc, calloc – main memory allocator

SYNOPSIS

```
char *malloc (size)
unsigned size;

void free (ptr)
char *ptr;

char *realloc (ptr, size)
char *ptr;
unsigned size;

char *calloc (nelem, elsize)
unsigned nelem, elsize;
```

DESCRIPTION

malloc and *free* provide a simple general-purpose memory allocation package. *malloc* returns a pointer to a block of at least *size* bytes suitably aligned for any use.

The argument to *free* is a pointer to a block previously allocated by *malloc*; after *free* is performed this space is made available for further allocation, but its contents are left undisturbed.

Undefined results will occur if the space assigned by *malloc* is overrun or if some random number is handed to *free*.

malloc allocates the first big enough contiguous reach of free space found in a circular search from the last block allocated or freed, coalescing adjacent free blocks as it searches. It calls *sbrk* [see *brk*(2)] to get more memory from the system when there is no suitable space already free.

realloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. If no free block of *size* bytes is available in the storage arena, then *realloc* will ask *malloc* to enlarge the arena by *size* bytes and will then move the data to the new space.

realloc also works if *ptr* points to a block freed since the last call of *malloc*, *realloc*, or *calloc*; thus sequences of *free*, *malloc* and *realloc* can exploit the search strategy of *malloc* to do storage compaction.

calloc allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

SEE ALSO

brk(2), malloc(3X).

DIAGNOSTICS

malloc, *realloc* and *calloc* return a NULL pointer if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block. When this happens the block pointed to by *ptr* may be destroyed.

NOTES

Search time increases when many objects have been allocated; that is, if a program allocates but never frees, then each successive allocation takes longer. For an alternate, more flexible implementation, see *malloc*(3X).

NAME

malloc, free, realloc, calloc, mallopt, mallinfo – fast main memory allocator

SYNOPSIS

```
#include <malloc.h>
char *malloc (size)
unsigned size;

void free (ptr)
char *ptr;

char *realloc (ptr, size)
char *ptr;
unsigned size;

char *calloc (nelem, elsize)
unsigned nelem, elsize;

int mallopt (cmd, value)
int cmd, value;

struct mallinfo mallinfo()
```

DESCRIPTION

malloc and *free* provide a simple general-purpose memory allocation package, which runs considerably faster than the *malloc*(3C) package. It is found in the library "malloc", and is loaded if the option "–lmalloc" is used with *cc*(1) or *ld*(1).

malloc returns a pointer to a block of at least *size* bytes suitably aligned for any use.

The argument to *free* is a pointer to a block previously allocated by *malloc*; after *free* is performed this space is made available for further allocation, and its contents have been destroyed (but see *mallopt* below for a way to change this behavior).

Undefined results will occur if the space assigned by *malloc* is overrun or if some random number is handed to *free*.

realloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes.

calloc allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

mallopt provides for control over the allocation algorithm. 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 and then doles them out very quickly. The default value for *maxfast* is 24.
M_NLBLKS	Set *numlblks* to *value*. The above mentioned "large groups" each contain *numlblks* blocks. *numlblks* must be greater than 0. 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 0. The default value of *grain* is the smallest number of bytes which will allow alignment of any data type. Value will be 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.

These values are defined in the <*malloc.h*> header file.

mallopt may be called repeatedly, but may not be called after the first small block is allocated.

mallinfo provides instrumentation describing space usage. 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.

SEE ALSO

brk(2), malloc(3C).

DIAGNOSTICS

malloc, realloc and *calloc* return a NULL pointer if there is not enough available memory. When *realloc* returns NULL, the block pointed to by *ptr* is left intact. If *mallopt* is called after any allocation or if *cmd* or *value* are invalid, non-zero is returned. Otherwise, it returns zero.

WARNINGS

This package usually uses more data space than *malloc*(3C).
The code size is also bigger than *malloc*(3C).
Note that unlike *malloc*(3C), this package does not preserve the contents of a block when it is freed, unless the M_KEEP option of *mallopt* is used.
Undocumented features of *malloc*(3C) have not been duplicated.

NAME

math – introduction to mathematical library functions

DESCRIPTION

These functions constitute the C math library *libm*. There are two versions of the math library *libm.a* and *libm43.a*.

The first, *libm.a,* contains routines written in MIPS assembly language and tuned for best performance and includes many routines for the *float* data type. The routines in there are based on the algorithms of Cody and Waite or those in the 4.3 BSD release, whichever provides the best performance with acceptable error bounds. Those routines with Cody and Waite implementations are marked with a '*' in the list of functions below.

The second version of the math library, *libm43.a,* contains routines all based on the original codes in the 4.3 BSD release. The difference between the two version's error bounds is typically around 1 unit in the last place, whereas the performance difference may be a factor of two or more.

The link editor searches this library under the "–lm" (or "–lm43") option. Declarations for these functions may be obtained from the include file *<math.h>*. The Fortran math library is described in "man 3f intro".

LIST OF FUNCTIONS

The cycle counts of all functions are approximate; cycle counts often depend on the value of argument. The error bound sometimes applies only to the primary range.

			Error Bound (ULPs)		*Cycles*	
Name	*Appears on Page*	*Description*	*libm.a*	*libm43.a*	*libm.a*	*libm43.a*
acos	sin.3m	inverse trigonometric function	3	3	?	?
acosh	asinh.3m	inverse hyperbolic function	3	3	?	?
asin	sin.3m	inverse trigonometric function	3	3	?	?
asinh	asinh.3m	inverse hyperbolic function	3	3	?	?
atan	sin.3m	inverse trigonometric function	1	1	152	260
atanh	asinh.3m	inverse hyperbolic function	3	3	?	?
atan2	sin.3m	inverse trigonometric function	2	2	?	?
cabs	hypot.3m	complex absolute value	1	1	?	?
cbrt	sqrt.3m	cube root	1	1	?	?
ceil	floor.3m	integer no less than	0	0	?	?
copysign	ieee.3m	copy sign bit	0	0	?	?
cos*	sin.3m	trigonometric function	2	1	128	243
cosh*	sinh.3m	hyperbolic function	?	3	142	294
drem	ieee.3m	remainder	0	0	?	?
erf	erf.3m	error function	?	?	?	?
erfc	erf.3m	complementary error function	?	?	?	?
exp*	exp.3m	exponential	2	1	101	230
expm1	exp.3m	exp(x)–1	1	1	281	281
fabs	floor.3m	absolute value	0	0	?	?
fatan*	sin.3m	inverse trigonometric function	3		64	
fcos*	sin.3m	trigonometric function	1		87	
fcosh*	sinh.3m	hyperbolic function	?		105	
fexp*	exp.3m	exponential	1		79	
flog*	exp.3m	natural logarithm	1		100	
floor	floor.3m	integer no greater than	0	0	?	?
fsin*	sin.3m	trigonometric function	1		68	

fsinh*	sinh.3m	hyperbolic function	?		44	
fsqrt	sqrt.3m	square root	1		95	
ftan*	sin.3m	trigonometric function	?		61	
ftanh*	sinh.3m	hyperbolic function	?		116	
hypot	hypot.3m	Euclidean distance	1	1	?	?
j0	j0.3m	bessel function	?	?	?	?
j1	j0.3m	bessel function	?	?	?	?
jn	j0.3m	bessel function	?	?	?	?
lgamma	lgamma.3m	log gamma function	?	?	?	?
log*	exp.3m	natural logarithm	2	1	119	217
logb	ieee.3m	exponent extraction	0	0	?	?
log10*	exp.3m	logarithm to base 10	3	3	?	?
log1p	exp.3m	log(1+x)	1	1	269	269
pow	exp.3m	exponential x**y	60–500	60–500	?	?
rint	floor.3m	round to nearest integer	0	0	?	?
scalb	ieee.3m	exponent adjustment	0	0	?	?
sin*	sin.3m	trigonometric function	2	1	101	222
sinh*	sinh.3m	hyperbolic function	?	3	79	292
sqrt	sqrt.3m	square root	1	1	133	133
tan*	sin.3m	trigonometric function	?	3	92	287
tanh*	sinh.3m	hyperbolic function	?	3	156	293
y0	j0.3m	bessel function	?	?	?	?
y1	j0.3m	bessel function	?	?	?	?
yn	j0.3m	bessel function	?	?	?	?

NOTES

In 4.3 BSD, distributed from the University of California in late 1985, most of the foregoing functions come in two versions, one for the double–precision "D" format in the DEC VAX–11 family of computers, another for double–precision arithmetic conforming to the IEEE Standard 754 for Binary Floating–Point Arithmetic. The two versions behave very similarly, as should be expected from programs more accurate and robust than was the norm when UNIX was born. For instance, the programs are accurate to within the numbers of *ulps* tabulated above; an *ulp* is one *U*nit in the *L*ast *P*lace. And the programs have been cured of anomalies that afflicted the older math library *libm* in which incidents like the following had been reported:

sqrt(−1.0) = 0.0 and log(−1.0) = −1.7e38.
cos(1.0e−11) > cos(0.0) > 1.0.
pow(x,1.0) ≠ x when x = 2.0, 3.0, 4.0, ..., 9.0.
pow(−1.0,1.0e10) trapped on Integer Overflow.
sqrt(1.0e30) and sqrt(1.0e−30) were very slow.

MIPS machines conform to the IEEE Standard 754 for Binary Floating–Point Arithmetic, to which only the notes for IEEE floating-point apply and are included here.

IEEE STANDARD 754 Floating–Point Arithmetic:

This standard is on its way to becoming more widely adopted than any other design for computer arithmetic.

The main virtue of 4.3 BSD's *libm* codes is that they are intended for the public domain; they may be copied freely provided their provenance is always acknowledged, and provided users assist the authors in their researches by reporting experience with the codes. Therefore no user of UNIX on a machine that conforms to IEEE 754 need use anything worse than the new *libm*.

Properties of IEEE 754 Double-Precision:

Wordsize: 64 bits, 8 bytes. Radix: Binary.

Precision: 53 significant bits, roughly like 16 significant decimals.

If x and x' are consecutive positive Double-Precision numbers (they differ by 1 *ulp*), then

$1.1e{-}16 < 0.5{**}53 < (x'-x)/x \leq 0.5{**}52 < 2.3e{-}16$.

Range: Overflow threshold = 2.0**1024 = 1.8e308

Underflow threshold = 0.5**1022 = 2.2e-308

Overflow goes by default to a signed ∞.

Underflow is *Gradual,* rounding to the nearest integer multiple of 0.5**1074 = 4.9e-324.

Zero is represented ambiguously as +0 or -0.

Its sign transforms correctly through multiplication or division, and is preserved by addition of zeros with like signs; but x-x yields +0 for every finite x. The only operations that reveal zero's sign are division by zero and copysign(x,$\pm$0). In particular, comparison ($x > y$, $x \geq y$, etc.) cannot be affected by the sign of zero; but if finite $x = y$ then $\infty = 1/(x-y) \neq -1/(y-x) = -\infty$.

∞ is signed.

it persists when added to itself or to any finite number. Its sign transforms correctly through multiplication and division, and (finite)/$\pm\infty$ = $\pm$0 (nonzero)/0 = $\pm\infty$. But $\infty-\infty$, $\infty*0$ and ∞/∞ are, like 0/0 and sqrt(-3), invalid operations that produce *NaN*. ...

Reserved operands:

there are 2**53-2 of them, all called *NaN* (*N*ot *a N*umber). Some, called Signaling *NaN*s, trap any floating-point operation performed upon them; they could be used to mark missing or uninitialized values, or nonexistent elements of arrays. The rest are Quiet *NaN*s; they are the default results of Invalid Operations, and propagate through subsequent arithmetic operations. If $x \neq x$ then x is *NaN*; every other predicate ($x > y$, $x = y$, $x < y$, ...) is FALSE if *NaN* is involved.

NOTE: Trichotomy is violated by *NaN*.

Besides being FALSE, predicates that entail ordered comparison, rather than mere (in)equality, signal Invalid Operation when *NaN* is involved.

Rounding:

Every algebraic operation (+, -, *, /, $\sqrt{}$) is rounded by default to within half an *ulp*, and when the rounding error is exactly half an *ulp* then the rounded value's least significant bit is zero. This kind of rounding is usually the best kind, sometimes provably so; for instance, for every x = 1.0, 2.0, 3.0, 4.0, ..., 2.0**52, we find (x/3.0)*3.0 == x and (x/10.0)*10.0 == x and ... despite that both the quotients and the products have been rounded. Only rounding like IEEE 754 can do that. But no single kind of rounding can be proved best for every circumstance, so IEEE 754 provides rounding towards zero or towards $+\infty$ or towards $-\infty$ at the programmer's option. And the same kinds of rounding are specified for Binary-Decimal Conversions, at least for magnitudes between roughtly 1.0e-10 and 1.0e37.

Exceptions:

IEEE 754 recognizes five kinds of floating-point exceptions, listed below in declining order of probable importance.

Exception	Default Result
Invalid Operation	*NaN*, or FALSE
Overflow	$\pm\infty$
Divide by Zero	$\pm\infty$
Underflow	Gradual Underflow
Inexact	Rounded value

NOTE: An Exception is not an Error unless handled badly. What makes a class of exceptions exceptional is that no single default response can be satisfactory in every instance. On the other hand, if a default response will serve most instances satisfactorily, the unsatisfactory instances cannot justify aborting computation every time the exception occurs.

For each kind of floating-point exception, IEEE 754 provides a Flag that is raised each time its exception is signaled, and stays raised until the program resets it. Programs may also test, save and restore a flag. Thus, IEEE 754 provides three ways by which programs may cope with exceptions for which the default result might be unsatisfactory:

1) Test for a condition that might cause an exception later, and branch to avoid the exception.

2) Test a flag to see whether an exception has occurred since the program last reset its flag.

3) Test a result to see whether it is a value that only an exception could have produced.
CAUTION: The only reliable ways to discover whether Underflow has occurred are to test whether products or quotients lie closer to zero than the underflow threshold, or to test the Underflow flag. (Sums and differences cannot underflow in IEEE 754; if $x \neq y$ then $x-y$ is correct to full precision and certainly nonzero regardless of how tiny it may be.) Products and quotients that underflow gradually can lose accuracy gradually without vanishing, so comparing them with zero (as one might on a VAX) will not reveal the loss. Fortunately, if a gradually underflowed value is destined to be added to something bigger than the underflow threshold, as is almost always the case, digits lost to gradual underflow will not be missed because they would have been rounded off anyway. So gradual underflows are usually *provably* ignorable. The same cannot be said of underflows flushed to 0.

At the option of an implementor conforming to IEEE 754, other ways to cope with exceptions may be provided:

4) ABORT. This mechanism classifies an exception in advance as an incident to be handled by means traditionally associated with error-handling statements like "ON ERROR GO TO ...". Different languages offer different forms of this statement, but most share the following characteristics:

- No means is provided to substitute a value for the offending operation's result and resume computation from what may be the middle of an expression. An exceptional result is abandoned.

- In a subprogram that lacks an error-handling statement, an exception causes the subprogram to abort within whatever program called it, and so on back up the

chain of calling subprograms until an error-handling statement is encountered or the whole task is aborted and memory is dumped.

5) STOP. This mechanism, requiring an interactive debugging environment, is more for the programmer than the program. It classifies an exception in advance as a symptom of a programmer's error; the exception suspends execution as near as it can to the offending operation so that the programmer can look around to see how it happened. Quite often the first several exceptions turn out to be quite unexceptionable, so the programmer ought ideally to be able to resume execution after each one as if execution had not been stopped.

6) ... Other ways lie beyond the scope of this document.

The crucial problem for exception handling is the problem of Scope, and the problem's solution is understood, but not enough manpower was available to implement it fully in time to be distributed in 4.3 BSD's *libm*. Ideally, each elementary function should act as if it were indivisible, or atomic, in the sense that ...

i) No exception should be signaled that is not deserved by the data supplied to that function.

ii) Any exception signaled should be identified with that function rather than with one of its subroutines.

iii) The internal behavior of an atomic function should not be disrupted when a calling program changes from one to another of the five or so ways of handling exceptions listed above, although the definition of the function may be correlated intentionally with exception handling.

Ideally, every programmer should be able *conveniently* to turn a debugged subprogram into one that appears atomic to its users. But simulating all three characteristics of an atomic function is still a tedious affair, entailing hosts of tests and saves-restores; work is under way to ameliorate the inconvenience.

Meanwhile, the functions in *libm* are only approximately atomic. They signal no inappropriate exception except possibly ...

Over/Underflow
when a result, if properly computed, might have lain barely within range, and

Inexact in *cabs*, *cbrt*, *hypot*, *log10* and *pow*
when it happens to be exact, thanks to fortuitous cancellation of errors.

Otherwise, ...

Invalid Operation is signaled only when
any result but *NaN* would probably be misleading.

Overflow is signaled only when
the exact result would be finite but beyond the overflow threshold.

Divide-by-Zero is signaled only when
a function takes exactly infinite values at finite operands.

Underflow is signaled only when
the exact result would be nonzero but tinier than the underflow threshold.

Inexact is signaled only when
greater range or precision would be needed to represent the exact result.

Exceptions on MIPS machines:

The exception enables and the flags that are raised when an exception occurs (as well as the rounding mode) are in the floating-point control and status register. This register can be read or written by the routines described on the man page *fpc*(3). This register's layout is described in the file *<mips/fpu.h>* in UMIPS-BSD releases and in

<sys/fpu.h> in UMIPS-SYSV releases.

A full implementation of IEEE 754 "user trap handlers" is under development at MIPS computer systems. At which time all functions in *libm* will appear atomic and the full functionality of user trap handlers will be supported in thoses language without other floating-point error handling intrinsics (i.e. ADA, Pl/1, etc). For a description of these trap handlers see section 8 of the IEEE 754 standard.

What is currently available is only the raw interface which was only intended to be used by the code to implement IEEE user trap handlers. IEEE floating-point exceptions are enabled by setting the enable bit for that exception in the floating-point control and status register. If an exception then occurs the UNIX signal SIGFPE is sent to the process. It is up to the signal handler to determine the instruction that caused the exception and to take the action specified by the user. The instruction that caused the exception is in one of two places. If the floating-point board is used (the floating-point implementation revision register indicates this in it's implementation field) then the instruction that caused the exception is in the floating-point exception instruction register. In all other implementations the instruction that caused the exception is at the address of the program counter as modified by the branch delay bit in the cause register. Both the program counter and cause register are in the sigcontext structure passed to the signal handler (see *signal*(3)). If the program is to be continued past the instruction that caused the exception the program counter in the signal context must be advanced. If the instruction is in a branch delay slot then the branch must be emulated to determine if the branch is taken and then the resulting program counter can be calculated (see *emulate_branch*(3) and the **NOTES (MIPS)** section in *signal*(3)).

BUGS

When signals are appropriate, they are emitted by certain operations within the codes, so a subroutine-trace may be needed to identify the function with its signal in case method 5) above is in use. And the codes all take the IEEE 754 defaults for granted; this means that a decision to trap all divisions by zero could disrupt a code that would otherwise get correct results despite division by zero.

SEE ALSO

fpc(3), signal(3), emulate_branch(3)
R2010 Floating Point Coprocessor Architecture
R2360 Floating Point Board Product Description
An explanation of IEEE 754 and its proposed extension p854 was published in the IEEE magazine MICRO in August 1984 under the title "A Proposed Radix- and Word-length-independent Standard for Floating-point Arithmetic" by W. J. Cody et al. Articles in the IEEE magazine COMPUTER vol. 14 no. 3 (Mar. 1981), and in the ACM SIGNUM Newsletter Special Issue of Oct. 1979, may be helpful although they pertain to superseded drafts of the standard.

AUTHOR

W. Kahan, with the help of Z-S. Alex Liu, Stuart I. McDonald, Dr. Kwok-Choi Ng, Peter Tang.

NAME

max, max0, amax0, max1, amax1, dmax1 - Fortran maximum-value functions

SYNOPSIS

```
integer i, j, k, l
real a, b, c, d
double precision dp1, dp2, dp3
l = max(i, j, k)
c = max(a, b)
dp = max(a, b, c)
k = max0(i, j)
a = amax0(i, j, k)
i = max1(a, b)
d = amax1(a, b, c)
dp3 = dmax1(dp1, dp2)
```

DESCRIPTION

The maximum-value functions return the largest of their arguments (of which there may be any number). *max* is the generic form which can be used for all data types and takes its return type from that of its arguments (which must all be of the same type). *max0* returns the integer form of the maximum value of its integer arguments; *amax0*, the real form of its integer arguments; *max1*, the integer form of its real arguments; *amax1*, the real form of its real arguments; and *dmax1*, the double-precision form of its double-precision arguments.

SEE ALSO

min(3F).

NAME

mclock – return Fortran time accounting

SYNOPSIS

integer i i = mclock( )

DESCRIPTION

mclock returns time accounting information about the current process and its child processes. The value returned is the sum of the current process's user time and the user and system times of all child processes.

SEE ALSO

times(2), clock(3C), system(3F).

NAME

memory: memccpy, memchr, memcmp, memcpy, memset – memory operations

SYNOPSIS

```
#include <memory.h>

char *memccpy (s1, s2, c, n)
char *s1, *s2;
int c, n;

char *memchr (s, c, n)
char *s;
int c, n;

int memcmp (s1, s2, n)
char *s1, *s2;
int n;

char *memcpy (s1, s2, n)
char *s1, *s2;
int n;

char *memset (s, c, n)
char *s;
int c, n;
```

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.

memccpy copies characters from memory area **s2** into **s1**, stopping after the first occurrence of character **c** has been copied, or after **n** characters have been copied, whichever comes first. It 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 returns a pointer to the first occurrence of character **c** in the first **n** characters of memory area **s**, or a NULL pointer if **c** does not occur.

memcmp compares its arguments, looking at the first **n** characters only, and returns an integer less than, equal to, or greater than 0, according as **s1** is lexicographically less than, equal to, or greater than **s2**.

memcpy copies **n** characters from memory area **s2** to **s1**. It returns **s1**.

memset sets the first **n** characters in memory area **s** to the value of character **c**. It returns **s**.

For user convenience, all these functions are declared in the optional *<memory.h>* header file.

CAVEATS

memcmp is implemented by using the most natural character comparison on the machine. Thus the sign of the value returned when one of the characters has its high order bit set is not the same in all implementations and should not be relied upon.

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

NAME

mil: ior, iand, not, ieor, ishft, ishftc, ibits, btest, ibset, ibclr, mvbits - Fortran Military Standard functions

SYNOPSIS

```
integer i, k, l, m, n, len
logical b

i = ior(m, n)
i = iand(m, n)
i = not(m)
i = ieor(m, n)
i = ishft(m, k)
i = ishftc(m, k, len)
i = ibits(m, k, len)
b = btest(n, k)
i = ibset(n, k)
i = ibclr(n, k)
call mvbits(m, k, len, n, l)
```

DESCRIPTION

mil is the general name for the bit field manipulation intrinsic functions and subroutines from the Fortran Military Standard (MIL-STD-1753). *ior, iand, not, ieor* - return the same results as *and, or, not, xor* as defined in *bool*(3F).

ishft, ishftc - **m** specifies the integer to be shifted. **k** specifies the shift count. **k** > 0 indicates a left shift. **k** = 0 indicates no shift. **k** < 0 indicates a right shift. In *ishft*, zeros are shifted in. In *ishftc*, the rightmost **len** bits are shifted circularly **k** bits. If **k** is greater than the machine word-size, *ishftc* will not shift.

Bit fields are numbered from right to left and the rightmost bit position is zero. The length of the **len** field must be greater than zero.

ibits - extract a subfield of **len** bits from **m** starting with bit position **k** and extending left for **len** bits. The result field is right justified and the remaining bits are set to zero.

btest - The **k**th bit of argument **n** is tested. The value of the function is **.TRUE.** if the bit is a 1 and **.FALSE.** if the bit is 0.

ibset - the result is the value of **n** with the **k**th bit set to 1.

ibclr - the result is the value of **n** with the **k**th bit set to 0.

mvbits - **len** bits are moved beginning at position **k** of argument **m** to position **l** of argument **n**.

SEE ALSO

bool(3F).

NAME

min, min0, amin0, min1, amin1, dmin1 – Fortran minimum-value functions

SYNOPSIS

```
integer i, j, k, l
real a, b, c, d
double precision dp1, dp2, dp3
l = min(i, j, k)
c = min(a, b)
dp = min(a, b, c)
k = min0(i, j)
a = amin0(i, j, k)
i = min1(a, b)
d = amin1(a, b, c)
dp3 = dmin1(dp1, dp2)
```

DESCRIPTION

The minimum-value functions return the minimum of their arguments (of which there may be any number). *min* is the generic form which can be used for all data types and takes its return type from that of its arguments (which must all be of the same type). *min0* returns the integer form of the minimum value of its integer arguments; *amin0*, the real form of its integer arguments; *min1*, the integer form of its real arguments; *amin1*, the real form of its real arguments; and *dmin1*, the double-precision form of its double-precision arguments.

SEE ALSO

max(3F).

NAME

mktemp – make a unique file name

SYNOPSIS

char *mktemp (template)
char *template;

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* will replace the **X**s with a letter and the current process ID. The letter will be chosen so that the resulting name does not duplicate an existing file.

SEE ALSO

getpid(2), tmpfile(3S), tmpnam(3S).

DIAGNOSTIC

mktemp will assign to *template* the NULL string if it cannot create a unique name.

CAVEAT

If called more than 17,576 time in a single process, this function will start recycling previously used names.

NAME

mod, amod, dmod – Fortran remaindering intrinsic functions

SYNOPSIS

```
integer i, j, k
real r1, r2, r3
double precision dp1, dp2, dp3
k = mod(i, j)
r3 = amod(r1, r2)
r3 = mod(r1, r2)
dp3 = dmod(dp1, dp2)
dp3 = mod(dp1, dp2)
```

DESCRIPTION

mod returns the integer remainder of its first argument divided by its second argument. *amod* and *dmod* return, respectively, the real and double-precision whole number remainder of the integer division of their two arguments. The generic version *mod* will return the data type of its arguments.

NAME

monitor, monstartup, moncontrol – prepare execution profile

SYNOPSIS

monitor(lowpc, highpc, buffer, bufsize, nfunc)
int (*lowpc)(), (*highpc)();
short buffer[];

monstartup(lowpc, highpc)
int (*lowpc)(), (*highpc)();

moncontrol(mode)

DESCRIPTION

These functions use the *profil(2)* system call to control program-counter sampling. Using the option **–p** when compiling or linking a program (see *The MIPS Languages Programmer Guide*) automatically generates calls to these functions. You need not call them explicitly unless you want finer control.

Typically, you would call either **monitor** or *monstartup* to initialize pc-sampling and enable it; call *moncontrol* to disable or reenable it; and call *monitor* again at the end of execution to disable sampling and record the samples in a file.

Your initial call to *monitor* enables pc-sampling. *lowpc* and *highpc* specify the range of addresses to be sampled; the lowest address is that of *lowpc* and the highest is just below *highpc*. *buffer* is the address of a (user allocated) array of *bufsize* short integers, which holds a record of the samples; for best results, the buffer should not be less than a few times smaller than the range of addresses sampled. *nfunc* is ignored.

The environment variable PROFDIR determines the name of the output file and whether pc-sampling takes place: if it is not set, the file is named "mon.out"; if set to the empty string, no pc-sampling occurs; if set to a non-empty string, the file is named "string/pid.progname", where "pid" is the process id of the executing program and "progname" is the program's name as it appears in argv[0]. The subdirectory "string" must already exist.

To profile the entire program, use:

```
extern eprol(), etext();
. . .
monitor(eprol, etext, buf, bufsize, 0);
```

eprol lies just below the user program text, and *etext* lies just above it, as described in *end(3)*. (Because the user program does not necessarily start at a low memory address, using a small number in place of "eprol" is dangerous).

monstartup is an alternate form of *monitor* that calls *sbrk* (see *brk(2)*) for you to allocate the buffer.

moncontrol selectively disables and re-enables pc-sampling within a program, allowing you to measure the cost of particular operations. *moncontrol(0)* disables pc-sampling, and *moncontrol(1)* reenables it.

To stop execution monitoring and write the results in the output file, use:

```
monitor(0);
```

FILES

mon.out	default name for output file
libprof1.a	routines for pc-sampling

SEE ALSO

profil(2), brk(2).
cc(1), ld(1) in the *User's Reference Manual.*
The MIPS Languages Programmer Guide.

NAME

mount – keep track of remotely mounted filesystems

SYNOPSIS

#include <rpcsvc/mount.h>

RPC INFO

program number:

MOUNTPROG

xdr routines:

```
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;
};
```

SEE ALSO

mount(1M), showmount(1M), mountd(1M).

ORIGIN

Sun Microsystems

NAME

nlist – get entries from name list

SYNOPSIS

#include <nlist.h>

nlist(filename, nl)
char *filename;
struct nlist nl[];

cc ... -lmld

DESCRIPTION

NOTE: The *nlist* subroutine has moved from the standard C library to the "mld" library due to the difference in the object file format. Programs that need to use *nlist* must be linked with the **-lmld** option.

Nlist examines the name list in the given executable output file and selectively extracts a list of values. The name list consists of an array of structures containing names, types and values. The list is terminated with a null name. Each name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted in the next two fields. If the name is not found, both entries are set to 0. For the structure declaration, see */usr/include/nlist.h*.

This subroutine is useful for examining the system name list kept in the file **/vmunix**. In this way programs can obtain system addresses that are up to date.

SEE ALSO

a.out(5)

DIAGNOSTICS

If the file cannot be found or if it is not a valid namelist -1 is returned; otherwise, the number of unfound namelist entries is returned.

The type entry is set to 0 if the symbol is not found.

NAME

perror, errno, sys_errlist, sys_nerr – system error messages

SYNOPSIS

```
void perror (s)
char *s;

extern int errno;

extern char *sys_errlist[ ];

extern int sys_nerr;

extern char *sys_errnolist[ ];

extern int sys_nerrno;
```

DESCRIPTION

perror produces a message on the standard error output, describing the last error encountered during a call to a system or library function.

By default, the message printed consists of the text given by the argument to *perror*, followed by a colon and a space if the text is non-empty, followed by the system message corresponding to the error number. 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. All error messages end with a newline.

The environment variable PERROR_FMT can be set to a non-empty string containing the format of the error message. Text from the string is copied as-is except for a set of %-specifiers described below:

%p	The standard message as described above.
%t	The argument to *perror*.
%c	A colon (:) if the argument to *perror* is nonempty; otherwise an empty string.
%s	A colon (:) followed by a space if the argument to *perror* is nonempty; otherwise an empty string.
%m	The system message that corresponds to the error number.
%e	The symbolic name for the error number.
%n	The error number.
%%	The character %.

A % followed by any other character causes the % and the character to be printed.

As an example, assume that *perror* is called after a failed call to *open(2)* that sets the error number to 2, and that the argument to *perror* is "myfile". If PERROR_FMT is "%t %s%m - (%e)", the resulting message will be

```
myfile : No such file or directory - (ENOENT)
```

To simplify variant formatting of messages, the array of message strings *sys_errlist* and the array is provided; *errno* can be used as an index into these tables to get the strings without the new-line. *sys_nerr* and *sys_nerrno* are the number of messages in the tables; they should be checked because new error codes may be added to the system before they are added to the tables. Note that in future releases of this system, a routine will be provided to return the formatted message without the newline, so that programs that need to format messages can do so.

SEE ALSO

intro(2).

ERRORS

Many programs do not use *perror* so the formatting is not always useful. These programs should be fixed.

NAME

perror, gerror, ierrno – get system error messages

SYNOPSIS

subroutine perror (string)
character*(*) string

subroutine gerror (string)
character*(*) string

character*(*) function gerror()

function ierrno()

DESCRIPTION

Perror will write a message to fortran logical unit 0 appropriate to the last detected system error. *String* will be written preceding the standard error message.

Gerror returns the system error message in character variable *string*. *Gerror* may be called either as a subroutine or as a function.

Ierrno will return the error number of the last detected system error. This number is updated only when an error actually occurs. Most routines and I/O statements that might generate such errors return an error code after the call; that value is a more reliable indicator of what caused the error condition.

FILES

/usr/lib/libU77.a

SEE ALSO

intro(2), perror(3)
D. L. Wasley, *Introduction to the f77 I/O Library*

BUGS

String in the call to *perror* can be no longer than 127 characters.

The length of the string returned by *gerror* is determined by the calling program.

NOTES

UNIX system error codes are described in *intro*(2). The f77 I/O error codes and their meanings are:

100	"error in format"
101	"illegal unit number"
102	"formatted i/o not allowed"
103	"unformatted i/o not allowed"
104	"direct i/o not allowed"
105	"sequential i/o not allowed"
106	"can't backspace file"
107	"off beginning of record"
108	"can't stat file"
109	"no * after repeat count"
110	"off end of record"
111	"truncation failed"
112	"incomprehensible list input"
113	"out of free space"
114	"unit not connected"
115	"invalid data for integer format term"

116 "invalid data for logical format term"
117 "'new' file exists"
118 "can't find 'old' file"
119 "opening too many files or unknown system error"
120 "requires seek ability"
121 "illegal argument"
122 "negative repeat count"
123 "illegal operation for unit"
124 "invalid data for d, e, f, or g format term"

NAME

popen, pclose – initiate pipe to/from a process

SYNOPSIS

#include <stdio.h>

FILE *popen (command, type)
char *command, *type;

int pclose (stream)
FILE *stream;

DESCRIPTION

popen creates a pipe between the calling program and the command to be executed. The arguments to *popen* are pointers to null-terminated strings. *command* consists of a shell command line. *type* is an I/O mode, either **r** for reading or **w** for writing. The value returned is 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 may be used as an input filter and a type **w** as an output filter.

EXAMPLE

A typical call may be:

```
char *cmd = "ls *.c";
FILE *ptr;
if ((ptr = popen(cmd, "r")) != NULL)
    while (fgets(buf, n, ptr) != NULL)
        (void) printf("%s ",buf);
```

This will print in *stdout* [see *stdio* (3S)] all the file names in the current directory that have a ".c" suffix.

SEE ALSO

pipe(2), wait(2), fclose(3S), fopen(3S), stdio(3S), system(3S).

DIAGNOSTICS

popen returns a NULL pointer if files or processes cannot be created.

pclose returns –1 if *stream* is not associated with a "*popen*ed" command.

WARNING

If the original and "*popen*ed" processes concurrently read or write a common file, neither should use buffered I/O, because the buffering gets all mixed up. Problems with an output filter may be forestalled by careful buffer flushing, e.g. with *fflush* [see *fclose*(3S)].

NAME

printf, fprintf, sprintf – print formatted output

SYNOPSIS

```
#include <stdio.h>

int printf (format , arg . . . )
char *format;

int fprintf (stream, format , arg . . . )
FILE *stream;
char *format;

int sprintf (s, format [ , arg ] . . . )
char *s, *format;
```

DESCRIPTION

printf places output on the standard output stream *stdout*. *fprintf* places output on the named output *stream*. *sprintf* places "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 returns the number of characters transmitted (not including the **\0** in the case of *sprintf*), or a negative value if an output error was encountered.

Each of these functions converts, formats, and prints its *arg*s under control of the *format*. The *format* is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in fetching of 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, the excess *arg*s are simply ignored.

Each conversion specification is introduced by the character **%**. After the **%**, the following appear in sequence:

Zero or more *flags*, which modify the meaning of the conversion specification.

An optional decimal digit string specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment flag '–', described below, has been given) to the field width. The padding is with blanks unless the field width digit string starts with a zero, in which case the padding is with zeros.

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 decimal point for the **e**, **E**, and **f** conversions, the maximum number of significant digits for the **g** and **G** conversion, or the maximum number of characters to be printed from a string in **s** conversion. The precision takes the form of a period (**.**) followed by a decimal digit string; a null digit string is treated as zero. Padding specified by the precision overrides the padding specified by the field width.

An optional **l** (ell) specifying that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion character applies to a long integer *arg*. An **l** before any other conversion character is ignored.

A character that indicates the type of conversion to be applied.

A field width or precision or both may 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 must appear *before* the *arg* (if any) to be converted. A negative field width argument is taken as a **'–'** flag followed by a positive field width. If the precision argument is negative, it will be changed to zero.

The flag characters and their meanings are:

– The result of the conversion will be left-justified within the field.

+ The result of a signed conversion will always begin with a sign (**+** or **–**).

blank If the first character of a signed conversion is not a sign, a blank will be prefixed to the result. This implies that if the blank and **+** flags both appear, the blank flag will be ignored.

This flag specifies that the value is to be converted to an "alternate form." For **c**, **d**, **i**, **s**, 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 will have **0x or 0X** prefixed to it. For **e**, **E**, **f**, **g**, and **G** conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For **g** and **G** conversions, trailing zeroes will *not* be removed from the result (which they normally are).

The conversion characters and their meanings are:

d,i,o,u,x,X The integer *arg* is converted to signed decimal (**d** or **i**), unsigned octal, (**o**), decimal (**u**), or hexadecimal notation (**x** or **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 will be 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.

f The float or double *arg* is converted to decimal notation in the style "[**–**]ddd.ddd," where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, six digits are output; if the precision is explicitly 0, no decimal point appears.

e,E The float or double *arg* is converted in the style "[**–**]d.ddd**e**±dd," where there is one digit before the decimal point 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 decimal point appears. The **E** format code will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits.

g,G The float or 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** will be used only if the exponent resulting from the conversion is less than –4 or greater than the precision. Trailing zeroes are removed from the result; a decimal point appears only if it is followed by a digit.

c The character *arg* is printed.

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 characters 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* will yield undefined results.

%	Print a %; no argument is converted.

In printing floating point types (float and double), if the exponent is 0x7FF and the mantissa is not equal to zero, then the output is

[-]NaN0xdddddddd

where 0xdddddddd is the hexadecimal representation of the leftmost 32 bits of the mantissa. If the mantissa is zero, the output is

[±]inf.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by *printf* and *fprintf* are printed as if *putc*(3S) had been called.

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 %i, %d:%.2d", weekday, month, day, hour, min);
```

To print π to 5 decimal places:

```
printf("pi = %.5f", 4 * atan(1.0));
```

SEE ALSO

ecvt(3C), putc(3S), scanf(3S), stdio(3S).

publiclib – public domain packages written in Ada

DESCRIPTION

publiclib contains the packages CHARACTER_TYPE and VSTRINGS.

NOTE: These packages are neither supported by nor warranteed by MIPS.

CHARACTER_TYPE provided the following character handling functions.
ISLAPHA
ISUPPER
ISLOWER
ISDIGIT
ISXDIGIT
ISALNUM
ISSPACE
ISPUNCT
ISPRINT
ISCNTRL
ISASCII
TOUPPER
TOLOWER
TOASCII

VSTRINGS provides string replacement, searching, concatenation, and other string functions with a simple syntac and the ability to transfer data between its own data representation and the predefined Ada type STRING.

TYPES AND FUNCTIONS

subtype ASCII_INTEGER in TOASCII function

FILES

*/usr/vads5/publiclib/**

SEE ALSO

examples, standard, verdixlib

NAME

putc, fputc – write a character to a fortran logical unit

SYNOPSIS

integer function putc (char)
character char

integer function fputc (lunit, char)
character char

DESCRIPTION

These funtions write a character to the file associated with a fortran logical unit bypassing normal fortran I/O. *Putc* writes to logical unit 6, normally connected to the control terminal output.

The value of each function will be zero unless some error occurred; a system error code otherwise. See perror(3F).

FILES

/usr/lib/libU77.a

SEE ALSO

putc(3S), intro(2), perror(3F)

NAME

putc, putchar, fputc, putw – put character or word on a stream

SYNOPSIS

```
#include <stdio.h>

int putc (c, stream)
int c;
FILE *stream;

int putchar (c)
int c;

int fputc (c, stream)
int c;
FILE *stream;

int putw (w, stream)
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 macros.

fputc behaves like *putc*, but is a function rather than a macro. *fputc* runs more slowly than *putc*, but it takes less space per invocation and its name can be passed as an argument to a function.

putw writes the word (i.e. integer) *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.

SEE ALSO

fclose(3S), ferror(3S), fopen(3S), fread(3S), printf(3S), puts(3S), setbuf(3S), stdio(3S).

DIAGNOSTICS

On success, these functions (with the exception of *putw*) each return the value they have written. [*Putw* returns *ferror (stream)*]. On failure, they return the constant **EOF**. This will occur if the file *stream* is not open for writing or if the output file cannot grow. Because **EOF** is a valid integer, *ferror*(3S) should be used to detect *putw* errors.

CAVEATS

Because it is implemented as a macro, *putc* evaluates a *stream* argument more than once. In particular, **putc(c, *f++);** doesn't work sensibly. *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 read using *getw* on a different processor.

NAME

putenv – change or add value to environment

SYNOPSIS

int putenv (string)
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 will change the environment. The space used by *string* is no longer used once a new string-defining *name* is passed to *putenv*.

SEE ALSO

exec(2), getenv(3C), malloc(3C), environ(5).

DIAGNOSTICS

putenv returns non-zero if it was unable to obtain enough space via *malloc* for an expanded environment, otherwise 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*(3C) to enlarge the environment.
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.

NAME

putpwent – write password file entry

SYNOPSIS

```
#include <pwd.h>

int putpwent (p, f)
struct passwd *p;
FILE *f;
```

DESCRIPTION

putpwent is the inverse of *getpwent*(3C). Given a pointer to a passwd structure created by *getpwent* (or *getpwuid* or *getpwnam*), *putpwent* writes a line on the stream *f*, which matches the format of **/etc/passwd**.

SEE ALSO

getpwent(3C).

DIAGNOSTICS

putpwent returns non-zero if an error was detected during its operation, otherwise zero.

WARNING

The above routine uses **<stdio.h>**, which causes it to increase the size of programs, not otherwise using standard I/O, more than might be expected.

NAME

puts, fputs – put a string on a stream

SYNOPSIS

```
#include <stdio.h>

int puts (s)
char *s;

int fputs (s, stream)
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*.

Neither function writes the terminating null character.

SEE ALSO

ferror(3S), fopen(3S), fread(3S), printf(3S), putc(3S), stdio(3S).

DIAGNOSTICS

Both routines return **EOF** on error. This will happen if the routines try to write on a file that has not been opened for writing.

NOTES

puts appends a new-line character while *fputs* does not.

NAME

qsort – quicker sort

SYNOPSIS

void qsort ((char *) base, nel, sizeof (*base), compar)
unsigned nel;
int (*compar)( );

DESCRIPTION

qsort is an implementation of the quicker-sort algorithm. It sorts a table of data in place.

base points to the element at the base of the table. *nel* is the number of elements in the table. *compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. As the function must return an integer less than, equal to, or greater than zero, so must the first argument to be considered be less than, equal to, or greater than the second.

NOTES

The pointer to 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.

The order in the output of two items which compare as equal is unpredictable.

SEE ALSO

bsearch(3C), lsearch(3C), string(3C).
sort(1) in the *User's Reference Manual*.

NAME

qsort – quick sort

SYNOPSIS

subroutine qsort (array, len, isize, compar)
external compar
integer[*2] compar

DESCRIPTION

One dimensional *array* contains the elements to be sorted. *len* is the number of elements in the array. *isize* is the size of an element, typically -

4 for **integer** and **real**
8 for **double precision** or **complex**
16 for **double complex**
(length of character object) for **character** arrays

Compar is the name of a user supplied integer or integer*2 function that will determine the sorting order. You must declare *compar* as external with the "external" statement to be recognized as a function. This function will be called with 2 arguments that will be elements of *array*. The function must return -

negative if arg 1 is considered to precede arg 2
zero if arg 1 is equivalent to arg 2
positive if arg 1 is considered to follow arg 2

On return, the elements of *array* will be sorted.

FILES

/usr/lib/libU77.a

SEE ALSO

qsort(3)

NAME

rand, srand – simple random-number generator

SYNOPSIS

int rand ( )

void srand (seed)
unsigned 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.

NOTES

The spectral properties of *rand* are limited. *drand48(3C)* provides a much better, though more elaborate, random-number generator.

NAME

rand, irand, srand – random number generator

SYNOPSIS

```
integer iseed, i, irand
double precision s, rand

call srand(iseed)

i = irand( )

x = rand( )
```

DESCRIPTION

Irand generates successive pseudo-random integers in the range from 0 to 2**15–1. *rand* generates pseudo-random numbers distributed in [-, 1.0]. *Srand* uses its integer argument to reinitialize the seed for successive invocations of *irand* and *rand*.

SEE ALSO

rand(3C).

NAME

ranhashinit, ranhash, ranlookup – access routine for the symbol table definition file in archives

SYNOPSIS

```
#include <ar.h>

int ranhashinit(pran, pstr, size)
struct ranlib *pran;
char *pstr;
int size;

ranhash(name)
char *name;

struct ranlib *ranhash(name)
char *name;
```

DESCRIPTION

Ranhashinit initializes static information for future use by *ranhash* and *ranlookup*. *Pran* points to an array of ranlib structures. *Pstr* points to the corresponding ranlib string table (these are only used by *ranlookup)*. *Size* is the size of the hash table and should be a power of 2. If the size isn't a power of 2, a 1 is returned; otherwise, a 0 is returned.

Ranhash returns a hash number given a name. It uses a multiplicative hashing algorithm and the *size* argument to *ranhashinit*.

Ranlookup looks up *name* in the ranlib table specified by *ranhashinit*. It uses the *ranhash* routine as a starting point. Then, it does a rehash from there. This routine returns a pointer to a valid ranlib entry on a match. If no matches are found (the "emptiness" can be inferred if the ran_off field is zero), the empty ranlib structure hash table should be sparse. This routine does not expect to run out of places to look in the table. For example, if you collide on all entries in the table, an error is printed tostderr and a zero is returned.

AUTHOR

Mark I. Himelstein

SEE ALSO

ar(1), ar.h(5).

NAME

rcmd, rresvport, ruserok – routines for returning a stream to a remote command

SYNOPSIS

```
rem = rcmd(ahost, inport, locuser, remuser, cmd, fd2p);
char **ahost;
int inport;
char *locuser, *remuser, *cmd;
int *fd2p;

s = rresvport(options);
int options;

ruserok(rhost, ruser, luser);
char *rhost;
char *user, *luser;
```

DESCRIPTION

rcmd executes a command on a remote machine. It uses an authentication scheme based on reserved port numbers. Only the super user can use this command. *rresvport* returns a descriptor with an address in the privileged port space to a socket. *ruserok* authenticates clients requesting service with *rcmd*. All three functions are in the same file. *rshd*(1M) and other servers use these functions.

rcmd looks up the host **ahost* using *gethostbyname*(3N). It returns -1 if the host does not exist. Otherwise, **ahost* becomes the standard name of the host, and a connection is established to a server residing at the Internet port *inport*.

If the call succeeds, a SOCK_STREAM type socket is returned to the caller and then given to the remote command as *stdin* and *stdout*. This socket has the options specified in *socket*(3N). If *fd2p* is nonzero, an auxiliary channel 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 (unit 2) and accepts bytes (as UNIX signal numbers) for forwarding to the command's process group on this channel. If *fd2p* is 0, the *stderr* (unit 2 of the remote command) becomes the *stdout* and arbitrary signals cannot be sent to the remote process. See *rshd*(1M) for more details.

rresvport obtains a socket with a privileged address bound to it. *rcmd* and other routines use this socket. Privileged addresses consist of a port in the range 0 to 1023. Only the super user can bind a privileged address to this socket.

ruserok uses the remote host's name returned by the *raddr*(3N) *gethostent*(3N) routine, and two user names. Then it checks the files */etc/hosts.equiv* and *.rhosts* in the current working directory (the local user's home directory) to see if the service request is allowed. It returns a 1 if the *hosts.equiv* file has the machine name and the local and remote user are the same (and the local user is not root) or if the *.rhosts* file has the remote user name. Otherwise, *ruserok* returns a 0.

SEE ALSO

rlogin(1C), rsh(1C), rlogind(1M), rshd(1M)

ORIGIN

4.3BSD

NAME

regcmp, regex – compile and execute regular expression

SYNOPSIS

char *regcmp (string1 [, string2, ...], (char *)0)
char *string1, *string2, ...;

char *regex (re, subject[, ret0, ...])
char *re, *subject, *ret0, ...;

extern char *__loc1;

DESCRIPTION

regcmp compiles a regular expression (consisting of the concatenated arguments) and returns a pointer to the compiled form. *malloc*(3C) is used to create space for the compiled form. It is the user's responsibility to free unneeded space so allocated. A NULL return from *regcmp* indicates an incorrect argument. *regcmp*(1) has been written to generally preclude the need for this routine at execution time.

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 mostly 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 meaning in *ed*(1).

$ Matches the end of the string; **\n** matches a new-line.

– Within brackets the minus means *through*. For example, **[a–z]** is equivalent to **[abcd...xyz]**. The **–** can appear as 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 is to be applied. The value *m* is the minimum number and *u* is a number, less than 256, which is the maximum. If only *m* is present (e.g., {m}), it indicates the exact number of times the regular expression is to be applied. The value {m,} is analogous to {m,infinity}. The plus (**+**) and star (*) operations are equivalent to {1,} and {0,} respectively.

(...)$*n* The value of the enclosed regular expression is to be returned. The value will be stored in the *(n+1)*th argument following the subject argument. At most ten enclosed regular expressions are allowed. *regex* makes its assignments unconditionally.

(...) Parentheses are used for grouping. An operator, e.g., *, **+**, { }, can work on a single character or a regular expression enclosed in parentheses. For example, (a*(cb+)*)$0.

By necessity, all the above defined symbols are special. They must, therefore, be escaped with a \ (backslash) to be used as themselves.

EXAMPLES

Example 1:

```
char *cursor, *newcursor, *ptr;
        ...
```

```
newcursor = regex((ptr = regcmp("^\n", (char *)0)), cursor);
free(ptr);
```

This example will match a leading new-line in the subject string pointed at by cursor.

Example 2:

```
char ret0[9];
char *newcursor, *name;
        . . .
name = regcmp("([A-Za-z][A-za-z0-9]{0,7})$0", (char *)0);
newcursor = regex(name, "012Testing345", ret0);
```

This example will match through the string "Testing3" and will return the address of the character after the last matched character (the "4"). The string "Testing3" will be copied to the character array *ret0*.

Example 3:

```
#include "file.i"
char *string, *newcursor;
        . . .
newcursor = regex(name, string);
```

This example applies a precompiled regular expression in **file.i** [see *regcmp*(1)] against *string*.

These routines are kept in **/lib/libPW.a**.

SEE ALSO

regcmp(1), malloc(3C).
ed(1) in the *User's Reference Manual.*

ERRORS

The user program may run out of memory if *regcmp* is called iteratively without freeing the vectors no longer required.

NAME

rexec – return stream to a remote command

SYNOPSIS

rem = rexec(ahost, inport, user, passwd, cmd, fd2p);
char **ahost;
int inport;
char *user, *passwd, *cmd;
int *fd2p;

DESCRIPTION

rexec looks up the host **ahost* using *gethostbyname*(3N), returning –1 if the host does not exist. Otherwise **ahost* is set to the standard name of the host. If a username and password are both specified, then these are used to authenticate to the foreign host; otherwise the environment and then the user's *.netrc* file in his home directory are searched for appropriate information. If all this fails, the user is prompted for the information.

The port *inport* specifies which well-known DARPA Internet port to use for the connection; the call "getservbyname("exec", "tcp")" (see *getservent*(3N)) will return a pointer to a structure, which contains the necessary port. The protocol for connection is described in detail in *rexecd*(1M).

If the connection succeeds, a socket in the Internet domain of type SOCK_STREAM is returned to the caller, and given to the remote command as **stdin** and **stdout**. If *fd2p* is non-zero, then an auxiliary channel to a control process will be setup, and a descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (unit 2) on this channel, and will also accept bytes on this channel as being UNIX signal numbers, to be forwarded to the process group of the command. The diagnostic information returned does not include remote authorization failure, as the secondary connection is set up after authorization has been verified. If *fd2p* is 0, then the **stderr** (unit 2 of the remote command) will be made the same as the **stdout** and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

SEE ALSO

rcmd(3), rexecd(1M) s'

NAME

rnusers, rusers – return information about users on remote machines

SYNOPSIS

```
#include <rpcsvc/rusers.h>

rnusers(host)
        char *host

rusers(host, up)
        char *host
        struct utmpidlearr *up;
```

DESCRIPTION

Rnusers returns the number of users logged on to *host* (–1 if it cannot determine that number). *rusers* fills the *utmpidlearr* structure with data about *host*, and returns 0 if successful. 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 INFO

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

ORIGIN
Sun Microsystems

NAME

round: anint, dnint, nint, idnint – Fortran nearest integer functions

SYNOPSIS

```
integer i
real r1, r2
double precision dp1, dp2
r2 = anint(r1)
i = nint(r1)
dp2 = anint(dp1)
dp2 = dnint(dp1)
i = nint(dp1)
i = idnint(dp1)
```

DESCRIPTION

anint returns the nearest whole real number to its real argument (i.e., int(a+0.5) if a $\geq$ 0, int(a–0.5) otherwise). *dnint* does the same for its double-precision argument. *nint* returns the nearest integer to its real argument. *Idnint* is the double-precision version. *anint* is the generic form of *anint* and *dnint* , performing the same operation and returning the data type of its argument. *nint* is also the generic form of *idnint*.

NAME

rwall – write to specified remote machines

SYNOPSIS

#include <rpcsvc/rwall.h>

rwall(host, msg);
char *host, *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

SEE ALSO

rwall(1), shutdown(1m), rwalld(1m)

ORIGIN

Sun Microsystems

NAME

scanf, fscanf, sscanf – convert formatted input

SYNOPSIS

#include <stdio.h>

int scanf (format [, pointer] . . .)
char *format;

int fscanf (stream, format [, pointer] . . .)
FILE *stream;
char *format;

int sscanf (s, format [, pointer] . . .)
char *s, *format;

DESCRIPTION

scanf reads from the standard input stream *stdin*. *fscanf* reads from the named input *stream*. *sscanf* reads from the character string *s*. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string *format* described below, and a set of *pointer* arguments indicating where the converted input should be stored. The results are undefined in there are insufficient *arg*s for the format. If the format is exhausted while *arg*s remain, the excess *arg*s are simply ignored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. White-space characters (blanks, tabs, new-lines, or form-feeds) which, except in two cases described below, cause input to be read up to the next non-white-space character.
2. An ordinary character (not %), which must match the next character of the input stream.
3. Conversion specifications, consisting of the character %, an optional assignment suppressing character *, an optional numerical maximum field width, an optional **l** (ell) or **h** indicating the size of the receiving variable, and a conversion code.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by *. The suppression of assignment provides a way of describing an input field which is 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 usually 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 integer pointer.

x a hexadecimal integer is expected; the corresponding argument should be an integer pointer.

i an integer is expected; the corresponding argument should be an integer pointer. It will store the value of the next input item interpreted according to C conventions: a leading "0" implies octal; a leading "0x" implies hexadecimal; otherwise, decimal.

n stores in an integer argument the total number of characters (including white space) that have been scanned so far since the function call. No input is consumed.

e,f,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 decimal point, followed by an optional exponent field consisting of an **E** or an **e**, followed by an optional +, −, or space, followed by an integer.

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 will be added automatically. The input field is terminated by a white-space character.

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 should refer to a character array; the indicated number of characters is read.

[indicates string data and the normal skip over leading white space is suppressed. The left bracket is followed by a set of characters, which we will call 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. There are some conventions used in the construction of the scanset. A range of characters may be represented by the construct *first–last*, thus [0123456789] may be expressed [0–9]. Using this convention, *first* must be lexically less than or equal to *last*, or else the dash will stand for itself. The dash will also stand for itself whenever 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, and in this case it will not be syntactically interpreted 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 will be added automatically. At least one character must match for this conversion to be considered successful.

The conversion characters **d**, **u**, **o**, **x** and **i** may be preceded by **l** or **h** to indicate that a pointer to **long** or to **short** rather than to **int** is in the argument list. Similarly, the conversion characters **e**, **f**, and **g** may be preceded by **l** to indicate that a pointer to **double** rather than to **float** is in the argument list. The **l** or **h** modifier is ignored for other conversion characters.

scanf conversion terminates 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.

scanf returns the number of successfully matched and assigned input items; this number can be zero in the event of an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, **EOF** is returned.

EXAMPLES

The call:

```
int n ; float x; char name[50];
n = scanf ("%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E–1 thompson
```

will assign to *n* the value **3**, to *i* the value **25**, to *x* the value **5.432**, and *name* will contain **thompson\0** . Or:

```
int i, j; float x; char name[50];
(void) scanf ("%i%2d%f%*d %[0–9] ", &j, &i, &x, name);
```

with input:

```
011 56789 0123 56a72
```

will assign **9** to *j*, **56** to *i*, **789.0** to *x*, skip **0123**, and place the string **56\0** in *name*. The next call to *getchar* [see *getc*(3S)] will return **a**. Or:

```
int i, j, s, e; char name[50];
(void) scanf ("%i %i %n%s%n", &i, &j, &s, name, &e);
```

with input:

```
0x11 0xy johnson
```

will assign **17** to *i*, **0** to *j*, **6** to *s*, will place the string **xy\0** in *name*, and will assign **8** to *e*. Thus, the length of *name* is *e* - *s* = 2 . The next call to *getchar* [see *getc*(3S)] will return a blank.

SEE ALSO

getc(3S), printf(3S), stdio(3S), strtod(3C), strtol(3C).

DIAGNOSTICS

These functions return **EOF** on end of input and a short count for missing or illegal data items.

CAVEATS

Trailing white space (including a new-line) is left unread unless matched in the control string.

NAME

setbuf, setvbuf – assign buffering to a stream

SYNOPSIS

```
#include <stdio.h>

void setbuf (stream, buf)
FILE *stream;
char *buf;

int setvbuf (stream, buf, type, size)
FILE *stream;
char *buf;
int type, size;
```

DESCRIPTION

setbuf may 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 may be used after a stream has been opened but before it is read or written. *type* determines how *stream* will 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. If *buf* is not the **NULL** pointer, the array it points to will be used for buffering, instead of an automatically allocated buffer. *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.

NOTES

A common source of error is allocating buffer space as an "automatic" variable in a code block, and then failing to close the stream in the same block.

NAME

setjmp, longjmp – non-local goto

SYNOPSIS

```
#include <setjmp.h>

int setjmp (env)
jmp_buf env;

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

DESCRIPTION

These functions are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

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* will return 1. At the time of the second return from *setjmp,* all accessible data have values as of the time *longjmp* is called. However, global variables will have the expected values, i.e. those as of the time of the *longjmp* (see example).

EXAMPLE

```
#include <setjmp.h>

jmp_buf env;
int i = 0;
main ()
{
        void exit();

        if(setjmp(env) != 0) {
                (void) printf("value of i on 2nd return from setjmp: %d\n", i);
                exit(0);
        }
        (void) printf("value of i on 1st return from setjmp: %d\n", i);
        i = 1;
        g();
        /*NOTREACHED*/
}

g()
{
        longjmp(env, 1);
        /*NOTREACHED*/
}
```

If the a.out resulting from this C language code is run, the output will be:

```
        value of i on 1st return from setjmp:0
```

value of i on 2nd return from setjmp:1

SEE ALSO

signal(2).

WARNING

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 which has since returned, absolute chaos is guaranteed.

ERRORS

The values of the registers on the second return from *setjmp* are the register values at the time of the first call to *setjmp*, not those at the time of the *longjmp*. This means that variables in a given function may behave differently in the presence of *setjmp*, depending on whether they are register or stack variables.

NAME

setuid, seteuid, setruid, setgid, setegid, setrgid – set user and group ID

SYNOPSIS

#include <sys/types.h>

setuid(uid)
seteuid(euid)
setruid(ruid)
uid_t uid, euid, ruid;

setgid(gid)
setegid(egid)
setrgid(rgid)
gid_t gid, egid, rgid;

DESCRIPTION

setuid (*setgid*) sets both the real and effective user ID (group ID) of the current process to as specified.

seteuid (*setegid*) sets the effective user ID (group ID) of the current process.

setruid (*setrgid*) sets the real user ID (group ID) of the current process.

These calls are only permitted to the super-user or if the argument is the real or effective ID.

SEE ALSO

setreuid(2), setregid(2), getuid(2), getgid(2)

DIAGNOSTICS

Zero is returned if the user (group) ID is set; −1 is returned otherwise.

ORIGINS

BSD 4.3

NAME

gethostsex - get the byte sex of the host machine

swap_*() - swap the sex of the specified structure

SYNOPSIS

```
#include <sex.h>
#include <filehdr.h>
#include <aouthdr.h>
#include <scnhdr.h>
#include <sym.h>
#include <symconst.h>
#include <cmplrs/stsupport.h>
#include <reloc.h>
#include <ar.h>

int gethostsex()

long swap_word(word)
long word;

short swap_half(half)
short half;

void swap_filehdr(pfilehdr, destsex)
FILHDR *pfilehdr;
long destsex;

void swap_aouthdr(paouthdr, destsex)
AOUTHDR *paouthdr;
long destsex;

void swap_scnhdr(pscnhdr, destsex)
SCNHDR *pscnhdr;
long destsex;

void swap_hdr(phdr, destsex)
pHDRR phdr;
long destsex;

void swap_fd(pfd, count, destsex)
pFDR pfd;
long count;
long destsex;

void swap_fi(pfi, count, destsex)
pFIT pfi;
long count;
long destsex;

void swap_sym(psym, count, destsex)
pSYMR psym;
long count;
long destsex;

void swap_ext(pext, count, destsex)
pEXTR pext;
long count;
long destsex;
```

```
void swap_pd(ppd, count, destsex)
pPDR ppd;
long count;
long destsex;

void swap_dn(pdn, count, destsex)
pRNDXR pdn;
long count;
long destsex;

void swap_opt(popt, count, destsex)
pOPTR popt;
long count;
long destsex;

void swap_aux(paux, type, destsex)
pAUXU paux;
long type;
long destsex;

void swap_reloc(preloc, count, destsex)
struct reloc *preloc;
long count;
long destsex;

void swap_ranlib(pranlib, count, destsex)
struct ranlib *pranlib;
long count;
long destsex;
```

DESCRIPTION

To use these routines, the library *libmld.a* must be loaded.

Gethostsex returns one of two constants BIGENDIAN or LITTLEENDIAN for the sex of the host machine. These constants are in *sex.h*.

All *swap_** routines that swap headers take a pointer to a header structure to change the byte's sex. The *destsex* argument lets the swap routines decide whether to swap bitfields before or after swapping the words they occur in. If *destsex* equals the hostsex of the machine you are running on, the flip happens before the swap; otherwise, the flip happens after the swap. Although not all routines swap structures containing bitfields, the destsex is required in the anticipation of future need.

The *swap_aux* routine takes a pointer to an aux entry and a *type*, which is a ST_AUX_* constant in cmplrs/stsupport.h. The constant specifies the type of the aux entry to change the sex of. All other *swap_** routines are passed a pointer to an array of structures and a *count* of structures to change the byte sex of. The routines *swap_word* and *swap_half* are macros declared in *sex.h*. Only the include files necessary to describe the structures being swapped need be included.

AUTHOR

Kevin Enderby

NAME

sign, isign, dsign – Fortran transfer-of-sign intrinsic function

SYNOPSIS

```
integer i, j, k
real r1, r2, r3
double precision dp1, dp2, dp3
k = isign(i, j)
k = sign(i, j)
r3 = sign(r1, r2)
dp3 = dsign(dp1, dp2)
dp3 = sign(dp1, dp2)
```

DESCRIPTION

isign returns the magnitude of its first argument with the sign of its second argument. *sign* and *dsign* are its real and double-precision counterparts, respectively. The generic version is *sign* and will devolve to the appropriate type depending on its arguments.

NAME

signal – simplified software signal facilities

SYNOPSIS

#include <signal.h>

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

DESCRIPTION

A signal is generated by some abnormal event, initiated by a user at a terminal (quit, interrupt, stop), by a program error (bus error, etc.), by request of another program (kill), or when a process is stopped because it wishes to access its control terminal while in the background (see *tty(7)*). Signals are optionally generated when a process resumes after being stopped, when the status of child processes changes, or when input is ready at the control terminal. Most signals cause termination of the receiving process if no action is taken; some signals instead cause the process receiving them to be stopped, or are simply discarded if the process has not requested otherwise. Except for the SIGKILL and SIGSTOP signals, the **signal** call allows signals either to be ignored or to cause an interrupt to a specified location. The following is a list of all signals with names as in the include file *<signal.h>*:

SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3*	quit
SIGILL	4*	illegal instruction
SIGTRAP	5*	trace trap
SIGIOT	6*	IOT instruction
SIGEMT	7*	EMT instruction
SIGFPE	8*	floating point exception
SIGKILL	9	kill (cannot be caught or ignored)
SIGBUS	10*	bus error
SIGSEGV	11*	segmentation violation
SIGSYS	12*	bad argument to system call
SIGPIPE	13	write on a pipe with no
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGURG	16•	urgent condition present on socket
SIGSTOP	17†	stop (cannot be caught or ignored)
SIGTSTP	18†	stop signal generated from keyboard
SIGCONT	19•	continue after stop
SIGCHLD	20•	child status has changed
SIGTTIN	21†	background read attempted from control terminal
SIGTTOU	22†	background write attempted to control terminal
SIGIO	23•	i/o is possible on a descriptor
SIGXCPU	24	cpu time limit exceeded
SIGXFSZ	25	file size limit exceeded
SIGVTALRM	26	virtual time alarm
SIGPROF	27	profiling timer alarm
SIGWINCH	28•	Window size change
SIGUSR1	30	User defined signal 1
SIGUSR2	31	User defined signal 2

The starred signals in the list above cause a core image if not caught or ignored.

If *func* is SIG_DFL, the default action for signal *sig* is reinstated; this default is termination (with a core image for starred signals) except for signals marked with • or †. Signals marked with • are discarded if the action is SIG_DFL ; signals marked with † cause the process to stop. If *func* is SIG_IGNthe signal is subsequently ignored and pending instances of the signal are discarded. Otherwise, when the signal occurs further occurrences of the signal are automatically blocked and *func* is called.

A return from the function unblocks the handled signal and continues the process at the point it was interrupted. **Unlike previous signal facilities, the handler *func* remains installed after a signal has been delivered.**

If a caught signal occurs during certain system calls, causing the call to terminate prematurely, the call is automatically restarted. In particular this can occur during a *read(2)* or *write(2)* on a slow device (such as a terminal; but not a file) and during a *wait(2)*.

The value of **signal** is the previous (or initial) value of *func* for the particular signal.

After a *fork(2)* the child inherits all signals. *Execve* (see *exec(2)*) resets all caught signals to the default action; ignored signals remain ignored.

RETURN VALUE

The previous action is returned on a successful call. Otherwise, −1 is returned and *errno* is set to indicate the error.

ERRORS

Signal will fail and no action will take place if one of the following occur:

[EINVAL] *sig* is not a valid signal number.

[EINVAL] An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP.

[EINVAL] An attempt is made to ignore SIGCONT (by default SIGCONT is ignored).

SEE ALSO

kill(1), cacheflush(2), ptrace(2), kill(2), setjmp(3C), tty(7).
R2010 Floating Point Coprocessor Architecture
R2360 Floating Point Board Product Description

NOTES (MIPS)

The handler routine can be declared:

```
handler(sig, code, scp)
int sig, code;
struct sigcontext *scp;
```

Here *sig* is the signal number. MIPS hardware exceptions are mapped to specific signals as defined by the table below. *code* is a parameter that is either a constant as given below or zero. *scp* is a pointer to the *sigcontext* structure (defined in <*signal.h*>), that is the context at the time of the signal and is used to restore the context if the signal handler returns.

The following defines the mapping of MIPS hardware exceptions to signals and codes. All of these symbols are defined in either <*signal.h*> or <*mips/cpu.h*>:

Hardware exception	Signal	Code
Integer overflow	SIGFPE	EXC_OV
Segmentation violation	SIGSEGV	SEXC_SEGV
Illegal Instruction	SIGILL	EXC_II
Coprocessor Unusable	SIGILL	SEXC_CPU
Data Bus Error	SIGBUS	EXC_DBE
Instruction Bus Error	SIGBUS	EXC_IBE
Read Address Error	SIGBUS	EXC_RADE

Write Address Error	SIGBUS	EXC_WADEs+1
User Breakpoint (used by debuggers)	SIGTRAP	BRK_USERBP
Kernel Breakpoint (used by prom)	SIGTRAP	BRK_KERNELBP
Taken Branch Delay Emulation	SIGTRAP	BRK_BD_TAKEN
Not Taken Branch Delay Emulation	SIGTRAP	BRK_BD_NOTTAKEN
User Single Step (used by debuggers)	SIGTRAP	BRK_SSTEPBP
Overflow Check	SIGTRAP	BRK_OVERFLOW
Divide by Zero Check	SIGTRAP	BRK_DIVZERO
Range Error Check	SIGTRAP	BRK_RANGE

When a signal handler is reached, the program counter in the signal context structure (*sc_pc*) points at the instruction that caused the exception as modified by the *branch delay* bit in the *cause* register. The *cause* register at the time of the exception is also saved in the sigcontext structure (*sc_cause*). If the instruction that caused the exception is at a valid user address it can be retrieved with the following code sequence:

```
if(scp->sc_cause & CAUSE_BD){
    branch_instruction = *(unsigned long *)(scp->sc_pc);
    exception_instruction = *(unsigned long *)(scp->sc_pc + 4);
}
else
    exception_instruction = *(unsigned long *)(scp->sc_pc);
```

Where CAUSE_BD is defined in *<mips/cpu.h>*.

The signal handler may fix the cause of the exception and re-execute the instruction, emulate the instruction and then step over it or perform some non-local goto such as a *longjump()* or an *exit()*.

If corrective action is performed in the signal handler and the instruction that caused the exception would then execute without a further exception, the signal handler simply returns and re-executes the instruction (even when the *branch delay* bit is set).

If execution is to continue after stepping over the instruction that caused the exception the program counter must be advanced. If the *branch delay* bit is set the program counter is set to the target of the branch else it is incremented by 4. This can be done with the following code sequence:

```
if(scp->sc_cause & CAUSE_BD)
    emulate_branch(scp, branch_instruction);
else
    scp->sc_pc += 4;
```

emulate_branch() modifies the program counter value in the sigcontext structure to the target of the branch instruction. See *emulate_branch(3)* for more details.

For SIGFPE's generated by floating-point instructions (*code* == 0) the *floating-point control and status* register at the time of the exception is also saved in the sigcontext structure (*sc_fpc_csr*). This register has the information on which exceptions have occurred. When a signal handler is entered the register contains the value at the time of the exception but with the *exceptions bits* cleared. On a return from the signal handler the exception bits in the floating-point control and status register are also cleared so that another SIGFPE will not occur (all other bits are restored from *sc_fpc_csr*).

If the floating-point unit is a R2360 (a floating-point board) and a SIGFPE is generated by the floating-point unit (*code* == 0) and program counter does not point at the instruction that caused the exception. In this case the instruction that caused the exception is in the *floating-point instruction exception* register. The floating-point instruction exception register at the time of the exception is also saved in the sigcontext structure (*sc_fpc_eir*). In this case the

instruction that caused the exception can be retrieved with the following code sequence:

```
union fpc_irr fpc_irr;

fpc_irr.fi_word = get_fpc_irr();
if(sig == SIGFPE && code == 0 &&
   fpc_irr.fi_struct.implementation == IMPLEMENTATION_R2360)
    exception_instruction = scp->sc_fpc_eir;
```

The union *fpc_irr,* and the constant IMPLEMENTATION_R2360 are defined in *<mips/fpu.h>*. For the description of the routine *get_fpc_irr()* see *fpc(3)*. All other floating-point implementations are handled in the normal manner with the instruction that caused the exception at the program counter as modified by the *branch delay* bit.

For SIGSEGV and SIGBUS errors the faulting virtual address is saved in *sc_badvaddr* in the signal context structure.

The SIGTRAP's caused by **break** instructions noted in the above table and all other yet to be defined **break** instructions fill the *code* parameter with the first argument to the **break** instruction (bits 25-16 of the instruction).

NAME

signal – change the action for a signal

SYNOPSIS

integer function signal(signum, proc, flag)
integer signum, flag
external proc

DESCRIPTION

When a process incurs a signal (see *signal*(3C)) the default action is usually to clean up and abort. The user may choose to write an alternative signal handling routine. A call to *signal* is the way this alternate action is specified to the system.

Signum is the signal number (see *signal*(3C)). If *flag* is negative, then *proc* must be the name of the user signal handling routine. If *flag* is zero or positive, then *proc* is ignored and the value of *flag* is passed to the system as the signal action definition. In particular, this is how previously saved signal actions can be restored. Two possible values for *flag* have specific meanings: 0 means "use the default action" (See NOTES below), 1 means "ignore this signal".

A positive returned value is the previous action definition. A value greater than 1 is the address of a routine that was to have been called on occurrence of the given signal. The returned value can be used in subsequent calls to *signal* in order to restore a previous action definition. A negative returned value is the negation of a system error code. (See *perror*(3F))

FILES

/usr/lib/libU77.a

SEE ALSO

signal(3C), kill(3F), kill(1)

NOTES

f77 arranges to trap certain signals when a process is started. The only way to restore the default **f77** action is to save the returned value from the first call to *signal*.

If the user signal handler is called, it will be passed the signal number as an integer argument.

NAME

sin, dsin, csin – Fortran sine intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
complex cx1, cx2
r2 = sin(r1)
dp2 = dsin(dp1)
dp2 = sin(dp1)
cx2 = csin(cx1)
cx2 = sin(cx1)
```

DESCRIPTION

sin returns the real sine of its real argument. *dsin* returns the double-precision sine of its double-precision argument. *csin* returns the complex sine of its complex argument. The generic *sin* function becomes *dsin* or *csin* as required by argument type.

SEE ALSO

trig(3M).

NAME

sin, cos, tan, asin, acos, atan, atan2 – trigonometric functions and their inverses

SYNOPSIS

```
#include <math.h>

double sin(x)
double x;

float fsin(float x)
float x;

double cos(x)
double x;

float fcos(float x)
float x;

double tan(float x)
double x;

float ftan(float x)
float x;

double asin(x)
double x;

float fasin(float x)
float x;

double acos(x)
double x;

float facos(float x)
float x;

double atan(x)
double x;

float fatan(float x)
float x;

double atan2(y,x)
double y,x;

float fatan2(float y,float x)
float y,x;
```

DESCRIPTION

Sin, cos and tan return trigonometric functions of radian arguments x for double data types. Fsin, fcos and ftan do the same for float data types.

Asin and fasin returns the arc sine in the range $-\pi/2$ to $\pi/2$ for double and float data types respectively.

Acos and facos returns the arc cosine in the range 0 to π for double and float data types respectively.

Atan and fatan returns the arc tangent in the range $-\pi/2$ to $\pi/2$ for double and float data types respectively.

Atan2 and fatan2 returns the arctangent of y/x in the range $-\pi$ to π, using the signs of both arguments to determine the quadrant of the return value for double and float data types respectively.

DIAGNOSTICS

If |x| > 1 then asin(x) and acos(x) will return the default quiet *NaN*.

NOTES

Atan2 defines atan2(0,0) = 0. The reasons for assigning a value to atan2(0,0) are these:

(1) Programs that test arguments to avoid computing atan2(0,0) must be indifferent to its value. Programs that require it to be invalid are vulnerable to diverse reactions to that invalidity on diverse computer systems.

(2) Atan2 is used mostly to convert from rectangular (x,y) to polar (r,θ) coordinates that must satisfy $x = r*\cos\theta$ and $y = r*\sin\theta$. These equations are satisfied when (x=0,y=0) is mapped to ($r=0,\theta=0$). In general, conversions to polar coordinates should be computed thus:

r := hypot(x,y); ... := $\sqrt{(x^2+y^2)}$

θ := atan2(y,x).

(3) The foregoing formulas need not be altered to cope in a reasonable way with signed zeros and infinities on a machine, such as MIPS machines, that conforms to IEEE 754; the versions of hypot and atan2 provided for such a machine are designed to handle all cases. That is why atan2($\pm 0,-0$) = $\pm\pi$, for instance. In general the formulas above are equivalent to these:

r := $\sqrt{(x*x+y*y)}$; if r = 0 then x := copysign(1,x);

if x > 0 then θ := 2*atan(y/(r+x))

else θ := 2*atan((r−x)/y);

except if r is infinite then atan2 will yield an appropriate multiple of $\pi/4$ that would otherwise have to be obtained by taking limits.

ERROR (due to Roundoff etc.) for

Let P stand for the number stored in the computer in place of π = 3.14159 26535 89793 23846 26433 ... . Let "trig" stand for one of "sin", "cos" or "tan". Then the expression "trig(x)" in a program actually produces an approximation to trig(x*π/P), and "atrig(x)" approximates (P/π)*atrig(x). The approximations are close.

In the codes that run on MIPS machines, P differs from π by a fraction of an *ulp*; the difference matters only if the argument x is huge, and even then the difference is likely to be swamped by the uncertainty in x. Besides, every trigonometric identity that does not involve π explicitly is satisfied equally well regardless of whether P = π. For instance, $\sin^2(x)+\cos^2(x) = 1$ and $\sin(2x) = 2\sin(x)\cos(x)$ to within a few *ulps* no matter how big x may be. Therefore the difference between P and π is most unlikely to affect scientific and engineering computations.

SEE ALSO

math(3M), hypot(3M), sqrt(3M)

AUTHOR

Robert P. Corbett, W. Kahan, Stuart I. McDonald, Peter Tang and, for the codes for IEEE 754, Dr. Kwok−Choi Ng.

NAME

sinh, dsinh – Fortran hyperbolic sine intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2
r2 = sinh(r1)
dp2 = dsinh(dp1)
dp2 = sinh(dp1)

DESCRIPTION

sinh returns the real hyperbolic sine of its real argument. *dsinh* returns the double-precision hyperbolic sine of its double-precision argument. The generic form *sinh* may be used to return a double-precision value when given a double-precision argument.

SEE ALSO

sinh(3M).

NAME

sinh, cosh, tanh – hyperbolic functions

SYNOPSIS

```
#include <math.h>

double sinh(x)
double x;

float fsinh(float x)
float x;

double cosh(x)
double x;

float fcosh(float x)
float x;

double tanh(x)
double x;

float ftanh(float x)
float x;
```

DESCRIPTION

These functions compute the designated hyperbolic functions for double and float data types.

ERROR (due to Roundoff etc.)

Below 2.4 *ulps*; an *ulp* is one *U*nit in the *L*ast *P*lace.

DIAGNOSTICS

Sinh and cosh return $+\infty$ (and *sinh* may return $-\infty$ for negative x) if the correct value would overflow.

SEE ALSO

math(3M)

AUTHOR

W. Kahan, Kwok–Choi Ng

NAME

sleep – suspend execution for interval

SYNOPSIS

unsigned sleep (seconds)
unsigned seconds;

DESCRIPTION

The current process is suspended from execution for the number of *seconds* specified by the argument. The actual suspension time may be less than that requested for two reasons: (1) Because scheduled wakeups occur at fixed 1-second intervals, (on the second, according to an internal clock) and (2) because any caught signal will terminate the *sleep* following execution of that signal's catching routine. Also, the suspension time may be longer than requested by an arbitrary amount due to the scheduling of other activity in the system. The value returned by *sleep* will be 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.

The routine 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 till such 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. But 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*.

SEE ALSO

alarm(2), pause(2), signal(2).

NAME

sleep – suspend execution for an interval

SYNOPSIS

subroutine sleep (itime)

DESCRIPTION

Sleep causes the calling process to be suspended for *itime* seconds. The actual time can be up to 1 second less than *itime* due to granularity in system timekeeping.

FILES

/usr/lib/libU77.a

SEE ALSO

sleep(3)

NAME

sqrt, dsqrt, csqrt – Fortran square root intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
complex cx1, cx2
r2 = sqrt(r1)
dp2 = dsqrt(dp1)
dp2 = sqrt(dp1)
cx2 = csqrt(cx1)
cx2 = sqrt(cx1)
```

DESCRIPTION

sqrt returns the real square root of its real argument. *dsqrt* returns the double-precision square root of its double-precision argument. *csqrt* returns the complex square root of its complex argument. *sqrt*, the generic form, will become *dsqrt* or *csqrt* as required by its argument type.

SEE ALSO

exp(3M).

NAME

cbrt, sqrt – cube root, square root

SYNOPSIS

#include <math.h>

double cbrt(x)
double x;

double sqrt(x)
double x;

float fsqrt(float x)
float x;

DESCRIPTION

Cbrt(x) returns the cube root of x.

Sqrt(x) and fsqrt(x) returns the square root of x for double and float data types respectively.

DIAGNOSTICS

Sqrt returns the default quiet *NaN* when *x* is negative indicating the invalid operation.

ERROR (due to Roundoff etc.)

Cbrt is accurate to within 0.7 *ulp*s.

Sqrt on MIPS machines conforms to IEEE 754 and is correctly rounded in accordance with the rounding mode in force; the error is less than half an *ulp* in the default mode (round–to–nearest). An *ulp* is one *U*nit in the *L*ast *P*lace carried.

SEE ALSO

math(3M)

AUTHOR

W. Kahan

NAME

standard – VADS standard library

SYNOPSIS

standard

DESCRIPTION

standard contains the VADS implementation of package STANDARD containing all predefined indentifiers in the Ada RM as well as other predefined library units. The package STANDARD is an imaginary package that is available to every Ada program. The package enables Ada programmers to use predefined types, functions, and operations on those types.

Additional packages are available as described in the Ada RM.

The packages in **standard** include all types, functions, and operations described in the *Ada RM Annex C*, **Predefined Language Environment**.

FILES

*/usr/vads5/standard/**

SEE ALSO

examples, *publiclib*, *verdixlib*

NAME

stat, fstat – get file status

SYNOPSIS

integer function stat (name, statb)
character*(*) name
integer statb(12)
character*(*) name
integer statb(12)

integer function fstat (lunit, statb)
integer statb(12)

DESCRIPTION

These routines return detailed information about a file. *Stat* returns information about file *name*; *fstat* returns information about the file associated with fortran logical unit *lunit*. The order and meaning of the information returned in array *statb* is as described for the structure *stat* under *stat*(2). The "spare" values are not included.

The value of either function will be zero if successful; an error code otherwise.

FILES

/usr/lib/libU77.a

SEE ALSO

stat(2), access(3F), perror(3F), time(3F)

BUGS

Pathnames can be no longer than MAXPATHLEN as defined in *<sys/param.h>*.

NAME

staux – routines that provide scalar interfaces to auxiliaries

SYNOPSIS

```
#include <syms.h>

long st_auxbtadd(bt)
long bt;

long st_auxbtsize(iaux,width)
long iaux;
long width;

long st_auxisymadd (isym)
long isym;

long st_auxrndxadd (rfd,index)
long rfd;
long index;

long st_auxrndxadd (idn)
long idn;

void st_addtq (iaux,tq)
long iaux;
long tq;

long st_tqhigh_aux(iaux)
long iaux;

void st_shifttq (iaux, tq)
int iaux;
int tq;

long st_iaux_copyty (ifd, psym)
long ifd;
pSYMR psym;

void st_changeaux (iaux, aux)
long iaux;
AUXU aux;

void st_changeauxrndx (iaux, rfd, index)
long iaux;
long rfd;
long index;
```

DESCRIPTION

Auxiliary entries are unions with a fixed length of four bytes per entry. Much information is packed within the auxiliaries. Rather than have the compiler front-ends handle each type of auxiliary entry directly, the following set of routines provide a high-level scalar interface to the auxiliaries:

st_auxbtadd

Adds a type information record (TIR) to the auxiliaries. It sets the basic type (bt) to the argument and all other fields to zero. The index to this auxiliary entry is returned.

st_auxbtsize

Sets the bit in the TIR, pointed to by the *iaux* argument. This argument says the basic type is a bit field and adds an auxiliary with its width in bits.

st_auxisymadd

Adds an index into the symbol table (or any other scalar) to the auxiliaries. It sets the value to the argument that will occupy all four bytes. The index to this auxiliary entry is returned.

st_auxrndxadd
Adds a relative index, RNDXR, to the auxiliaries. It sets the rfd and index to their respective arguments. The index to this auxiliary entry is returned.

st_auxrndxadd_idn
Works the same as *st_auxrndxadd* except that RNDXR is referenced by an index into the dense number table.

st_iaux_copyty
Copies the type from the specified file (ifd) for the specified symbol into the auxiliary table for the current file. It returns the index to the new aux.

st_shifttq
Shifts in the specified type qualifier, tq, into the auxiliary entry TIR, which is specified by the 'iaux' index into the current file. The current type qualifiers shift up one tq so that the first tq (tq0) is free for the new entry.

st_addtq
Adds a type qualifier in the highest or most significant non-tqNil type qualifier.

st_tqhigh_iaux
Returns the most significant type qualifier given an index into the files aux table.

st_changeaux
Changes the iauxth aux in the current file's auxiliary table to aux.

st_changeauxrndx
Converts the relative index (RNDXR) auxiliary, which is specified by iaux, to the specified arguments.

AUTHOR Mark I. Himelstein

SEE ALSO
stfd(3)

BUGS
The interface will added to incrementally, as needed.

NAME

stcu – routines that provide a compilation unit symbol table interface

SYNOPSIS

```
#include <syms.h>

pCHDRR st_cuinit ()

void st_setchdr (pchdr)
pCHDRR      pchdr;

pCHDRR st_currentpchdr()

void st_free()

long st_extadd (iss, value, st, sc, index)
long iss;
long value;
long st;
long sc;
long index;

pEXTR st_pext_iext (iext)
long    iext;

pEXTR st_pext_rndx (rndx)
RNDXR rndx;

long st_iextmax()

long st_extstradd (str)
char *str;

char *st_str_extiss (iss)
long iss;

long st_idn_index_fext (index, fext)
long index;
long fext;

long st_idn_rndx (rndx)
RNDXR rndx;

pRNDXR st_pdn_idn (idn)
long idn;
RNDXR st_rndx_idn (idn)
long idn;

void st_setidn (idndest, idnsrc)
long idndest;
long idnsrc;
```

DESCRIPTION

The *stcu* routines provide an interface to objects that occur once per object rather than once per file descriptor (for example, external symbols, strings, and dense numbers). The routines provide access to the current *chdr* (compile time hdr), which represents the symbol table in running processes with pointers to symbol table sections rather than indices and offsets used in the disk file representation.

A new symbol table can be created with *st_cuinit*. This routine creates and initializes a CHDRR. The CHDRR is the current chdr and is used in all later calls. **NOTE**: A chdr can also be created with the read routines (see *stio*(3)). The *st_cuinit* routine returns a pointer to the new CHDRR record.

st_currentchdr
Returns a pointer the current chdr.

st_setchdr
Sets the current chdr to the *pchdr* argument and sets the per file structures to reflect a change in symbol tables.

st_free Frees all constituent structures associated with the current chdr.

st_extadd
Lets you add to the externals table. It returns the index to the new external for future reference and use. The *ifd* field for the external is filled in by the current file (see *stfd*(3)).

st_pext_iext

and *st_pext_rndx*
Returns pointers to the external, given a index referencing them. The latter routine requires a relative index where the *index* field should be the index in external symbols and the *rfd* field should be the constant ST_EXTIFD. **NOTE**: The externals contain the same structure as symbols (see the *SYMR* and *EXTR* definitions).

st_iextmax
Returns the current number of entries in the external symbol table.

The *iss* field in external symbols (the index into string space) must point into external string space.

st_extstradd
Adds a null-terminated string to the external string space and returns its index.

st_str_extiss
Converts that index into a pointer to the external string.

The dense number table provides a convenience to the code optimizer, generator, and assembler. This table lets them reference symbols from different files and externals with unique densely packed numbers.

st_idn_index_fext
Returns a new dense number table index, given an index into the symbol table of the current file (or if *fext* is set, the externals table).

st_idn_rndx
Returns a new dense number, but expects a RNDXR to specify both the file index and the symbol index rather than implying the file index from the current file. The RNDXR contains two fields: an index into the externals table and a file index (*rsyms* can point into the symbol table, as well). The file index is ST_EXTIFD for externals.

st_rndx_idn
Returns a RNDX, given an index into the dense number table.

st_pdn_idn
Returns a pointer to the RNDXR index by the 'idn' argument.

AUTHOR Mark I. Himelstein

SEE ALSO
stfe(3), stfd(3)

NAME

stdio – standard buffered input/output package

SYNOPSIS

#include <stdio.h>

FILE *stdin, *stdout, *stderr;

DESCRIPTION

The functions described in the entries of sub-class 3S of this manual constitute an efficient, user-level I/O buffering scheme. The in-line macros *getc*(3S) and *putc*(3S) handle characters quickly. The macros *getchar* and *putchar*, and the higher-level routines *fgetc*, *fgets*, *fprintf*, *fputc*, *fputs*, *fread*, *fscanf*, *fwrite*, *gets*, *getw*, *printf*, *puts*, *putw*, and *scanf* all use or act as if they use *getc* and *putc*; they can be freely intermixed.

A file with associated buffering is called a *stream* and is declared to be a pointer to a defined type **FILE**. *Fopen*(3S) creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the <stdio.h> header file and associated with the standard open files:

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 the individual descriptions for details).

An integer constant **BUFSIZ** specifies the size of the buffers used by the particular implementation.

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 the entries of sub-class 3S of this manual are declared in that header file and need no further declaration. The constants and the following "functions" are implemented as macros (redeclaration of these names is perilous): *getc*, *getchar*, *putc*, *putchar*, *ferror*, *feof*, *clearerr*, and *fileno*.

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)] will cause it to become buffered or line-buffered. 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 it 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). *Setbuf*(3S) or *setvbuf*() in *setbuf*(3S) may be used to change the stream's buffering strategy.

SEE ALSO

open(2), close(2), lseek(2), pipe(2), read(2), write(2), ctermid(3S), cuserid(3S), fclose(3S), ferror(3S), fopen(3S), fread(3S), fseek(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).

DIAGNOSTICS

Invalid *stream* pointers will usually cause grave disorder, possibly including program termination. Individual function descriptions describe the possible error conditions.

NAME

stdipc: ftok – standard interprocess communication package

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok(path, id)
char *path;
char id;
```

DESCRIPTION

All interprocess communication facilities require the user to supply a key to be used by the *msgget*(2), *semget*(2), and *shmget*(2) system calls to obtain interprocess communication identifiers. One suggested method for forming a key is to use the *ftok* subroutine described below. Another way to compose keys is to include the project ID in the most significant byte and to 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 which uniquely identifies a project. Note that *ftok* will return the same key for linked files when called with the same *id* and that it will return different keys when called with the same file name but different *ids*.

SEE ALSO

intro(2), msgget(2), semget(2), shmget(2).

DIAGNOSTICS

ftok returns **(key_t) −1** if *path* does not exist or if it is not accessible to the process.

WARNING

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, then *ftok* is likely to return a different key than it did the original time it was called.

NAME

stfd – routines that provide access to per file descriptor section of the symbol table

SYNOPSIS

```
#include <syms.h>

long st_currentifd ()

long st_ifdmax ()

void st_setfd (ifd)
long ifd;

long st_fdadd (filename)
char *filename;

long st_symadd (iss, value, st, sc, freloc, index)
long iss;
long value;
long st;
long sc;
long freloc;
long index;

long st_auxadd (aux)
AUXU aux;

long st_stradd (cp)
char *cp;

long st_lineadd (line)
long line;

long st_pdadd (isym)
long isym;

long st_ifd_pcfd (pcfd1)
pCFDR pcfd1;

pCFDR st_pcfd_ifd (ifd)
long ifd;

pSYMR st_psym_ifd_isym (ifd, isym)
long ifd;
long isym;

pAUXU st_paux_ifd_iaux (ifd, iaux)
long ifd;
long iaux;

pAUXU st_paux_iaux (iaux)
long iaux;

char *st_str_iss (iss)
long iss;

char *st_str_ifd_iss (ifd, iss)
long ifd;
long iss;

pPDR st_ppd_ifd_isym (ifd, isym)
long ifd;
long isym;
```

char * st_malloc (ptr, psize, itemsize, baseitems)
char *ptr;
long *size;
long itemsize;
long baseitems;

DESCRIPTION

The *stfd* routines provide an interface to objects handled on a per file descriptor (or fd) level (for example, local symbols, auxiliaries, local strings, line numbers, optimization entries, procedure descriptor entries, and the file descriptors). These routines constitute a group because they deal with objects corresponding to fields in the *FDR* structure.

A fd can be activated by reading an existing one into memory or by creating a new one. The compilation unit routines *st_readbinary* and *st_readst* read file descriptors and their constituent parts into memory from a symbol table on disk.

St_fdadd adds a file descriptor to the list of file descriptors. The *lang* field is initialized from a user specified global *st_lang* that should be set to a constant designated for the language in *symconst.h*. The *fMerge* field is initialized from the user specified global *st_merge* that specifies whether the file is to start with the attribute of being able to be merged with identical files at load time. The *fBigendian* field is initialized by the *gethostsex(3)* routine, which determines the permanent byte ordering for the auxiliary and line number entries for this file.

St_fdadd adds the null string to the new files string table that is accessible by the constant issNull (0). It also adds the filename to the string table and sets the *rss* field. Finally, the current file is set to the newly added file so that later calls operate on that file.

All routines for fd-level objects handle only the current file unless a file index is specified. The current file can also be set with *st_setfd*.

Programs can find the current file by calling *st_currentifd,* which returns the current index. Programs can find the number of files by calling *st_ifdmax*. The fd routines only require working with indices to do most things. They allow more in-depth manipulation by allowing users to get the compile time file descriptor (*CFDR*) that contains memory pointers to the per file tables (rather than indices or offsets used in disk files). Users can retrieve a pointer to the CFDR by calling *st_pcfd_ifd* with the index to the desired file. The inverse mapping *st_ifd_pcfd* exists, as well.

Each of fd's constituent parts has an add routine: *st_symadd, st_stradd, st_lineadd, st_pdadd,* and *st_auxadd*. The parameters of the add routines correspond to the fields of the added object. The *pdadd* routine lets users fill in the isym field only. Further information can be added by directly accessing the procedure descriptor entry.

The add routines return an index that can be used to retrieve a pointer to part of the desired object with one of the following routines: *st_psym_isym, st_str_iss,* and *st_paux_iaux*. **NOTE:** These routines only return objects within the current file. The following routines allow for file specification: *st_psym_ifd_isym, st_aux_ifd_iaux,* and *st_str_ifd_iss*.

St_ppd_ifd_isym allows access to procedures through the file index for the file where they occur and the isym field of the entry that points at the local symbol for that procedure.

The return index from *st_symadd* should be used to get a dense number (see *stcu*(3)). That number should be the ucode block number for the object the symbol describes.

AUTHOR Mark I. Himelstein

SEE ALSO

stfe(3), stcu(3).

BUGS

The interface will added to incrementally, as needed.

NAME

stfe – routines that provide a high-level interface to basic functions needed to access and add to the symbol table

SYNOPSIS

```
#include <syms.h>

long st_filebegin (filename, lang, merge, glevel)
char *filename;
long lang;
long merge;
long glevel;

long st_endallfiles ()

long st_fileend (idn)
long idn;

long st_blockbegin(iss, value, sc)
long iss;
long value;
long sc;

long st_textblock()

long st_blockend(size)
long size;

long st_procend(idn)
long idn

long st_procbegin (idn)
long idn;

char *st_str_idn (idn)
long idn;

char *st_sym_idn (idn, value, sc, st, index)
long idn;
long *value;
long *sc;
long *st;
long *index;

long st_abs_ifd_index (ifd, index)
long ifd;
long index;

long st_fglobal_idn (idn)
long idn;

pSYMR st_psym_idn_offset (idn, offset)
long idn;
long offset;

long st_pdadd_idn (idn)
long idn;
```

DESCRIPTION

The *stfe* routines provide a high-level interface to the symbol table based on common needs of the compiler front-ends.

st_filebegin

should be called upon encountering each cpp directive in the front end. It calls *st_fileadd* to add symbols and will find the appropriate open file or start a new file. It takes a filename, language constant (see *symconst.h*), a merge flag (0 or 1) and the -g level constant (see *symconst.h*). It returns a dense number pointing to the file symbol to be used in line number directives.

st_fileend

Requires the dense number from the corresponding *st_filebegin* call for the file in question. It then generates an end symbol and patches the references so that the index field of the begin file points to that of one beyond the end file. The end file points to the begin file.

st_endallfiles

Is called at the end of execution to close off all files that haven't been ended by previous calls to *st_filebegin*. CPP directives might not reflect the return to the original source file; therefore, this routine can possibly close many files.

st_blockbegin

Supports both language blocks (for example, C's left curly brace blocks), beginning of structures, and unions. If the storage class is scText, it is the former; if it is scInfo, it is one of the latter. The iss (index into string space) specifies the name of the structure/etc, if any.

If the storage class is scText, we must check the result of *st_blockbegin*. It returns a dense number for outer blocks and a zero for nested blocks. The non-zero block number should be used in the BGNB ucode. Users of languages without nested blocks that provide variable declarations can ignore the rest of this paragraph. Nested blocks are two-staged: one stage happens when we detect the language block and the other stage happens when we know the block has content. If the block has content (for example, local variables), the front-end must call *st_textblock* to get a non-zero dense number for the block's BGNB ucode. If the block has no content and *st_textblock* is not called, the block's *st_blockbegin* and *st_blockend* do not produce block and end symbols.

If it is scInfo, *st_blockbegin* creates a begin block symbol in the symbol table and returns a dense number referencing it. The dense number is necessary to build the auxiliary required to reference the structure/etc. It goes in the aux after the TIR along with a file index. This dense number is also noted in a stack of blocks used by *st_blockend*.

St_blockbegin should not be called for language blocks when the front-end is not producing debugging symbols.

St_blockend requires that blocks occur in a nested fashion. It retrieves the dense number for the most recently started block and creates a corresponding end symbol. As in *fileend*, both the begin and end symbol index fields point at the other end's symbol. If the symbol ends a structure/etc., as determined by the storage class of the begin symbol, the size parameter is assigned to the begin symbol's value field. It's usually the size of the structure or max value of a enum. We only know it at this point. The dense number of the end symbol is returned so that the ucode ENDB can be use it. If it is an ignored text block, the dense number is zero and no ENDB should be generated.

In general, defined external procedures or functions appear in the symbols table and the externals table. The external table definition must occur first through the use of a *st_extadd*. After that definition, *st_procbegin* can be called with a dense number referring to the external symbol for that procedure. It checks to be sure we have a defined procedure (by checking the storage class). It adds a procedure symbol to the symbol table. The external's index should point at its auxiliary data type information (or if debugging is off, indexNil). This index is copied into the regular symbol's index field or a copy of its type is generated (if the external is

in a different file than the regular symbol). Next, we put the index to symbol in the external's index field. The external's dense number is used as a block number in ucodes referencing it and is used to add a procedure when in the *st_pdadd_idn*.

st_procend

Creates an end symbol and fixes the indices as in *blockend* and *fileend*, except that the end procedure reference is kept in the begin procedure's aux rather than in the index field (because the begin procedure has a type as well as an end reference). This must be called with the dense number of the procedure's external symbol as an argument and returns the dense number of the end symbol to be used in the END ucode.

st_str_idn

Returns the string associated with symbol or external referenced by the dense number argument. If the symbol was anonymous (for example, there was no symbol) a (char *), -1 is returned.

st_sym_idn

Returns the same result as *st_str_idn*, except that the rest of the fields of the symbol specified by the *idn* are returned in the arguments.

st_fglobal_idn

Returns a 1 if the symbol associated with the specified idn is non-static; otherwise, a 0 is returned.

st_abs_ifd_index

Returns the absolute offset for a dense number. If the symbol is global, the global's index is returned. If the symbol occurred in a file, the sum of all symbols in files occurring before that file and the symbol's index within the file is returned.

st_pdadd_idn

Adds an entry to the procedure table for the *st_proc entry* generated by procbegin. This should be called when the front-end generates code for the procedure in question.

AUTHOR Mark I. Himelstein

SEE ALSO

stcu(3), stfd(3)

NAME

stio – routines that provide a binary read/write interface to the MIPS symbol table

SYNOPSIS

```
#include <syms.h>

long st_readbinary (filename, how)
char *filename;
char how;

long st_readst (fn, how, filebase, pchdr, flags)
long fn;
char how;
long filebase;
pCHDRR pchdr;
long flags;

void st_writebinary (filename, flags)
char *filename;
long flags;

void st_writest (fn, flags)
long fn;
long flags;
```

DESCRIPTION

The CHDRR structure (see *stcu*(3)) represents a symbol table in memory. A new CHDRR can be created by reading a symbol table in from disk. *St_readbinary* and *st_readst* read a symbol table in from disk.

St_readbinary takes the file name of the symbol table and assumes the symbol table header HDRR occurs at the beginning of the file. *St_readst* assumes that its file number references a file positioned at the beginning of the symbol table header and that the *filebase* parameter specifies where the object or symbol table file is based (for example, non-zero for archives).

The second parameter to the read routines can be 'r' for read only or 'a' for appending to the symbol table. Existing local symbol, line, procedure, auxiliary, optimization, and local string tables can not be appended. If they didn't exist on disk, they can be created. This restriction stems from the allocation algorithm for those symbol table sections when read in from disk and follows the standard pattern for building the symbol table.

The symbol table can be read incrementally. If *pchdr* is zero, *st_readst* assumes that no symbol table has been read yet; therefore, it reads in the symbol table header and file descriptors. The *flags* argument is a bit mask that defines what other tables should be read. *St_p** constants for each table can be ORed. If *flags* equals '-1', all tables are read. If *pchdr* is set, the tables specified by *flags* are added to the tables that have already been read. The value of *pchdr* can be gotten from *st_current_pchdr* (see *stcu*(3)).

Line number entries are encoded on disk, and the read routines expand them to longs. See the *MIPS System Programmer Guide*.

If the version stamp is out of date, a warning message is issued to *stderr*. If the magic number in the HDRR is incorrect, *st_error* is called. All other errors cause the read routines to read non-zero; otherwise, a zero is returned.

St_writebinary and *st_writest* are symmetric to the read routines, excluding the *how* and *pchdr* parameters. The *flags* parameter is a bit mask that defines what table should be written. *St_p** constants for each table can be ORed. If *flags* equals '-1', all tables are written.

The write routines write sections of the table in the approved order, as specified in the link editor (*ld*) specification.

Line numbers are compressed on disk. See the *MIPS System Programmer Guide*.

The write routines start all sections of the symbol table on four-byte boundaries.

If the write routines encounter an error, *st_error* is called. After writing the symbol table, further access to the table by other routines is undefined.

AUTHOR Mark I. Himelstein

SEE ALSO

stcu(3),stfe(3), stfd(3).
The *MIPS System Programmer Guide*.

NAME

stprint – routines to print the symbol table

SYNOPSIS

```
#include <syms.h>

#include <stdio.h>

char    *st_mlang_ascii [];
char    *st_mst_ascii [];
char    *st_msc_ascii [];
char    *st_mbt_ascii [];
char    *st_mtq_ascii [];

void st_dump (fd, flags)
FILE *fd;
long flags;

void st_printfd (fd, ifd, flags)
FILE *fd;
long ifd;
long flags;
```

DESCRIPTION

The *stprint* routines and arrays provide an easy way to print the MIPS symbol table. The print the symbol table from *st_current pchdr()*.

The arrays map constants to their ASCII equivalents. The constants can be found in *symconst.h* and represent languages (*lang*), symbol types (*st*), storage classes (*sc*), basic types (*bt*), and type qualifiers (*tq*).

The *st_dump* routine prints an ASCII version of the symbol. If fd is NULL , the routine prints file *fd* and stdout. The flags can be a mask of a section of symbol table specified by ORing *ST_P** constants together from *cmplrs/stsupport.h*. This routine modifies the current file.

st_printfd prints the sections associated with the file specified by the ifd argument. The other arguments are the same as in *st_dump*. These arguments modify the current file, as well.

AUTHOR Mark I. Himelstein

BUGS

The interface will be added to incrementally as needed.

NAME

strcmp: lge, lgt, lle, llt – string comparison intrinsic functions

SYNOPSIS

character*N a1, a2
logical l

l = lge(a1, a2)
l = lgt(a1, a2)
l = lle(a1, a2)
l = llt(a1, a2)

DESCRIPTION

These functions return .TRUE. if the inequality holds and .FALSE. otherwise.

NAME

string: strcat, strdup, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok – string operations

SYNOPSIS

```
#include <string.h>
#include <sys/types.h>

char *strcat (s1, s2)
char *s1, *s2;

char *strdup (s1)
char *s1;

char *strncat (s1, s2, n)
char *s1, *s2;
size_t n;

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

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

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

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

int strlen (s)
char *s;

char *strchr (s, c)
char *s;
int c;

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

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

int strspn (s1, s2)
char *s1, *s2;

int strcspn (s1, s2)
char *s1, *s2;

char *strtok (s1, s2)
char *s1, *s2;
```

DESCRIPTION

The arguments **s1, s2** and **s** point to strings (arrays of characters terminated by a null character). The functions *strcat*, *strncat*, *strcpy*, and *strncpy* all alter **s1**. These functions do not check for overflow of the array pointed to by **s1**.

strcat appends a copy of string **s2** to the end of string **s1**.

strdup returns a pointer to a new string which is a duplicate of the string pointed to by **s1.** The space for the new string is obtained using *malloc*(3C). If the new string can not be created, null is returned.

strncat appends at most **n** characters. Each returns a pointer to the null-terminated result.

strcmp compares its arguments and returns an integer less than, equal to, or greater than 0, according as **s1** is lexicographically less than, equal to, or greater than **s2**. *strncmp* makes the same comparison but looks at at most **n** characters.

strcpy copies string **s2** to **s1**, stopping after the null character has been copied. *strncpy* copies exactly **n** characters, truncating **s2** or adding null characters to **s1** if necessary. The result will not be null-terminated if the length of **s2** is **n** or more. Each function returns **s1**.

strlen returns the number of characters in **s**, not including the terminating null character.

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 character terminating a string is considered to be part of the string.

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 initial segment of string **s1** which consists entirely of characters from (not from) string **s2**.

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 pointer **s1** specified) returns a pointer to the first character of the first token, and will have written a null character into **s1** immediately following the returned token. The function keeps track of its position in the string between separate calls, so that subsequent calls (which must be made with the first argument a NULL pointer) will work through the string **s1** immediately following that token. In this way subsequent calls will work through the string **s1** until no tokens remain. The separator string **s2** may be different from call to call. When no token remains in **s1**, a NULL pointer is returned.

For user convenience, all these functions are declared in the optional <*string.h*> header file.

SEE ALSO

malloc(3C), malloc(3X).

CAVEATS

strcmp and *strncmp* are implemented by using the most natural character comparison on the machine. Thus the sign of the value returned when one of the characters has its high-order bit set not the same in all implementations and should not be relied upon.

Character movement is performed differently in different implementations. Thus overlapping moves may yield surprises.

NAME

strtod, atof – convert string to double-precision number

SYNOPSIS

double strtod (str, ptr)
char *str, **ptr;

double atof (str)
char *str;

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 up to the first unrecognized character.

strtod 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 decimal point, then an optional **e** or **E** followed by an optional sign or space, followed by an integer.

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*, and zero is returned.

atof(str) is equivalent to *strtod(str, (char **)NULL)*.

SEE ALSO

ctype(3C), scanf(3S), strtol(3C).

DIAGNOSTICS

If the correct value would cause overflow, ±**HUGE** (as defined in **<math.h>**) 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**.

NAME

strtol, atol, atoi – convert string to integer

SYNOPSIS

long strtol (str, ptr, base)
char *str, **ptr;
int base;

long atol (str)
char *str;

int atoi (str)
char *str;

DESCRIPTION

strtol returns as a long integer the value represented by the character string pointed to by *str*. 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 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, that location is set to *str*, and zero is returned.

If *base* is positive (and not greater than 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 thusly: After an optional leading sign a leading zero indicates octal conversion, and a leading "0x" or "0X" hexadecimal conversion. Otherwise, decimal conversion is used.

Truncation from long to int can, of course, take place upon assignment or by an explicit cast.

atol(str) is equivalent to *strtol(str, (char **)NULL, 10)*.

astoi(str) is equivalent to *(int) strtol(str, (char **)NULL, 10)*.

SEE ALSO

ctype(3C), scanf(3S), strtod(3C).

CAVEAT

Overflow conditions are ignored.

NAME

swab – swap bytes

SYNOPSIS

void swab (from, to, nbytes)
char *from, *to;
int nbytes;

DESCRIPTION

swab copies *nbytes* bytes pointed to by *from* to the array pointed to by *to*, exchanging adjacent even and odd bytes. *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.

NAME

system – execute a UNIX command

SYNOPSIS

integer function system (string)
character*(*) string

DESCRIPTION

System causes *string* to be given to your shell as input as if the string had been typed as a command. If environment variable **SHELL** is found, its value will be used as the command interpreter (shell); otherwise *sh*(1) is used.

The current process waits until the command terminates. The returned value will be the exit status of the shell. See *wait*(2) for an explanation of this value.

FILES

/usr/lib/libU77.a

SEE ALSO

exec(2), wait(2), system(3)

BUGS

String can not be longer than NCARGS–50 characters, as defined in *<sys/param.h>*.

NAME

system – issue a shell command

SYNOPSIS

#include <stdio.h>

int system (string)
char *string;

DESCRIPTION

system causes the *string* to be given to *sh*(1) as input, as if the string had been typed as a command at a terminal. The current process waits until the shell has completed, then returns the exit status of the shell.

FILES

/bin/sh

SEE ALSO

exec(2).
sh(1) in the *User's Reference Manual.*

DIAGNOSTICS

system forks to create a child process that in turn exec's /bin/sh in order to execute *string*. If the fork or exec fails, *system* returns a negative value and sets *errno*.

NAME

tan, dtan – Fortran tangent intrinsic function

SYNOPSIS

```
real r1, r2
double precision dp1, dp2
r2 = tan(r1)
dp2 = dtan(dp1)
dp2 = tan(dp1)
```

DESCRIPTION

tan returns the real tangent of its real argument. *dtan* returns the double-precision tangent of its double-precision argument. The generic *tan* function becomes *dtan* as required with a double-precision argument.

SEE ALSO

trig(3M).

NAME

tanh, dtanh – Fortran hyperbolic tangent intrinsic function

SYNOPSIS

real r1, r2
double precision dp1, dp2
r2 = tanh(r1)
dp2 = dtanh(dp1)
dp2 = tanh(dp1)

DESCRIPTION

tanh returns the real hyperbolic tangent of its real argument. *dtanh* returns the double-precision hyperbolic tangent of its double-precision argument. The generic form *tanh* may be used to return a double-precision value given a double-precision argument.

SEE ALSO

sinh(3M).

NAME

time, ctime, ltime, gmtime – return system time

SYNOPSIS

integer function time()

character*(*) function ctime (stime)
integer stime

subroutine ltime (stime, tarray)
integer stime, tarray(9)

subroutine gmtime (stime, tarray)
integer stime, tarray(9)

DESCRIPTION

Time returns the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds. This is the value of the UNIX system clock.

Ctime converts a system time to a 24 character ASCII string. The format is described under *ctime*(3). No 'newline' or NULL will be included.

Ltime and *gmtime* disect a UNIX time into month, day, etc., either for the local time zone or as GMT. The order and meaning of each element returned in *tarray* is described under *ctime*(3).

FILES

/usr/lib/libU77.a

SEE ALSO

ctime(3), itime(3F), idate(3F), fdate(3F)

NAME

tmpfile – create a temporary file

SYNOPSIS

#include <stdio.h>

FILE *tmpfile ()

DESCRIPTION

tmpfile creates a temporary file using a name generated by *tmpnam*(3S), and returns a corresponding FILE pointer. If the file cannot be opened, an error message is printed using *perror*(3C), and a NULL pointer is returned. The file will automatically be deleted when the process using it terminates. The file is opened for update ("w+").

SEE ALSO

creat(2), unlink(2), fopen(3S), mktemp(3C), perror(3C), stdio(3S), tmpnam(3S).

NAME

tmpnam, tempnam - create a name for a temporary file

SYNOPSIS

```
#include <stdio.h>

char *tmpnam (s)
char *s;

char *tempnam (dir, pfx)
char *dir, *pfx;
```

DESCRIPTION

These functions 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* will destroy 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 a name for an appropriate directory, the path-prefix defined as **P_tmpdir** in the *<stdio.h>* header file is used. If that directory is not accessible, **/tmp** will be used as a last resort. This entire sequence can be up-staged by providing an environment variable **TMPDIR** in the user's environment, whose value is the name of the desired temporary-file directory. Many applications prefer their temporary files to have certain favorite initial letter sequences in their names. Use the *pfx* argument for this. This argument may be NULL or point to a string of up to five characters to be used as the first few characters of the temporary-file name. *tempnam* uses *malloc*(3C) to get space for the constructed file name, and returns a pointer to this area. Thus, any pointer value returned from *tempnam* may serve as an argument to *free* [see *malloc*(3C)]. If *tempnam* cannot return the expected result for any reason, i.e. *malloc*(3C) failed, or none of the above mentioned attempts to find an appropriate directory was successful, a NULL pointer will be returned.

NOTES

These functions generate a different file name each time they are called. Files created using these functions and either *fopen*(3S) or *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 its use is ended.

SEE ALSO

creat(2), unlink(2), fopen(3S), malloc(3C), mktemp(3C), tmpfile(3S).

CAVEATS

If called more than 17,576 times in a single process, these functions will start recycling previously used names.

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 to render duplication by other means unlikely.

NAME

tsearch, tfind, tdelete, twalk – manage binary search trees

SYNOPSIS

#include <search.h>

char *tsearch ((char *) key, (char **) rootp, compar)
int (*compar)( );

char *tfind ((char *) key, (char **) rootp, compar)
int (*compar)( );

char *tdelete ((char *) key, (char **) rootp, compar)
int (*compar)( );

void twalk ((char *) root, action)
void (*action)( );

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. 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 a datum to be accessed or stored. If there is a datum in the tree equal to *key (the value pointed to by key), a pointer to this found datum is returned. Otherwise, *key is inserted, and a pointer to it returned. 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 will be set to point to the datum which will be at the root of the new tree.

Like *tsearch*, *tfind* will search for a datum in the tree, returning a pointer to it if found. However, if it is not found, *tfind* will return a NULL pointer. The arguments for *tfind* are the same as for *tsearch*.

tdelete deletes a node from a binary search tree. The arguments are the same as for *tsearch*. The variable pointed to by **rootp** will be 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. The first argument is the address of the node being visited. The 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. The third argument is the level of the node in the tree, with the root being level zero.

The pointers to the key and the root of the tree should be of type pointer-to-element, and cast to type pointer-to-character. Similarly, although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

EXAMPLE

The following code reads in 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.

```
#include <search.h>
#include <stdio.h>

struct node {           /* pointers to these are stored in the tree */
        char *string;
        int length;
};
char string_space[10000];       /* space to store strings */
struct node nodes[500];         /* nodes to store */
struct node *root = NULL;       /* this points to the root */

main( )
{
        char *strptr = string_space;
        struct node *nodeptr = nodes;
        void print_node( ), twalk( );
        int i = 0, node_compare( );

        while (gets(strptr) != NULL && i++ < 500)  {
                /* set node */
                nodeptr->string = strptr;
                nodeptr->length = strlen(strptr);
                /* put node into the tree */
                (void) tsearch((char *)nodeptr, (char **) &root,
                        node_compare);
                /* adjust pointers, so we don't overwrite tree */
                strptr += nodeptr->length + 1;
                nodeptr++;
        }
        twalk((char *)root, print_node);
}
/*
        This routine compares two nodes, based on an
        alphabetical ordering of the string field.
*/
int
node_compare(node1, node2)
char *node1, *node2;
{
        return strcmp(((struct node *)node1)->string,
        ((struct node *) node2)->string);
}
/*
        This routine prints out a node, the first time
        twalk encounters it.
*/
void
print_node(node, order, level)
```

```
char **node;
VISIT order;
int level;
{
        if (order == preorder || order == leaf)  {
                (void)printf("string = %20s,  length = %d\n",
                        (*((struct node **)node))->string,
                        (*((struct node **)node))->length);
        }
}
```

SEE ALSO

bsearch(3C), hsearch(3C), lsearch(3C).

DIAGNOSTICS

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 *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*.

There are two nomenclatures 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.

CAVEAT

If the calling function alters the pointer to the root, results are unpredictable.

NAME

ttyname, isatty – find name of a terminal

SYNOPSIS

char *ttyname (fildes)
int fildes;

int isatty (fildes)
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.

FILES

/dev/*

DIAGNOSTICS

ttyname returns a NULL pointer if *fildes* does not describe a terminal device in directory **/dev**.

CAVEAT

The return value points to static data whose content is overwritten by each call.

NAME

ttynam, isatty – find name of a terminal port

SYNOPSIS

character*(*) function ttynam (lunit)

logical function isatty (lunit)

DESCRIPTION

Ttynam returns a blank padded path name of the terminal device associated with logical unit *lunit*.

Isatty returns **.true.** if *lunit* is associated with a terminal device, **.false.** otherwise.

FILES

/dev/*
/usr/lib/libU77.a

DIAGNOSTICS

Ttynam returns an empty string (all blanks) if *lunit* is not associated with a terminal device in directory '/dev'.

NAME

ttyslot – find the slot in the utmp file of the current user

SYNOPSIS

int ttyslot ( )

DESCRIPTION

ttyslot returns the index of the current user's entry in the **/etc/utmp** file. This is accomplished by actually scanning the file **/etc/inittab** for the name of the terminal associated with the standard input, the standard output, or the error output (0, 1 or 2).

FILES

/etc/inittab
/etc/utmp

SEE ALSO

getut(3C), ttyname(3C).

DIAGNOSTICS

A value of 0 is returned if an error was encountered while searching for the terminal name or if none of the above file descriptors is associated with a terminal device.

NAME

handle_unaligned_traps, print_unaligned_summary – gather statistics on unaligned references

SYNOPSIS

```
void handle_unaligned_traps()

void print_unaligned_summary()

long unaligned_load_word(addr)
char *addr;

long unaligned_load_half(addr)
char *addr;

long unaligned_load_uhalf(addr)
char *addr;

float unaligned_load_float(addr)
char *addr;

double unaligned_load_double(addr)
char *addr;

void unaligned_store_word(addr, value)
char *addr;
long value;

void unaligned_store_half(addr, value)
char *addr;
long value;

void unaligned_store_float(addr, float value)
char *addr;
float value;

void unaligned_store_double(addr, value)
char *addr;
double value;
```

DESCRIPTION

The first two routines implement a facility for finding unaligned references. The MIPS hardware traps load and store operations where the address is not a multiple of the number of bytes loaded or stored. Usually this trap indicates incorrect program operation and so by default the kernel converts this trap into a SIGBUS signal to the process, typically causing a core dump for debugging.

Older programs developed on systems with lax alignment constraints sometimes make occasional misaligned references in course of correct operation. The best way to port such programs to MIPS hardware is to correct the program by aligning the data.

A call to *handle_unaligned_traps* installs a SIGBUS handler that fixes unaligned memory references and keeps a record of the types, counts, and instruction addresses of these traps. A call to *print_unaligned_summary* prints the accumulated information. The following is an example of the output produced by *print_unaligned_summary*:

```
################################
#       unaligned reference summary            #
# byte aligned lw       5000  33.3%             #
# byte aligned sw      10000  66.7%             #
# 0x0040024c/i          5000  33.3%  33.3%      #
# 0x004002a8/i          5000  33.3%  66.7%      #
# 0x004002b4/i          5000  33.3% 100.0%      #
################################
```

The listing is written to standard error and describes the type and number of unaligned references, followed by a list of every address that contains an unaligned reference. To convert the addresses into a *dbx(1)* script and run the script, pipe the output (both standard output and standard error) through the following command. The output from **dbx** will be the name of the function and line number of the misalignment.

```
sed -n -e 's;^ # [0-9a-f]*/i).*#$;1;p' | dbx prog
```

This information can be used to decide the best way to correct the problem. If not all of the data can be aligned, or not all of the identified program locations that reference unaligned data can be changed, the *sysmips*(2) [MIPS_FIXADE] system call may be appropriate.

The other routines load or store their indicated data type at the address specified. The address need not meet the normal alignment constraints.

There exist fortran entry points for these routines so they may be called directly from fortran with the names documented here.

DIAGNOSTICS

If these routines try to load or store to an address that is outside the program's address space a SIGSEGV signal will be generated from inside these routines. If the program did not use these routines and the address was unaligned then the program would generate a SIGBUS signal. This is because the check for alignment is done before the address is checked to be in the program's address space.

SEE ALSO

dbx(1), sysmips(2) [MIPS_FIXADE], signal(2), sigset(2).

NAME

ungetc – push character back into input stream

SYNOPSIS

```
#include <stdio.h>

int ungetc (c, stream)
int c;
FILE *stream;
```

DESCRIPTION

ungetc inserts the character *c* into the buffer associated with an input *stream*. That character, *c*, will be returned by the next *getc*(3S) call on that *stream*. *ungetc* returns *c*, and leaves the file *stream* unchanged.

One character of pushback is guaranteed, provided something has already been read from the stream and the stream is actually buffered.

If *c* equals **EOF**, *ungetc* does nothing to the buffer and returns **EOF**.

Fseek(3S) erases all memory of inserted characters.

SEE ALSO

fseek(3S), getc(3S), setbuf(3S), stdio(3S).

DIAGNOSTICS

ungetc returns **EOF** if it cannot insert the character.

BUGS

When *stream* is *stdin*, one character may be pushed back onto the buffer without a previous read statement.

NAME

unlink – remove a directory entry

SYNOPSIS

integer function unlink (name)
character*(*) name

DESCRIPTION

Unlink causes the directory entry specified by pathname *name* to be removed. If this was the last link to the file, the contents of the file are lost. The returned value will be zero if successful; a system error code otherwise.

FILES

/usr/lib/libU77.a

SEE ALSO

unlink(2), link(3F), filsys(5), perror(3F)

BUGS

Pathnames can be no longer than MAXPATHLEN as defined in *<sys/param.h>*.

NAME

vprintf, vfprintf, vsprintf – print formatted output of a varargs argument list

SYNOPSIS

```
#include <stdio.h>
#include <varargs.h>

int vprintf (format, ap)
char *format;
va_list ap;

int vfprintf (stream, format, ap)
FILE *stream;
char *format;
va_list ap;

int vsprintf (s, format, ap)
char *s, *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 the use of *vfprintf* to write an error routine.

```
#include <stdio.h>
#include <varargs.h>
        .
        .
        .
/*
 *      error should be called like
 *       error(function_name, format, arg1, arg2...);  */
/*VARARGS */
void
error(va_alist)
/* Note that the function_name and format arguments cannot be
 *    separately declared because of the definition of varargs.  */
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

printf(3S), varargs(5).

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_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_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_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_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 discriminated unions 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

SEE ALSO

External Data Representation Protocol Specification, in *Networking on the Sun Workstation*.

NAME

verdixlib – MIPS-supported Ada library packages

SYNOPSIS

verdixlib

DESCRIPTION

verdixlib contains the packages MATH, COMPLEX_ARITH, ORDERING, COMMAND_LINE, and UNIX_CALLS. MATH uses the UNIX C mathematics llibrary to provide most standard mathematical functions and many constants. COMPLEX_ARITH defines the private type type COMPLEX and provides arithmetic functions for complex numbers. ORDERING includes sorting packages (QUICKSORT, HEAPSORT, and INSERTIONSORT) and a permuting package (PERMUTE).

COMMAND_LINE lets the user access the command line arguments and environments variables of an Ada program. UNIX_CALLS provides an interface to commonly used UNIX system calls.

TYPES AND FUNCTIONS

private type COMPLEX in COMPLEX_ARITH

FILES

*/usr/vads5/verdislib/**

SEE ALSO

MATH fully describes the MATH and COMPLEX_ARITH packages. Other libraries of Ada programs are *standard*, *publiclib*, and *examples*.

NAME

a.out – assembler and link editor output

SYNOPSIS

#include <a.out.h>

DESCRIPTION

A.out is the output file format of the assembler *as*(1) and the link editor *ld*(1). Both programs make *a.out* executable if there were no errors and no unresolved external references. The debugger uses the *a.out* file to provide symbolic information to the user.

The MIPS compilers and operating systems use a file format that is similar to standard AT&T System V COFF (common object file format). For more information, see the *MIPS Assembly Language Programmer's Guide.*

The MIPS File Header definition is based on the AT&T System V header file *filehdr.h* with the following changes (also see *filehdr*(4)):

- The symbol table file pointer, *f_symptr,* and the number of symbol table entries, *f_nsyms,* now specify the file pointer and the size of the Symbolic Header respectively.
- All tables that specify symbolic information have their file pointers and number of entries in the Symbolic Header.

The Optional Header definition has the same format as the AT&T System V header file *aouthdr.h* (the "standard" (pre-COFF) UNIX system a.out header) except the following fields have been added: *bss_start, gprmask, cprmask,* and *gp_value.*

The Section Header definition has the same format as the AT&T System V header file *scnhdr.h,* except the line number fields (*s_lnnoptr* and *s_nlnno*) are used for gp tables (see *scnhdr*(4)).

The MIPS relocation information definition is similar to that in Berkeley 4.3 UNIX, which has "local" relocation types (see *reloc*(4)). Also see the section entitled "Section Relocation Information" in the chapter 10 of the *MIPS Assembly Language Programmer's Guide* for the most detailed information.

For more information about AT&T System V COFF, refer to the AT&T UNIX System V Support Tools Guide.

The MIPS file format follows this scheme:

- File Header
- Optional Header
- Section Headers
- Section Data–includes text, read-only data, large data, 8 and 4 byte literal pools, small data, small bss (0 size), and large bss (0 size). As well as the shared library information.
- Section Relocation Information–includes information for text, read-only data, large data, 8 and 4 byte literal pools, and small data.
- Gp tables–missing if relocation information is not saved.
- Symbolic Header–missing if fully stripped.
- Line Numbers–created only if debugging is on, and missing if stripped of non-globals or fully stripped.

- Procedure Descriptor Table–missing if fully stripped.
- Local Symbols–missing if stripped of non-globals or if fully stripped.
- Optimization Symbols–created only if debugging is on, and missing if stripped of nonglobals or fully stripped.
- Auxiliary Symbols–created only if debugging is on, and missing if stripped of nonglobals or fully stripped.
- Local Strings–missing if stripped of non-globals or if fully stripped.
- External Strings–missing if fully stripped.
- Relative File Descriptors–missing if stripped of non-globals or if fully stripped.
- File Descriptors–missing if stripped of non-globals or if fully stripped.
- External Symbols–missing is fully stripped.

SEE ALSO

as(1), ld(1), nm(1), dbx(1), strip(1), filehdr(4), scnhdr(4), reloc(4), syms(4), linenum(4).

NAME

acct – per-process accounting file format

SYNOPSIS

#include <sys/acct.h>

DESCRIPTION

Files produced as a result of calling *acct*(2) have records in the form defined by **<sys/acct.h>**, whose contents are:

```
typedef  ushort comp_t;        /* "floating point" */
                /* 13-bit fraction, 3-bit exponent  */

struct   acct
{
        char    ac_flag;       /* Accounting flag */
        char    ac_stat;       /* Exit status */
        ushort  ac_uid;
        ushort  ac_gid;
        dev_t   ac_tty;
        time_t  ac_btime;      /* Beginning time */
        comp_t  ac_utime;      /* acctng user time in clock ticks */
        comp_t  ac_stime;      /* acctng system time in clock ticks */
        comp_t  ac_etime;      /* acctng elapsed time in clock ticks */
        comp_t  ac_mem;        /* memory usage in clicks */
        comp_t  ac_io;         /* chars trnsfrd by read/write */
        comp_t  ac_rw;         /* number of block reads/writes */
        char    ac_comm[8];    /* command name */
};

extern  struct  acct        acctbuf;
extern  struct  inode       *acctp;  /* inode of accounting file */

#define ACCTF   0300        /* record type: 00 = acct */
```

In *ac_flag*, the AFORK flag is turned on by each *fork*(2) and turned off by an *exec*(2). The *ac_comm* field is inherited from the parent process and is reset by any *exec*. Each time the system charges the process with a clock tick, it also adds to *ac_mem* the current process size, computed as follows:

(data size) + (text size) / (number of in-core processes using text)

The value of *ac_mem / (ac_stime + ac_utime)* can be viewed as an approximation to the mean process size, as modified by text-sharing.

The structure **tacct.h**, which resides with the source files of the accounting commands, represents the total accounting format used by the various accounting commands:

```
/*
 * total accounting (for acct period), also for day
 */

struct tacct {
        uid_t           ta_uid;     /* userid */
        char            ta_name[8]; /* login name */
        float           ta_cpu[2];  /* cum. cpu time, p/np (mins) */
        float           ta_kcore[2]; /* cum kcore-minutes, p/np */
        float           ta_con[2];  /* cum. connect time, p/np, mins */
        float           ta_du;      /* cum. disk usage */
        long            ta_pc;      /* count of processes */
        unsigned short  ta_sc;      /* count of login sessions */
        unsigned short  ta_dc;      /* count of disk samples */
        unsigned short  ta_fee;     /* fee for special services */
};
```

SEE ALSO

acct(2), exec(2), fork(2).

ERRORS

The *ac_mem* value for a short-lived command gives little information about the actual size of the command, because *ac_mem* may be incremented while a different command (e.g., the shell) is being executed by the process.

NAME

aliases – aliases file for sendmail

SYNOPSIS

/usr/lib/aliases

DESCRIPTION

This file describes user id aliases used by */usr/lib/sendmail*. It is formatted as a series of lines of the form

name: name_1, name2, name_3, . . .

The *name* is the name to alias, and the *name_n* are the aliases for that name. Lines beginning with white space are continuation lines. Lines beginning with '#' are comments.

Aliasing occurs only on local names. Loops can not occur, since no message will be sent to any person more than once.

Aliasing can be prevented by escaping the first character with a backslash (\).

After aliasing has been done, local and valid recipients who have a ".forward" file in their home directory have messages forwarded to the list of users defined in that file (see *forward(4)* for details).

This is only the raw data file; the actual aliasing information is placed into a binary format in the files */usr/lib/aliases.dir* and */usr/lib/aliases.pag* using the program *newaliases*(1). A *newaliases* command should be executed each time the aliases file is changed for the change to take effect.

SEE ALSO

newaliases(1), dbm(3X), forward(4), sendmail(1M)
SENDMAIL Installation and Operation Guide.
SENDMAIL An Internetwork Mail Router.

ERRORS

Because of restrictions in *dbm*(3X) a single alias cannot contain more than about 1000 bytes of information. You can get longer aliases by "chaining"; that is, make the last name in the alias be a dummy name which is a continuation alias.

NAME

ar – archive (library) file format

SYNOPSIS

#include <ar.h>

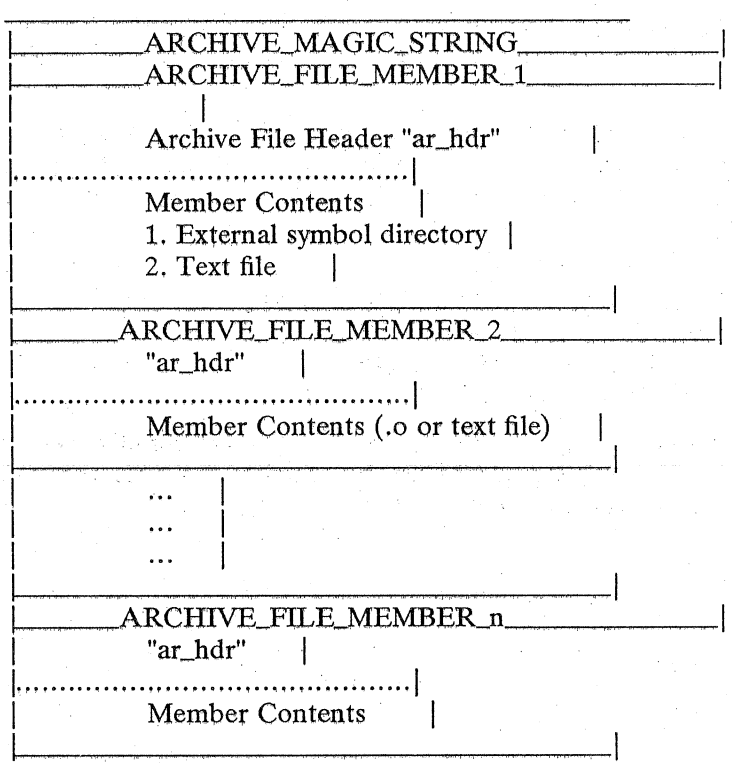
DESCRIPTION

The archive command *ar* combines several files into one. Archives are used mainly as libraries to be searched by the link-editor *ld*.

A file produced by *ar* has a magic string at the start, followed by the constituent files, each preceded by a file header. The magic number and header layout as described in the include file are:

COMMON ARCHIVE FORMAT

ARCHIVE File Organization:



The name is a blank-padded string. The *ar_fmag* field contains ARFMAG to help verify the presence of a header. The other fields are left-adjusted, blank-padded numbers. They are decimal except for *ar_mode*, which is octal. The date is the modification date of the file at the time of its insertion into the archive.

Each file begins on a even (0 mod 2) boundary; a new-line is inserted between files if necessary. Nevertheless the size given reflects the actual size of the file exclusive of padding.

There is no provision for empty areas in an archive file.

The encoding of the header is portable across machines. If an archive contains printable files, the archive itself is printable.

SEE ALSO

ar(1), ld(1), nm(1)

BUGS

File names lose trailing blanks. Most software dealing with archives takes even an included blank as a name terminator.

NAME

checklist – list of file systems processed by fsck.s51k and ncheck.s51k

DESCRIPTION

checklist resides in directory **/etc** and contains a list of, at most, 15 *special file* names. Each *special file* name is contained on a separate line and corresponds to a file system. Each file system will then be automatically processed by the *fsck.s51k*(1M) command.

FILES

/etc/checklist

SEE ALSO

fsck.s51k(1M), ncheck.s51k(1M) in the *System Administrator's Reference Manual.*

NAME

core – format of memory image file

SYNOPSIS

#include <sys/param.h>

DESCRIPTION

The UNIX System writes out a memory image of a terminated process when any of various errors occur. See *sigvec*(2) for the list of reasons; the most common are memory violations, illegal instructions, bus errors, and user-generated quit signals. The memory image is called 'core' and is written in the process's working directory (provided it can be; normal access controls apply).

The maximum size of a *core* file is limited by *setrlimit*(2). Files which would be larger than the limit are not created.

The core file consists of the *u.* area, whose size (in pages) is defined by the UPAGES manifest in the *<sys/param.h>* file. The *u.* area starts with a *user* structure as given in *<sys/user.h>*. The remainder of the core file consists first of the data pages and then the stack pages of the process image. The amount of data space image in the core file is given (in pages) by the variable *u_dsize* in the *u.* area. The amount of stack image in the core file is given (in pages) by the variable *u_ssize* in the *u.* area. The size of a "page" is given by the constant NBPG (also from *<sys/param.h>*).

In general the debugger *dbx*(1) is sufficient to deal with core images.

SEE ALSO

dbx(1), sigvec(2), setrlimit(2)

NAME

cpio – format of cpio archive

DESCRIPTION

The *header* structure, when the **–c** option of *cpio*(1) is not used, is:

```
struct {
        short   h_magic,
                h_dev;
        ushort  h_ino,
                h_mode,
                h_uid,
                h_gid;
        short   h_nlink,
                h_rdev,
                h_mtime[2],
                h_namesize,
                h_filesize[2];
        char    h_name[h_namesize rounded to word];
} Hdr;
```

When the **–c** option is used, the *header* information is described by:

```
sscanf(Chdr,"%6o%6o%6o%6o%6o%6o%6o%6o%11lo%6o%11lo%s",
        &Hdr.h_magic, &Hdr.h_dev, &Hdr.h_ino, &Hdr.h_mode,
        &Hdr.h_uid, &Hdr.h_gid, &Hdr.h_nlink, &Hdr.h_rdev,
        &Longtime, &Hdr.h_namesize,&Longfile,Hdr.h_name);
```

Longtime and *Longfile* are equivalent to *Hdr.h_mtime* and *Hdr.h_filesize*, respectively. The contents of each file are recorded in an element of the array of varying length structures, *archive*, together with other items describing the file. Every instance of *h_magic* contains the constant 070707 (octal). The *h_dev* and *h_inode* values combine to make one unsigned 32-bit number, rather than two shorts. It is a number created by **cpio** to uniquely identify linked files. The *h_dev* contains the high-order 16 bits of the 32-bit number, and *h_inode* contains the low-order 16 bits of the 32-bit number. This number does not reflect the actual device/inode pair of the file. The first number assigned by **cpio** is 3, and is sequentially incremented for each file processed by **cpio**. The items *h_mode* through *h_mtime* have meanings explained in *stat*(2). The length of the null-terminated path name *h_name*, including the null byte, is given by *h_namesize*.

The last record of the *archive* always contains the name TRAILER!!!. Special files, directories, and the trailer are recorded with *h_filesize* equal to zero.

SEE ALSO

stat(2).
cpio(1), find(1) in the *User's Reference Manual*.

NAME

DEV_DB – device description database

DESCRIPTION

The directory */dev/DEV_DB* contains a set of files that make up the database used by *MKDEV(1M)* to create system device entries. Note that **MKDEV** uses the subdirectory *./DEV_DB*, so that if **MKDEV** is run in another directory, the database must be there.

The database is the concatenation of the following files in the given order:

common.system
common.local
common.*hostname*
machine.system
machine.local
machine.hostname

where *hostname* is found by executing the command *hostname(1)* (if it exists) and *machine* is found by executing *uname(1)* with the option **–t**. Data definitions are overwritten, so data found in common.system can be overridden by data found in, for example, *machine*.system.

The intent of this organization is to make network and host-specific administration simpler. The **.system** files are supplied with the system. The **.local** and *.hostname* files can be maintained on a single "master" system and then copied periodically to the other systems, thus centralizing the administration task.

The database is a simple hierarchy of device "classes" and "operations". A class is defined by the syntax

class(*name,alias...*) {*operations*}

(NOTE: Separators, such as parentheses and commas, may be surrounded by spaces and tabs, and the braces may be surrounded by spaces, tabs, and newlines.) Class names and aliases may not contain spaces, tabs, newlines, parentheses, braces, or the comment character (#). Comments are any string beginning with a # and ending with a newline. There is no way to escape the # character.

As stated previously, the same class may be defined multiple times, and each definition overrides the previous, thus allowing the machine-specific and local files to override the system defaults.

The *operations* part of the class definition consists of listings for devices, "iterative" devices, links, "iterative" links, messages, and other classes. Listings are separated by newlines or semi-colons, and trailing semi-colons are ignored.

A **device** listing is of the form

device(*name,type,major,minor,mode,owner,group*)

This creates a device entry called *name*, with type *type* ('b' for block special or 'c' for character special), the given major and minor device numbers, the given mode (number), and the named owner and group.

An iterative device is a group of devices whose names and minor numbers are related and have values that increase by 1. An example of an iterative device is the set of standard terminals, /dev/tty[0-5], which have minor device numers 0-5. The iterative device listing is of the form

idevice(*count,name,start,type,major,minor,mode,owner,group*)

This creates a set of *count* device entries. The name of each entry is the concatenation of *name* and a counter beginning at *start* increasing by 1. The minor device number starts with *minor* and increases by 1 for each entry.

A **link** listing is of the form

link(*file,linkname*)

This creates a link called *linkname* to *file*, which must already exist.

An iterative link is a group of links whose source and target names both end with numbers, and each number is incremented by 1. The iterative link listing is of the form

ilink(*count,file,start1,linkname,start2*)

This creates a set of *count* device entries. The name of each entry is the concatenation of *linkname* and a counter beginning at *start2*. The file to be linked in each case is the concatenation of *file* and a counter beginning at *start1*.

A **message** listing causes a message to be printed on the standard output when **MKDEV** is executed. It is of the form

message(*text*)

The *text* field may contain any characters except for separators (parentheses, commas, semicolons, and braces), and is processed by *echo(1)*. Leading and trailing spaces and tabs are removed, and intermediate whitespace may be compressed into single spaces. It is therefore best to use \t for tabs.

All other listings found in a class definition are considered to be the names of classes, which are effectively included. For example, if class "foo" is defined with a listing "bar", "bar" is considered to be a class, and its listings are included in "foo".

It is a good idea to execute **MKDEV** with the **-n** option to check the syntax of the database files after any changes.

EXAMPLES

The following defines the class of terminals, which consists of /dev/tty, /dev/ttyh0 through /dev/ttyh15, /dev/ttyi0 through /dev/ttyi5, /dev/tty0 through /dev/tty5, /dev/ttym0 through /dev/ttym5, and the special links to the console.

```
class(tty,terminals) {
        device(tty, c, 2, 0, 622, root, bin)
        idevice(16, ttyh, 0, c, 16, 0, 622, root, bin)
        idevice(16, ttyi, 0, c, 16, 16, 622, root, bin)
        idevice(6, tty, 0, c, 0, 0, 622, root, bin)
        idevice(6, ttym, 0, c, 0, 64, 622, root, bin)
        console
}

class(console) {
        link(tty0, console)
        link(tty0, systty)
        link(tty0, syscon)
}
```

Executing the command "MKDEV tty" or "MKDEV terminals" will create all of these files and links, while executing "MKDEV console" will only create links to /dev/tty0 if it exists.

FILES

/dev/DEV_DB Device database directory

/dev/DEV_DB/*.system
Mips-defined device database entries

/dev/DEV_DB/*.local
Locally-defined device database entries

/dev/DEV_DB/*.*hostname*
Host-specific device database entries

SEE ALSO

MKDEV(1M).

BUGS

The syntax checking, especially with regard to commas, is not very thorough, and the syntax error messages are vague.

NAME

dir – format of directories

SYNOPSIS

#include <sys/fs/s5dir.k>

DESCRIPTION

Note: *The obsolete s51k file system has been kept for backward compatibility. The Fast File System (FFS) is preferred. See FS(4ffs).*

A directory behaves exactly like an ordinary file, save that no user may write into a directory. The fact that a file is a directory is indicated by a bit in the flag word of its i-node entry [see *fs*(4)]. The structure of a directory entry for an s51k file system is as follows:

```
#ifndef DIRSIZ
#define DIRSIZ      14
#endif
struct   direct
{
        ushort d_ino;
        char   d_name[DIRSIZ];
};
```

By convention, the first two entries in each directory are for . and .. The first is an entry for the directory itself. The second is for the parent directory. The meaning of .. is modified for the root directory of the master file system; there is no parent, so .. has the same meaning as ..

SEE ALSO

dir(4ffs), fs(4s51k), fs(4ffs).

NAME

dir – format of directories

SYNOPSIS

#include <sys/types.h>
#include <sys/fs/ffs_dir.h>

DESCRIPTION

A directory behaves exactly like an ordinary file, save that no user may write into a directory. The fact that a file is a directory is indicated by a bit in the flag word of its i-node entry; see *fs*(4). The structure of a directory entry for an FFS file system is as follows:

```
/*
 * A directory consists of some number of blocks of DIRBLKSIZ
 * bytes, where DIRBLKSIZ is chosen such that it can be transferred
 * to disk in a single atomic operation (e.g. 512 bytes on most machines).
 *
 * Each DIRBLKSIZ byte block contains some number of directory entry
 * structures, which are of variable length.  Each directory entry has
 * a struct direct at the front of it, containing its inode number,
 * the length of the entry, and the length of the name contained in
 * the entry.  These are followed by the name padded to a 4 byte boundary
 * with null bytes.  All names are guaranteed null terminated.
 * The maximum length of a name in a directory is MAXNAMLEN.
 *
 * The macro DIRSIZ(dp) gives the amount of space required to represent
 * a directory entry.  Free space in a directory is represented by
 * entries which have dp->d_reclen > DIRSIZ(dp).  All DIRBLKSIZ bytes
 * in a directory block are claimed by the directory entries.  This
 * usually results in the last entry in a directory having a large
 * dp->d_reclen.  When entries are deleted from a directory, the
 * space is returned to the previous entry in the same directory
 * block by increasing its dp->d_reclen.  If the first entry of
 * a directory block is free, then its dp->d_ino is set to 0.
 * Entries other than the first in a directory do not normally have
 * dp->d_ino set to 0.
 */
#define BFS_DIRBLKSIZ 512
#endif

#define BFS_MAXNAMLEN 255

/*
 * The BFS_DIRSIZ macro gives the minimum record length which will hold
 * the directory entry.  This requires the amount of space in struct direct
 * without the d_name field, plus enough space for the name with a terminating
 * null byte (dp->d_namlen+1), rounded up to a 4 byte boundary.
 */
#undef BFS_DIRSIZ
#define BFS_DIRSIZ(dp) \
    ((sizeof (struct bfs_direct) - (MAXNAMLEN+1)) + (((dp)->d_namlen+1 + 3) &~ 3))

struct    bfs_direct {
        u_long      d_ino;
```

```
        short       d_reclen;
        short       d_namlen;
        char        d_name[MAXNAMLEN + 1];
        /* typically shorter */
};
```

By convention, the first two entries in each directory are for '.' and '..'. The first is an entry for the directory itself. The second is for the parent directory. The meaning of '..' is modified for the root directory of the master file system ("/"), where '..' has the same meaning as '.'.

SEE ALSO

fs(4FFS)

NAME

dirent – file system independent directory entry

SYNOPSIS

#include <sys/types.h>
#include <sys/dirent.h>

DESCRIPTION

Different file system types may have different directory entries. The *dirent* structure defines a file system independent directory entry, which contains information common to directory entries in different file system types. A set of these structures is returned by the *getdents*(2) system call.

The *dirent* structure is defined below.

```
struct   dirent {
                    long            d_ino;
                    off_t           d_off;
                    unsigned short  d_reclen;
                    char            d_name[1];
                };
```

The *d_ino* is a number which is unique for each file in the file system. The field *d_off* is the offset of that directory entry in the actual file system directory. The field *d_name* is the beginning of the character array giving the name of the directory entry. This name is null terminated and may have at most MAXNAMLEN characters. This results in file system independent directory entries being variable length entities. The value of *d_reclen* is the record length of this entry. This length is defined to be the number of bytes between the current entry and the next one, so that it will always result in the next entry being on a long boundary.

FILES

/usr/include/sys/dirent.h

SEE ALSO

getdents(2).

NAME

dvh – format of MIPS disk volume header

SYNTAX

#include <sys/dvh.h>

DESCRIPTION

Disk volumes on MIPS computers systems contain a volume header that describes the contents of the disk and parameters of the physical disk drive. Volume headers are created by *format*(8), and manipulated by *dvhtool*(1M). The MIPS PROM MONITOR reads disk volume headers to determine the appropriate file to boot on autobooting.

The volume header is a block located at the beginning of all disk media. It contains information pertaining to physical device parameters and logical partition information. The volume header is manipulated by disk formatters/verifiers, partition builders (e.g. newfs/mkfs), and device drivers. A copy of the volume header is located at sector 0 of each track of cylinder 0. The volume header is constrained to be less than 512 bytes long. A particular copy is assumed valid if no drive errors are detected, the magic number is correct, and the 32 bit 2's complement of the volume header is correct. The checksum is calculated by initially zeroing vh_csum, 2's complement summing the entire structure and then storing the 2's complement of the sum. Thus a resumming a previously checksum'ed header should yield 0 to verify the volume header.

The error summary table, bad sector replacement table, and boot blocks are located by searching the volume directory within the volume header. Tables are sized simply by the integral number of table records that will fit in the space indicated by the directory entry. The amount of space allocated to the volume header, replacement blocks, and other tables is user defined when the device is formatted.

Device parameters are in the volume header to determine mapping from logical block numbers to physical device addresses, allow the driver to properly configure itself and the controller for the given disk drive, and to allow the operating system to know various parameters (such as transfer rate) of the disk system.

The partition table describes logical device partitions (device drivers examine this to determine mapping from logical units to cylinder groups, device formatters/verifiers examine this to determine location of replacement tracks/sectors, etc). NOTE: the field pt_firstlbn should be cylinder aligned.

The error table records media defects, and allows for "automatic" replacement of bad blocks and more informative error logging.

The bad sector table is used to map from bad sectors/tracks to replacement sector/tracks. To identify available replacement sectors/tracks, allocate replacements in increasing block number from a replacement partition. When a new replacement sector/track is needed scan the bad sector table to determine current highest replacement sector/track block number and then scan the device from the next block until a defect free replacement sector/track is found or the end of replacement partition is reached. If bt_rpltype == BSTTYPE_TRKFWD, then bt_badlbn refers to the bad logical block within the bad track, and bt_rpllbn refers to the first sector of the replacement track. If bt_rpltype == BSTTYPE_SLIPSEC or bt_rpltype == BSTTYPE_SLIPBAD, then bt_rpllbn has no meaning.

The format of the *dvh* disk volume header is:

```
struct device_parameters {
        u_char  dp_skew;        /* spiral addressing skew */
        u_char  dp_gap1;        /* words of 0 before header */
        u_char  dp_gap2;        /* words of 0 between hdr and data */
```

```
        u_char  dp_spare0;      /* spare space */
        u_short dp_cyls;        /* number of cylinders */
        u_short dp_shd0;        /* starting head vol 0 */
        u_short dp_trks0;       /* number of tracks vol 0 */
        u_short dp_shd1;        /* starting head vol 1 */
        u_short dp_trks1;       /* number of tracks vol 1 */
        u_short dp_secs;        /* number of sectors/track */
        u_short dp_secbytes;    /* length of sector in bytes */
        u_short dp_interleave;  /* sector interleave */
        int     dp_flags;       /* controller characteristics */
        int     dp_datarate;    /* bytes/sec for kernel stats */
        int     dp_nretries;    /* max num retries on data error */
        int     dp_spare1;      /* spare entries */
        int     dp_spare2;
        int     dp_spare3;
        int     dp_spare4;
};

/*
 * Device characterization flags
 * (dp_flags)
 */
#define DP_SECTSLIP             0x00000001 /* sector slip to spare sector */
#define DP_SECTFWD              0x00000002 /* forward to replacement sector */
#define DP_TRKFWD               0x00000004 /* forward to replacement track */
#define DP_MULTIVOL             0x00000008 /* multiple volumes per spindle */
#define DP_IGNOREERRORS         0x00000010 /* transfer data regardless of errors */
#define DP_RESEEK               0x00000020 /* recalibrate as last resort */

#define VDNAMESIZE 8

struct volume_directory {
        char    vd_name[VDNAMESIZE]; /* name */
        int     vd_lbn;         /* logical block number */
        int     vd_nbytes;      /* file length in bytes */
};

struct partition_table {  /* one per logical partition */
        int     pt_nblks;       /* # of logical blks in partition */
        int     pt_firstlbn;    /* first lbn of partition */
        int     pt_type;        /* use of partition */
};

#define PTYPE_VOLHDR    0       /* partition is volume header */
#define PTYPE_TRKREPL   1       /* partition is used for repl trks */
#define PTYPE_SECREPL   2       /* partition is used for repl secs */
#define PTYPE_RAW       3       /* partition is used for data */
#define PTYPE_BSD42     4       /* partition is 4.2BSD file system */
#define PTYPE_SYSV      5       /* partition is SysV file system */
#define PTYPE_VOLUME    6       /* partition is entire volume */

#define VHMAGIC                 0xbe5a941 /* randomly chosen value */
```

```
#define NPARTAB         16      /* 16 unix partitions */
#define NVDIR           15      /* max of 15 directory entries */
#define BFNAMESIZE 16   /* max 16 chars in boot file name */

struct volume_header {
        int vh_magic;                   /* identifies volume header */
        short vh_rootpt;                /* root partition number */
        short vh_swappt;                /* swap partition number */
        char vh_bootfile[BFNAMESIZE];   /* name of file to boot */
        struct device_parameters vh_dp;         /* device parameters */
        struct volume_directory vh_vd[NVDIR]; /* other vol hdr contents */
        struct partition_table vh_pt[NPARTAB]; /* device partition layout */
        int vh_csum;                    /* volume header checksum */
};

#define ERR_SECC        0       /* soft ecc */
#define ERR_HECC        1       /* hard ecc */
#define ERR_HCSUM       2       /* header checksum */
#define ERR_SOTHER      3       /* any other soft errors */
#define ERR_HOTHER      4       /* any other hard errors */
#define NERRTYPES       5       /* Total number of error types */

struct error_table {            /* one per defective logical block */
        int     et_lbn;         /* defective block number */
        int     et_errcount[NERRTYPES];     /* counts for each error type */
};

struct bst_table {
        int     bt_badlbn;      /* bad logical block */
        int     bt_rpllbn;      /* replacement logical block */
        int     bt_rpltype;     /* replacement method */
};

/*
 * replacement types
 */
#define BSTTYPE_EMPTY       0       /* slot unused */
#define BSTTYPE_SLIPSEC     1       /* sector slipped to next sector */
#define BSTTYPE_SECFWD      2       /* sector forwarded to replacment sector */
#define BSTTYPE_TRKFWD      3       /* track forwarded to replacement track */
#define BSTTYPE_SLIPBAD     4       /* sector reserved for slipping has defect */

/*
 * The following structs are parameters to various driver ioctls
 * for disk formatting, etc.
 */

/*
 * controller information struct
 * returned via DIOCGETCTLR
 * mostly to determine appropriate method for bad block handling
 */
```

```
#define CITYPESIZE 32

struct ctlr_info {
        int     ci_flags;               /* same as DP_* flags */
        char    ci_type[CITYPESIZE];    /* controller model and manuf. */
};

/*
 * verify sectors information
 * Passed to device driver via ioctl DIOCVFYSEC
 */
struct verify_info {
        int     vi_lbn;         /* logical block number */
        int     vi_bcnt;        /* logical block count */
};

/*
 * cause controller to run diagnostics
 */
struct diag_info {
        int     di_errcode;     /* error code */
        int     di_lbn;         /* logical block number */
        int     di_bcnt;        /* logical block count */
        char    *di_addr;       /* buffer address */
};

/*
 * information necessary to perform one of the following actions:
 *      format a track
 *         fmi_cyl and fmi_trk identify track to format
 *      map a track
 *         fmi_cyl and fmi_trk identify defective track
 *         fmi_rplcyl and fmi_rpltrk identify replacement track
 *      map a sector
 *         fmi_cyl, fmi_trk, and fmi_sec identify defective sector
 *         fmi_rplcyl, fmi_rpltrk, and fmi_rplsec identify
 *              replacement sector
 *      slip a sector
 *         fmi_cyl, fmi_trk, and fmi_sec identify defective sector
 */
#define FMI_FORMAT_TRACK1       /* format a track */
#define FMI_MAP_TRACK           2       /* map a track */
#define FMI_MAP_SECTOR          3       /* map a sector */
#define FMI_SLIP_SECTOR         4       /* slip a sector */

struct fmt_map_info {
        int     fmi_action;     /* action desired, see FMI_ above */
        u_short fmi_cyl;        /* cylinder with defect or one with */
                                /* track to format */
        u_char  fmi_trk;        /* track with defect or one to format */
        u_char  fmi_sec;        /* sector with defect */
        u_short fmi_rplcyl;     /* replacement cylinder */
```

```
        u_char  fmi_rpltrk;     /* replacement track */
        u_char  fmi_rplsec;     /* replacement sector */
};
```

SEE ALSO

MIPS System Programmer's Guide
System Programmer's Package Reference

NAME

ethers – ethernet address to hostname database

DESCRIPTION

The *ethers* file contains information regarding the known (48 bit) ethernet addresses of hosts on the internet. For each host on an ethernet, a single line should be present with the following information:

ethernet-address
official host name

Items are separated by any number of blanks and/or tabs. A '#' indicates the beginning of a comment extending to the end of line.

The standard form for ethernet addresses is "x:x:x:x:x:x" where *x* is a hexadecimal number between 0 and ff, representing one byte. The address bytes are always in network order. Host names may contain any printable character other than a space, tab, newline, or comment character. It is intended that host names in the *ethers* file correspond to the host names in the *hosts*(4) file.

The *ether_line*() routine from the ethernet address manipulation library, *ethers*(3Y) may be used to scan lines of the *ethers* file.

FILES

/etc/ethers

SEE ALSO

ethers(3Y), hosts(4)

ORIGIN

Sun Microsystems

NAME

exports – NFS file systems being exported

SYNOPSIS

/etc/exports

DESCRIPTION

The file */etc/exports* describes the file systems which are being exported to *nfs*(4) clients. It is created by the system administrator using a text editor and processed by the *mount* request daemon *mountd*(1M) each time a mount request is received.

The file consists of a list of file systems, the *netgroup*(4) or machine names allowed to remote mount each file system, and possibly a list of options. The file system names are left justified and followed by a list of names separated by white space. The names will be looked up in */etc/netgroups* and then in */etc/hosts*. Options begin with a hyphen and are separated by commas. Currently *mountd* understands the following options:

ro Prevent clients from writing to this entry's filesystem; allow reading only.

rw Allow clients to both read and write this entry's filesystem.

hide Prevents a client who mounts an exported filesystem to access files in other exported filesystems that are mounted on directories under the mounted filesystem.

nohide Allows a client who mounts an exported filesystem to access files in other exported filesystems that are mounted on directories under the mounted filesystem.

rootid=*uid*
Translate credentials for client operations issued by *root* on a client to have effective user-id *uid*. *uid* may be either a name or an integer user-id from */etc/passwd*

The default options are **rw,hide,rootid=nobody**. A file system name with no name list following means export to everyone. A "#" anywhere in the file indicates a comment extending to the end of the line it appears on. Lines beginning with white space are continuation lines.

EXAMPLE

```
/usr      clients              # export to my clients
/usr/local                     # export to the world
/usr2     phoenix sun sundae   # export to only these machines
/usr3     -rootid=guest        # map client root to guest
/         -nohide,ro           # export all local filesystems read-only
```

FILES

/etc/exports

SEE ALSO

mountd(1M)

ORIGIN

Sun Microsystems

NAME

filehdr – file header for MIPS object files

SYNOPSIS

#include < filehdr.h>

DESCRIPTION

Every MIPS object file begins with a 20-byte header. The following C **struct** declaration is used:

```
struct   filehdr
{
        unsigned short   f_magic;   /* magic number */
        unsigned short   f_nscns;   /* number of sections */
        long             f_timdat;  /* time & date stamp */
        long             f_symptr;  /* file pointer to symbolic header */
        long             f_nsyms;   /* sizeof(symbolic header) */
        unsigned short   f_opthdr;  /* sizeof(optional header) */
        unsigned short   f_flags;   /* flags */
};
```

F_symptr is the byte offset into the file at which the symbolic header can be found. Its value can be used as the offset in *fseek*(3S) to position an I/O stream to the symbolic header. The UMIPS system optional header is 56-bytes. The valid magic numbers are given below:

```
#define MIPSEBMAGIC   0x0160  /* objects for MIPS big-endian machines */
#define MIPSELMAGIC   0x0162  /* objects for MIPS little-endian machines */
#define MIPSEBUMAGIC  0x0180  /* ucode objects for MIPS big-endian machines */
#define MIPSELUMAGIC  0x0182  /* ucode objects for MIPS little-endian machines */
```

MIPS object files can be loaded and examined on machines differing from the object's target byte sex. Therefore, for object file magic numbers, the byte swapped values have define constants associated with them:

```
#define SMIPSEBMAGIC  0x6001
#define SMIPSELMAGIC  0x6201
```

The value in *f_timdat* is obtained from the *time*(2) system call. Flag bits used in MIPS objects are:

```
#define F_RELFLG      0000001   /* relocation entries stripped */
#define F_EXEC        0000002   /* file is executable */
#define F_LNNO        0000004   /* line numbers stripped */
#define F_LSYMS       0000010   /* local symbols stripped */
```

SEE ALSO

time(2), fseek(3S), a.out(4).

NAME

forward – mail forwarding file

SYNOPSIS

$HOME/.forward

DESCRIPTION

When *sendmail(1M)* resolves mail addresses, it resolves aliases (see *aliases(4)*) and then forwards the mail using the contents of the file $HOME/.forward.

The forward file should contain a list of addresses, each of which results in having the mail forwarded. These may be separated by commas, whitespace, or newlines.

There are three types of forwarding addresses that are particularly useful:

address	A mail address sends the mail to that user. The address is subject to alias resolution (in other words, the forward file may contain aliases).
name	An escaped name is not subject to alias resolution. A typical use of escaped names is shown below.
"| *command ...*"	Execute the given *command* with the mail message as the standard input.

In general, forward files are used to set up mail forwarding in a local area network. Users tend to have a single "base machine" where mail is kept, and in this case the forward file is set up to forward mail to that machine.

One special use of the forward file involves automatic replies to senders while the recipient is on vacation. The forward file can be set up to contain the line

```
\user,"| vacation user"
```

where "user" is replaced by the name of the recipient. The command *vacation(1)* will send a specified message, usually one indicating when the recipient will return, to the sender.

Another special use of the forward file involves the command /usr/new/lib/mh/slocal, which can be set up to automatically file mail into folders, send replies based on message contents, place mail into files, ignore mail, and many other things. In this case, the forward file contains

```
"| /usr/new/lib/mh/slocal"
```

See the manual page *mhook(1)* for more information.

SEE ALSO

mhook(1), vacation(1), aliases(4), sendmail(1M).

NAME

fs: file system – format of s51k system volume

SYNOPSIS

#include <sys/filsys.h>
#include <sys/types.h>
#include <sys/param.h>

DESCRIPTION

Note: *The obsolete s51k file system has been kept for backward compatibility. The Fast File System (FFS) is preferred. See fs(4FFS).*

Every s51k file system storage volume has a common format for certain vital information. Every such volume is divided into a certain number of 512-byte long sectors. Sector 0 is unused and is available to contain a bootstrap program or other information.

Sector 1 is the *super-block*. The format of a super-block is:

```
struct   filsys
{
        ushort      s_isize;          /* size in blocks of i-list */
        daddr_t     s_fsize;          /* size in blocks of entire volume */
        short       s_nfree;          /* number of addresses in s_free */
        daddr_t     s_free[NICFREE];  /* free block list */
        short       s_ninode;         /* number of i-nodes in s_inode */
        ushort      s_inode[NICINOD]; /* free i-node list */
        char        s_flock;          /* lock during free list manipulation */
        char        s_ilock;          /* lock during i-list manipulation */
        char        s_fmod;           /* super block modified flag */
        char        s_ronly;          /* mounted read-only flag */
        time_t      s_time;           /* last super block update */
        short       s_dinfo[4];       /* device information */
        daddr_t     s_tfree;          /* total free blocks*/
        ushort      s_tinode;         /* total free i-nodes */
        char        s_fname[6];       /* file system name */
        char        s_fpack[6];       /* file system pack name */
        long        s_fill[12];       /* ADJUST to make sizeof filsys
                                         be 512 */
        long        s_state;          /* file system state */
        long        s_magic;          /* magic number to denote new
                                         file system */
        long        s_type;           /* type of new file system */
};

#define FsMAGIC     0xfd187e20        /* s_magic number */

#define Fs1b        1                 /* 512-byte block */
#define Fs2b        2                 /* 1024-byte block */

#define FsOKAY      0x7c269d38        /* s_state: clean */
#define FsACTIVE    0x5e72d81a        /* s_state: active */
#define FsBAD       0xcb096f43        /* s_state: bad root */
#define FsBADBLK    0xbadbc14b        /* s_state: bad block corrupted it */
```

s_type indicates the file system type. Currently, two types of file systems are supported: the original 512-byte logical block and the improved 1024-byte logical block. *s_magic* is used to distinguish the original 512-byte oriented file systems from the newer file systems. If this field is not equal to the magic number, *fsMAGIC*, the type is assumed to be *fs1b*, otherwise the *s_type* field is used. In the following description, a block is then determined by the type. For the original 512-byte oriented file system, a block is 512-bytes. For the 1024-byte oriented file system, a block is 1024-bytes or two sectors. The operating system takes care of all conversions from logical block numbers to physical sector numbers.

s_state indicates the state of the file system. A cleanly unmounted, not damaged file system is indicated by the FsOKAY state. After a file system has been mounted for update, the state changes to FsACTIVE. A special case is used for the root file system. If the root file system appears damaged at boot time, it is mounted but marked FsBAD. Lastly, after a file system has been unmounted, the state reverts to FsOKAY.

s_isize is the address of the first data block after the i-list; the i-list starts just after the super-block, namely in block 2; thus the i-list is *s_isize*−2 blocks long. *s_fsize* is the first block not potentially available for allocation to a file. These numbers are used by the system to check for bad block numbers; if an "impossible" block number is allocated from the free list or is freed, a diagnostic is written on the on-line console. Moreover, the free array is cleared, so as to prevent further allocation from a presumably corrupted free list.

The free list for each volume is maintained as follows. The *s_free* array contains, in *s_free*[1], . . ., *s_free*[*s_nfree*−1], up to 49 numbers of free blocks. *s_free*[0] is the block number of the head of a chain of blocks constituting the free list. The first long in each free-chain block is the number (up to 50) of free-block numbers listed in the next 50 longs of this chain member. The first of these 50 blocks is the link to the next member of the chain. To allocate a block: decrement *s_nfree*, and the new block is *s_free*[*s_nfree*]. If the new block number is 0, there are no blocks left, so give an error. If *s_nfree* became 0, read in the block named by the new block number, replace *s_nfree* by its first word, and copy the block numbers in the next 50 longs into the *s_free* array. To free a block, check if *s_nfree* is 50; if so, copy *s_nfree* and the *s_free* array into it, write it out, and set *s_nfree* to 0. In any event set *s_free*[*s_nfree*] to the freed block's number and increment *s_nfree*.

s_tfree is the total free blocks available in the file system.

s_ninode is the number of free i-numbers in the *s_inode* array. To allocate an i-node: if *s_ninode* is greater than 0, decrement it and return *s_inode*[*s_ninode*]. If it was 0, read the i-list and place the numbers of all free i-nodes (up to 100) into the *s_inode* array, then try again. To free an i-node, provided *s_ninode* is less than 100, place its number into *s_inode*[*s_ninode*] and increment *s_ninode*. If *s_ninode* is already 100, do not bother to enter the freed i-node into any table. This list of i-nodes is only to speed up the allocation process; the information as to whether the i-node is really free or not is maintained in the i-node itself.

s_tinode is the total free i-nodes available in the file system.

s_flock and *s_ilock* are flags maintained in the core copy of the file system while it is mounted and their values on disk are immaterial. The value of *s_fmod* on disk is likewise immaterial; it is used as a flag to indicate that the super-block has changed and should be copied to the disk during the next periodic update of file system information.

s_ronly is a read-only flag to indicate write-protection.

s_time is the last time the super-block of the file system was changed, and is the number of seconds that have elapsed 00:00 Jan. 1, 1970 (GMT). During a reboot, the *s_time* of the super-block for the root file system is used to set the system's idea of the time.

s_fname is the name of the file system and *s_fpack* is the name of the pack.

I-numbers begin at 1, and the storage for i-nodes begins in block 2. Also, i-nodes are 64 bytes long. I-node 1 is reserved for future use. I-node 2 is reserved for the root directory of the file system, but no other i-number has a built-in meaning. Each i-node represents one file. For the format of an i-node and its flags, see *inode*(4).

SEE ALSO

mount(2), inode(4).
fsck.s51k(1M), fsdb.s51k(1M), mkfs.s51k(1M) in the *System Administrator's Reference Manual*.

NAME

fs, inode – format of FFS file system volume

SYNOPSIS

```
#include <sys/types.h>
#include <sys/fs/ffs_fs.h>
#include <sys/inode.h>
```

DESCRIPTION

Every FFS file system storage volume (disk, nine-track tape, for instance) has a common format for certain vital information. Every such volume is divided into a certain number of blocks. The block size is a parameter of the file system. Sectors beginning at BFS_BBLOCK and continuing for BBSIZE are used to contain primary and secondary bootstrapping programs.

The actual file system begins at sector BFS_SBLOCK with the *super block* that is of size BFS_SBSIZE. The layout of the super block as defined by the include file *<sys/fs/ffs_fs.h>* is:

```
#define BFS_FS_MAGIC        0x011954
struct  fs {
        struct  fs *fs_link;            /* linked list of file systems */
        struct  fs *fs_rlink;           /*     used for incore super blocks */
        daddr_t fs_sblkno;              /* addr of super-block in filesys */
        daddr_t fs_cblkno;              /* offset of cyl-block in filesys */
        daddr_t fs_iblkno;              /* offset of inode-blocks in filesys */
        daddr_t fs_dblkno;              /* offset of first data after cg */
        long    fs_cgoffset;            /* cylinder group offset in cylinder */
        long    fs_cgmask;              /* used to calc mod fs_ntrak */
        time_t  fs_time;                /* last time written */
        long    fs_size;        /* number of blocks in fs */
        long    fs_dsize;               /* number of data blocks in fs */
        long    fs_ncg;                 /* number of cylinder groups */
        long    fs_bsize;               /* size of basic blocks in fs */
        long    fs_fsize;               /* size of frag blocks in fs */
        long    fs_frag;        /* number of frags in a block in fs */
/* these are configuration parameters */
        long    fs_minfree;             /* minimum percentage of free blocks */
        long    fs_rotdelay;            /* num of ms for optimal next block */
        long    fs_rps;                 /* disk revolutions per second */
/* these fields can be computed from the others */
        long    fs_bmask;               /* "blkoff" calc of blk offsets */
        long    fs_fmask;               /* "fragoff" calc of frag offsets */
        long    fs_bshift;              /* "lblkno" calc of logical blkno */
        long    fs_fshift;              /* "numfrags" calc number of frags */
/* these are configuration parameters */
        long    fs_maxcontig;           /* max number of contiguous blks */
        long    fs_maxbpg;              /* max number of blks per cyl group */
/* these fields can be computed from the others */
        long    fs_fragshift;           /* block to frag shift */
        long    fs_fsbtodb;             /* fsbtodb and dbtofsb shift constant */
        long    fs_sbsize;              /* actual size of super block */
        long    fs_csmask;              /* csum block offset */
        long    fs_csshift;             /* csum block number */
        long    fs_nindir;              /* value of NINDIR */
        long    fs_inopb;               /* value of INOPB*/
```

```
        long    fs_nspf;                /* value of NSPF */
        long    fs_optim;               /* optimization preference, see below */
        long    fs_sparecon[5];         /* reserved for future constants */
/* sizes determined by number of cylinder groups and their sizes */
        daddr_t fs_csaddr;              /* blk addr of cyl grp summary area */
        long    fs_cssize;              /* size of cyl grp summary area */
        long    fs_cgsize;              /* cylinder group size */
/* these fields should be derived from the hardware */
        long    fs_ntrak;               /* tracks per cylinder */
        long    fs_nsect;               /* sectors per track */
        long    fs_spc;                 /* sectors per cylinder */
/* this comes from the disk driver partitioning */
        long    fs_ncyl;                /* cylinders in file system */
/* these fields can be computed from the others */
        long    fs_cpg;                 /* cylinders per group */
        long    fs_ipg;                 /* inodes per group */
        long    fs_fpg;                 /* blocks per group * fs_frag */
/* this data must be re-computed after crashes */
        struct  csum fs_cstotal;        /* cylinder summary information */
/* these fields are cleared at mount time */
        char    fs_fmod;                /* super block modified flag */
        char    fs_clean;               /* file system is clean flag */
        char    fs_ronly;               /* mounted read-only flag */
        char    fs_flags;               /* currently unused flag */
        char    fs_fsmnt[BFS_MAXMNTLEN]; /* name mounted on */
/* these fields retain the current block allocation info */
        long    fs_cgrotor;             /* last cg searched */
        struct  csum *fs_csp[BFS_MAXCSBUFS];/* list of fs_cs info buffers */
        long    fs_cpc;                 /* cyl per cycle in postbl */
        short   fs_postbl[BFS_MAXCPG][BFS_NRPOS];/* head of blocks for each rotation */
        long    fs_magic;               /* magic number */
        u_char  fs_rotbl[1];            /* list of blocks for each rotation */
/* actually longer */
};
```

Each disk drive contains some number of file systems. A file system consists of a number of cylinder groups. Each cylinder group has inodes and data.

A file system is described by its super-block, which in turn describes the cylinder groups. The super-block is critical data and is replicated in each cylinder group to protect against catastrophic loss. This is done at file system creation time and the critical super-block data does not change, so the copies need not be referenced further unless disaster strikes.

Addresses stored in inodes are capable of addressing fragments of 'blocks'. File system blocks of at most size BFS_MAXBSIZE can be optionally broken into 2, 4, or 8 pieces, each of which is addressable; these pieces may be BFS_DEV_BSIZE, or some multiple of a BFS_DEV_BSIZE unit.

Large files consist of exclusively large data blocks. To avoid undue wasted disk space, the last data block of a small file is allocated as only as many fragments of a large block as are necessary. The file system format retains only a single pointer to such a fragment, which is a piece of a single large block that has been divided. The size of such a fragment is determinable from information in the inode, using the "blksize(fs, ip, lbn)" macro.

The file system records space availability at the fragment level; to determine block availability, aligned fragments are examined.

The root inode is the root of the file system. Inode 0 can't be used for normal purposes and historically bad blocks were linked to inode 1, thus the root inode is 2 (inode 1 is no longer used for this purpose; however, numerous dump tapes make this assumption). The *lost+found* directory is given the next available inode when it is initially created by *mkfs*.

fs_minfree gives the minimum acceptable percentage of file system blocks that may be free. If the freelist drops below this level only the super-user may continue to allocate blocks. This may be set to 0 if no reserve of free blocks is deemed necessary, however severe performance degradations will be observed if the file system is run at greater than 90% full; thus the default value of *fs_minfree* is 10%.

Empirically the best trade-off between block fragmentation and overall disk utilization at a loading of 90% comes with a fragmentation of 4, thus the default fragment size is a fourth of the block size.

fs_optim specifies whether the file system should try to minimize the time spent allocating blocks, or if it should attempt to minimize the space fragmentation on the disk. If the value of fs_minfree (see above) is less than 10%, then the file system defaults to optimizing for space to avoid running out of full sized blocks. If the value of minfree is greater than or equal to 10%, fragmentation is unlikely to be problematical, and the file system defaults to optimizing for time.

cylinder group related limits: Each cylinder keeps track of the availability of blocks at different rotational positions, so that sequential blocks can be laid out with minimum rotational latency. NRPOS is the number of rotational positions which are distinguished. With NRPOS 8 the resolution of the summary information is 2ms for a typical 3600 rpm drive.

fs_rotdelay gives the minimum number of milliseconds to initiate another disk transfer on the same cylinder. It is used in determining the rotationally optimal layout for disk blocks within a file; the default value for *fs_rotdelay* is 2ms.

Each file system has a statically allocated number of inodes. An inode is allocated for each BFS_NBPI bytes of disk space. The inode allocation strategy is extremely conservative.

BFS_MAXIPG bounds the number of inodes per cylinder group, and is needed only to keep the structure simpler by having the only a single variable size element (the free bit map).

N.B.: BFS_MAXIPG must be a multiple of BFS_INOPB(fs).

BFS_MINBSIZE is the smallest allowable block size. With a BFS_MINBSIZE of 4096 it is possible to create files of size 2^32 with only two levels of indirection. BFS_MINBSIZE must be big enough to hold a cylinder group block, thus changes to (struct cg) must keep its size within BFS_MINBSIZE. BFS_MAXCPG is limited only to dimension an array in (struct cg); it can be made larger as long as that structure's size remains within the bounds dictated by BFS_MINBSIZE. Note that super blocks are never more than size SBSIZE.

The path name on which the file system is mounted is maintained in *fs_fsmnt*. BFS_MAXMNTLEN defines the amount of space allocated in the super block for this name. The limit on the amount of summary information per file system is defined by BFS_MAXCSBUFS. It is currently parameterized for a maximum of two million cylinders.

Per cylinder group information is summarized in blocks allocated from the first cylinder group's data blocks. These blocks are read in from *fs_csaddr* (size *fs_cssize*) in addition to the super block.

N.B.: sizeof (struct csum) must be a power of two in order for the "fs_cs" macro to work.

super block for a file system: BFS_MAXBPC bounds the size of the rotational layout tables and is limited by the fact that the super block is of size BFS_SBSIZE. The size of these tables is **inversely** proportional to the block size of the file system. The size of the tables is increased when sector sizes are not powers of two, as this increases the number of cylinders included before the rotational pattern repeats (*fs_cpc*). The size of the rotational layout tables is derived from the number of bytes remaining in (struct fs).

BFS_MAXBPG bounds the number of blocks of data per cylinder group, and is limited by the fact that cylinder groups are at most one block. The size of the free block table is derived from the size of blocks and the number of remaining bytes in the cylinder group structure (struct cg).

inode: The inode is the focus of all file activity in the UNIX file system. There is a unique inode allocated for each active file, each current directory, each mounted-on file, text file, and the root. An inode is 'named' by its device/i-number pair. For further information, see the include file *<sys/inode.h>* and *<sys/fs/bfs_inode.h>*.

NAME

fspec – format specification in text files

DESCRIPTION

It is sometimes convenient to maintain text files on the UNIX system with non-standard tabs, (i.e., tabs which are not set at every eighth column). Such files must generally be converted to a standard format, frequently by replacing all tabs with the appropriate number of spaces, before they can be processed by UNIX system commands. A format specification occurring in the first line of a text file specifies how tabs are to be expanded in the remainder of the file.

A format specification consists of a sequence of parameters separated by blanks and surrounded by the brackets <: and :>. Each parameter consists of a keyletter, possibly followed immediately by a value. The following parameters are recognized:

t*tabs* — The **t** parameter specifies the tab settings for the file. The value of *tabs* must be one of the following:

1. a list of column numbers separated by commas, indicating tabs set at the specified columns;
2. a **–** followed immediately by an integer *n*, indicating tabs at intervals of *n* columns;
3. a **–** followed by the name of a "canned" tab specification.

Standard tabs are specified by **t–8**, or equivalently, **t1,9,17,25,**etc. The canned tabs which are recognized are defined by the *tabs*(1) command.

s*size* — The **s** parameter specifies a maximum line size. The value of *size* must be an integer. Size checking is performed after tabs have been expanded, but before the margin is prepended.

m*margin* — The **m** parameter specifies a number of spaces to be prepended to each line. The value of *margin* must be an integer.

d — The **d** parameter takes no value. Its presence indicates that the line containing the format specification is to be deleted from the converted file.

e — The **e** parameter takes no value. Its presence indicates that the current format is to prevail only until another format specification is encountered in the file.

Default values, which are assumed for parameters not supplied, are **t–8** and **m0**. If the **s** parameter is not specified, no size checking is performed. If the first line of a file does not contain a format specification, the above defaults are assumed for the entire file. The following is an example of a line containing a format specification:

```
* <:t5,10,15 s72:> *
```

If a format specification can be disguised as a comment, it is not necessary to code the **d** parameter.

SEE ALSO

ed(1) in the *User's Reference Manual*.

NAME

fstab – static information about filesystems

SYNOPSIS

#include <mntent.h>

DESCRIPTION

The file */etc/fstab* describes the filesystems and swapping partitions used by the local machine. The system administrator can modify it with a text editor. It is read by commands that mount, unmount, dump, restore, and check the consistency of filesystems; also by the system when providing swap space. The file consists of a number of lines of the form:

fsname dir type opts freq passno

for example:

/dev/xy0a / efs rw,noquota 1 2

The entries from this file are accessed using the routines in *getmntent*(3), which returns a structure of the following form:

```
struct mntent {
        char    *mnt_fsname;    /* filesystem name */
        char    *mnt_dir;       /* filesystem path prefix */
        char    *mnt_type;      /* efs, nfs, or ignore */
        char    *mnt_opts;      /* rw, ro, noquota, quota, hard, soft */
        int     mnt_freq;       /* dump frequency, in days */
        int     mnt_passno;     /* pass number on parallel fsck */
};
```

Fields are separated by white space; a '#' as the first non-white character indicates a comment.

The *mnt_dir* fields is the full path name of the directory to be mounted on.

The *mnt_type* field determines how the *mnt_fsname* and *mnt_opts* fields will be interpreted. Here is a list of the filesystem types currently supported, and the way each of them interprets these fields:

efs	*mnt_fsname*	must be a block special device.
nfs	*mnt_fsname*	the path on the server of the directory to be served.

If the *mnt_type* is specified as **ignore** , then the entry is ignored. This is useful to show disk partitions not currently used.

The *mnt_opts* field contains a list of comma-separated option words. Some *mnt_opts* are valid for all filesystem types, while others apply to a specific type only:

mnt_opts valid on *all* file systems (the default is **rw,suid**):

rw	read/write.
ro	read-only.
suid	set-uid execution allowed.
nosuid	set-uid execution not allowed.
suid	and **nosuid** are not supported.
raw=path	the filesystem's raw device interface pathname.
fsck	*fsck*(1M) invoked with no filesystem arguments should check this

filesystem.

nofsck — *fsck*(1M) should not check this filesystem by default.

hide — ignore this entry during a **mount −a** command to allow you to define *fstab* entries for commonly used filesystems you don't want to automatically mount.

mnt_opts specific to **nfs** (NFS) file systems (the defaults are:

fg,retry=1,timeo=7,retrans=4,port=NFS_PORT,hard

with defaults for *rsize* and *wsize* set by the kernel):

bg — if the first attempt fails, retry in the background.

fg — retry in foreground.

retry=*n* — set number of failure retries to *n*.

rsize=*n* — set read buffer size to *n* bytes.

wsize=*n* — set write buffer size to *n bytes.*

timeo=*n* — set NFS timeout to *n* tenths of a second.

retrans=*n* — set number of NFS retransmissions to *n*.

port=*n* — set server IP port number to *n*.

soft — return error if server doesn't respond.

hard — retry request until server responds.

The **bg** option causes *mount* to run in the background if the server's *mountd*(1M) does not respond. *mount* attempts each request **retry=***n* times before giving up. Once the filesystem is mounted, each **nfs** request made in the kernel waits **timeo=***n* tenths of a second for a response. If no response arrives, the time-out is multiplied by **2** and the request is retransmitted. When **retrans=***n* retransmissions have been sent with no reply a **soft** mounted filesystem returns an error on the request and a **hard** mounted filesystem retries the request. The number of bytes in a read or write request can be set with the **rsize** and **wsize** options.

mnt_freq and *mnt_passno* are not supported.

FILES

/etc/fstab

SEE ALSO

getmntent(3), fsck(1M), mount(1M).

ORIGIN

Sun Microsystems

NAME

gettydefs – speed and terminal settings used by getty

DESCRIPTION

The **/etc/gettydefs** file contains information used by *getty*(1M) to set up the speed and terminal settings for a line. It supplies information on what the *login* prompt should look like. It also supplies the speed to try next if the user indicates the current speed is not correct by typing a <*break*> character.

Each entry in **/etc/gettydefs** has the following format:

label# initial-flags # final-flags # login-prompt #next-label

Each entry is followed by a blank line. The various fields can contain quoted characters of the form **\b**, **\n**, **\c**, etc., as well as *\nnn*, where *nnn* is the octal value of the desired character. The various fields are:

label — This is the string against which *getty* tries to match its second argument. It is often the speed, such as **1200**, at which the terminal is supposed to run, but it need not be (see below).

initial-flags — These flags are the initial *ioctl*(2) settings to which the terminal is to be set if a terminal type is not specified to *getty*. The flags that *getty* understands are the same as the ones listed in **/usr/include/sys/termio.h** [see *termio*(7)]. Normally only the speed flag is required in the *initial-flags*. *getty* automatically sets the terminal to raw input mode and takes care of most of the other flags. The *initial-flag* settings remain in effect until *getty* executes *login*(1).

final-flags — These flags take the same values as the *initial-flags* and are set just prior to *getty* executes *login*. The speed flag is again required. The composite flag **SANE** takes care of most of the other flags that need to be set so that the processor and terminal are communicating in a rational fashion. The other two commonly specified *final-flags* are **TAB3**, so that tabs are sent to the terminal as spaces, and **HUPCL**, so that the line is hung up on the final close.

login-prompt — This entire field is printed as the *login-prompt*. Unlike the above fields where white space is ignored (a space, tab or new-line), they are included in the *login-prompt* field.

next-label — If this entry does not specify the desired speed, indicated by the user typing a <*break*> character, then *getty* will search for the entry with *next-label* as its *label* field and set up the terminal for those settings. Usually, a series of speeds are linked together in this fashion, into a closed set; For instance, **2400** linked to **1200**, which in turn is linked to **300**, which finally is linked to **2400**.

If *getty* is called without a second argument, then the first entry of **/etc/gettydefs** is used, thus making the first entry of **/etc/gettydefs** the default entry. It is also used if *getty* can not find the specified *label*. If **/etc/gettydefs** itself is missing, there is one entry built into the command which will bring up a terminal at **300** baud.

It is strongly recommended that after making or modifying **/etc/gettydefs**, it be run through *getty* with the check option to be sure there are no errors.

FILES

/etc/gettydefs

SEE ALSO

ioctl(2).
getty(1M), termio(7) in the *System Administrator's Reference Manual*.
login(1) in the *User's Reference Manual*.

NAME

group – group file

SYNOPSIS

/etc/group

DESCRIPTION

group file group contains for each group the following information:

- group name
- encrypted password
- numerical group ID
- a comma separated list of all users allowed in the group

This is an ASCII file. The fields are separated by colons; each group is separated from the next by a new-line. If the password field is null, no password is demanded.

This file resides in the */etc* directory. Because of the encrypted passwords, it can and does have general read permission and can be used, for example, to map numerical group ID's to names.

A group file can have a line beginning with a plus (+), which means to incorporate entries from the Yellow Pages. There are two styles of + entries: All by itself, + means to insert the entire contents of the Yellow Pages group file at that point; *+name* means to insert the entry (if any) for *name* from the Yellow Pages at that point. If a + entry has a non-null password or group member field, the contents of that field will overide what is contained in the Yellow Pages. The numerical group ID field cannot be overridden.

EXAMPLE

```
+myproject:::bill, steve
+:
```

If these entries appear at the end of a group file, then the group *myproject* will have members *bill* and *steve*, and the password and group ID of the Yellow Pages entry for the group *myproject*., All the groups listed in the Yellow Pages will be pulled in and placed after the entry for *myproject*.

FILES

/etc/group

SEE ALSO

crypt(3), passwd(1), passwd(4)

ERRORS

The *passwd*(1) command won't change group passwords.

ORIGIN

Sun Microsystems

NAME

hosts – host name data base

DESCRIPTION

The *hosts* file contains information regarding the known hosts on the network. For each host a single line should be present with the following information:

official host name
Internet address
aliases

Items are separated by any number of blanks and/or tab characters. A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file.

When using the name server, this file provides a backup when the name server is not running. For the name server, it is suggested that only a few addresses be included in this file. These include address for the local interfaces that *ifconfig*(1M) needs at boot time and a few machines on the local network.

This file may be created from the official host data base maintained at the Network Information Control Center (NIC), though local changes may be required to bring it up to date regarding unofficial aliases and/or unknown hosts. As the data base maintained at NIC is incomplete, use of the name server is recommend for sites on the DARPA Internet.

Network addresses are specified in the conventional "." notation using the *inet_addr*() routine from the Internet address manipulation library, *inet*(3N). Host names may contain any printable character other than a field delimiter, newline, or comment character.

FILES

/etc/hosts

SEE ALSO

gethostbyname(3N), ifconfig(1M)
Name Server Operations Guide for BIND

NAME

hosts.equiv – list of trusted hosts

DESCRIPTION

Hosts.equiv resides in directory */etc* and contains a list of trusted hosts. When an *rlogin(1C)* or *rsh(1C)* request from such a host is made, and the initiator of the request is in */etc/passwd*, then, no further validity checking is done. That is, *rlogin* does not prompt for a password, and *rsh* completes successfully. So a remote user is "equivalenced" to a local user with the same user name when the remote user is in **hosts.equiv.**

The format of **hosts.equiv** is a list of names, as in this example:

```
host1
host2
+@group1
-@group2
```

A line consisting of a simple host name means that anyone logging in from that host is trusted. A line consisting of +*@group* means that all hosts in that network group are trusted. A line consisting of −*@group* means that hosts in that group are not trusted. Programs scan **hosts.equiv** linearly, and stop at the first hit (either positive for hostname and +@ entries, or negative for −@ entries). A line consisting of a single + means that everyone is trusted.

The *.rhosts* file has the same format as **hosts.equiv.** When user *XXX* executes *rlogin* or *rsh*, the *.rhosts* file from *XXX*'s home directory is conceptually concatenated onto the end of **hosts.equiv** for permission checking. However, −@ entries are not sticky. If a user is excluded by a minus entry from **hosts.equiv** but included in *.rhosts,* then that user is considered trusted. In the special case when the user is root, then only the */.rhosts* file is checked.

It is also possible to have two entries (separated by a single space) on a line of these files. In this case, if the remote host is equivalenced by the first entry, then the user named by the second entry is allowed to log in as anyone, that is, specify any name to the **−l** flag (provided that name is in the */etc/passwd* file, of course). Thus the entry

```
sundown john
```

in */etc/hosts.equiv* allows *john* to log in from sundown as anyone. The usual usage would be to put this entry in the *.rhosts* file in the home directory for *bill.* Then *john* may log in as *bill* when coming from sundown. The second entry may be a netgroup, thus

```
+@group1 +@group2
```

allows any user in *group2* coming from a host in *group1* to log in as anyone.

FILES

/etc/hosts.equiv
~/.rhosts

WARNING

The references to network groups (+@ and −@ entries) in **hosts.equiv** and *.rhosts* are only supported when the *netgroup* file is supplied by the Yellow Pages.

SEE ALSO

netgroup(4), rhosts(4)

ORIGIN

4.3 BSD

NAME

inittab – script for the init process

DESCRIPTION

The *inittab* file supplies the script to *init*'s role as a general process dispatcher. The process that constitutes the majority of *init*'s process dispatching activities is the line process **/etc/getty** that initiates individual terminal lines. Other processes typically dispatched by *init* are daemons and the shell.

The *inittab* file is composed of entries that are position dependent and have the following format:

id:rstate:action:process

Each entry is delimited by a newline, however, a backslash (\) preceding a newline indicates a continuation of the entry. Up to 512 characters per entry are permitted. Comments may be inserted in the *process* field using the *sh*(1) convention for comments. Comments for lines that spawn *getty*s are displayed by the *who*(1) command. It is expected that they will contain some information about the line such as the location. There are no limits (other than maximum entry size) imposed on the number of entries within the *inittab* file. The entry fields are:

id This is one or two characters used to uniquely identify an entry.

rstate This defines the *run-level* in which this entry is to be processed. *run-levels* effectively correspond to a configuration of processes in the system. That is, each process spawned by *init* is assigned a *run-level* or *run-levels* in which it is allowed to exist. The *run-levels* are represented by a number ranging from **0** through **6**. As an example, if the system is in *run-level* **1**, only those entries having a **1** in the *rstate* field will be processed. When *init* is requested to change *run-levels,* all processes which do not have an entry in the *rstate* field for the target *run-level* will be sent the warning signal (**SIGTERM**) and allowed a 20-second grace period before being forcibly terminated by a kill signal (**SIGKILL**). The *rstate* field can define multiple *run-levels* for a process by selecting more than one *run-level* in any combination from **0–6**. If no *run-level* is specified, then the process is assumed to be valid at all *run-levels* **0–6**. There are three other values, **a**, **b** and **c**, which can appear in the *rstate* field, even though they are not true *run-levels*. Entries which have these characters in the *rstate* field are processed only when the *telinit* [see *init*(1M)] process requests them to be run (regardless of the current *run-level* of the system). They differ from *run-levels* in that *init* can never enter *run-level* **a**, **b** or **c**. Also, a request for the execution of any of these processes does not change the current *run-level*. Furthermore, a process started by an **a**, **b** or **c** command is not killed when *init* changes levels. They are only killed if their line in **/etc/inittab** is marked **off** in the *action* field, their line is deleted entirely from **/etc/inittab**, or *init* goes into the *SINGLE USER* state.

action Key words in this field tell *init* how to treat the process specified in the *process* field. The actions recognized by *init* are as follows:

respawn
If the process does not exist then start the process, do not wait for its termination (continue scanning the *inittab* file), and when it dies restart the process. If the process currently exists then do nothing and continue scanning the *inittab* file.

wait Upon *init*'s entering the *run-level* that matches the entry's *rstate*, start the

process and wait for its termination. All subsequent reads of the *inittab* file while *init* is in the same *run-level* will cause *init* to ignore this entry.

once Upon *init*'s entering a *run-level* that matches the entry's *rstate*, start the process, do not wait for its termination. When it dies, do not restart the process. If upon entering a new *run-level*, where the process is still running from a previous *run-level* change, the program will not be restarted.

boot The entry is to be processed only at *init*'s boot-time read of the *inittab* file. *init* is to start the process, not wait for its termination; and when it dies, not restart the process. In order for this instruction to be meaningful, the *rstate* should be the default or it must match *init*'s *run-level* at boot time. This action is useful for an initialization function following a hardware reboot of the system.

bootwait
The entry is to be processed the first time *init* goes from single-user to multi-user state after the system is booted. (If **initdefault** is set to **2**, the process will run right after the boot.) *init* starts the process, waits for its termination and, when it dies, does not restart the process.

powerfail
Execute the process associated with this entry only when *init* receives a power fail signal [**SIGPWR** see *signal*(2)].

powerwait
Execute the process associated with this entry only when *init* receives a power fail signal (**SIGPWR**) and wait until it terminates before continuing any processing of *inittab*.

off If the process associated with this entry is currently running, send the warning signal (**SIGTERM**) and wait 20 seconds before forcibly terminating the process via the kill signal (**SIGKILL**). If the process is nonexistent, ignore the entry.

ondemand
This instruction is really a synonym for the **respawn** action. It is functionally identical to **respawn** but is given a different keyword in order to divorce its association with *run-levels*. This is used only with the **a**, **b** or **c** values described in the *rstate* field.

initdefault
An entry with this *action* is only scanned when *init* initially invoked. *init* uses this entry, if it exists, to determine which *run-level* to enter initially. It does this by taking the highest *run-level* specified in the **rstate** field and using that as its initial state. If the *rstate* field is empty, this is interpreted as **0123456** and so *init* will enter *run-level* **6**. Additionally, if *init* does not find an **initdefault** entry in **/etc/inittab**, then it will request an initial *run-level* from the user at reboot time.

sysinit Entries of this type are executed before *init* tries to access the console (i.e., before the **Console Login:** prompt). It is expected that this entry will be only used to initialize devices on which *init* might try to ask the *run-level* question. These entries are executed and waited for before continuing.

process This is a *sh* command to be executed. The entire **process** field is prefixed with *exec* and passed to a forked *sh* as **sh -c 'exec** *command'*. For this reason, any legal *sh* syntax can appear in the *process* field. Comments can be inserted with the ; *#comment* syntax.

FILES

/etc/inittab

SEE ALSO

exec(2), open(2), signal(2).
getty(1M), init(1M) in the *System Administrator's Reference Manual*.
sh(1), who(1) in the *User's Reference Manual*.

NAME

inode – format of a s51k i-node

SYNOPSIS

#include <sys/types.h>
#include <sys/ino.h>

DESCRIPTION

Note: *The obsolete s51k file system has been kept for backward compatibility. The Fast File System (FFS) is preferred. See fs(4FFS).*

An i-node for a plain file or directory in an s51k file system has the following structure defined by **<sys/ino.h>**.

```
/* Inode structure as it appears on a disk block. */
struct dinode
{
        ushort  di_mode;    /* mode and type of file */
        short   di_nlink;   /* number of links to file */
        ushort  di_uid;     /* owner's user id */
        ushort  di_gid;     /* owner's group id */
        off_t   di_size;    /* number of bytes in file */
        char    di_addr[40]; /* disk block addresses */
        time_t  di_atime;   /* time last accessed */
        time_t  di_mtime;   /* time last modified */
        time_t  di_ctime;   /* time of last file status change */
};
/*
 * the 40 address bytes:
 *     39 used; 13 addresses
 *     of 3 bytes each.
 */
```

For the meaning of the defined types *off_t* and *time_t* see *types*(5).

SEE ALSO

stat(2), fs (4FFS), types(5).

NAME

intro – introduction to special files and hardware support

DESCRIPTION

This section describes the special files, related driver functions, and networking support available in the system. In this part of the manual, the SYNOPSIS section of each configurable device gives a sample specification for use in constructing a system description for the *config*(8) program. The DIAGNOSTICS section lists messages that might appear on the console and/or in the system error log */usr/adm/messages* from errors in device operation; see *syslogd*(8) for more information.

This section contains both devices that might be configured into the system, "4" entries, and network related information, "4N", "4P", and "4F" entries; The networking support is introduced in *intro*(4N).

MIPS DEVICE SUPPORT

This section describes the hardware supported on the MIPS systems. Software support for these devices comes in two forms. A hardware device may be supported with a character or block *device driver*, or it may be used within the networking subsystem and have a *network interface driver*. Block and character devices are accessed through files in the file system of a special type; c.f. *mknod*(8). Network interfaces are indirectly accessed through the interprocess communication facilities provided by the system; see *socket*(2).

A hardware device is identified to the system at configuration time and the appropriate device or network interface driver is then compiled into the system. When the resultant system is booted, the autoconfiguration facilities in the system probe for the device on the VMEbus and, if found, enable the software support for it. If a VMEbus device does not respond at autoconfiguration time it is not accessible at any time afterwards. To enable a VMEbus device that did not autoconfigure, the system will have to be rebooted.

The autoconfiguration system is described in *autoconf*(4). A list of the supported devices is given below.

SEE ALSO

intro(4).

Building 4.3BSD UNIX Systems with *Config*

LIST OF DEVICES

The devices listed below are supported in this incarnation of the system. Pseudo-devices are not listed. Devices are indicated by their functional interface. If second vendor products provide functionally identical interfaces they should be usable with the supplied software. **(Beware, however, that we promise the software works ONLY with the hardware indicated on the appropriate manual page.)** Occasionally, new devices of a similar type may be added simply by creating appropriate table entries in the driver.

cp	ISI Communications Processor
dkip	Interphase V-SMD 3200 Disk Controller
enp	CMC 10Mb/s Ethernet Controller
mt	Tape Drive Interface
st	ISI QIC-2 1/4" Streaming Tape Drive Interface

NAME

issue – issue identification file

DESCRIPTION

The file **/etc/issue** contains the *issue* or project identification to be printed as a login prompt. This is an ASCII file which is read by program *getty* and then written to any terminal spawned or respawned from the *lines* file.

FILES

/etc/issue

SEE ALSO

login(1) in the *User's Reference Manual.*

NAME

ldfcn – common object file access routines

SYNOPSIS

#include <stdio.h>
#include <filehdr.h>
#include <syms.h>
#include <ldfcn.h>

DESCRIPTION

The common object file access routines are a collection of functions for reading an object file that is in common object file form. Although the calling program must know the detailed structure of the parts of the object file that it processes, the routines effectively insulate the calling program from knowledge of the overall structure of the object file.

The interface between the calling program and the object file access routines is based on the defined type **LDFILE** (defined as **struct ldfile**), which is declared in the header file **<ldfcn.h>**. Primarily, this structure provides uniform access to simple object files and object files that are members of an archive file.

The function *ldopen*(3X) allocates and initializes the **LDFILE** structure, reads in the symbol table header, if present, and returns a pointer to the structure to the calling program. The fields of the **LDFILE** structure can be accessed individually through macros defined in **<ldfcn.h>**. The fields contain the following information:

LDFILE *ldptr;

TYPE(ldptr) The file magic number, used to distinguish between archive members and simple object files.

IOPTR(ldptr) The file pointer returned by *fopen*(3S) and used by the standard input/output functions.

OFFSET(ldptr) The file address of the beginning of the object file; if the object file is a member of an archive file, the offset is non-zero.

HEADER(ldptr)
The file header structure of the object file.

SYMHEADER(ldptr)
The symbolic header structure for the symbol table associated with the object file.

PFD(ldptr) The file table associated with the symbol table.

SYMTAB(ldptr) A pointer to a copy of the symbol table in memory. It's accessed through the pCHDR structure (see cmplrs/stsupport.h). If no symbol table is present, this field is NULL. *NOTE:* This macro causes the whole symbol table to be read.

LDSWAP(ldptr) If the header and symbol table structures are swapped within the object file and all access requires using libsex, this field is set to true. **NOTE:** If you use *libmld* routines, all structures, except the optional header and auxiliaries, are swapped.

The object file access functions can be divided into five categories:

(1) functions that open or close an object file

ldopen(3X) and *ldaopen*
open a common object file

ldclose(3X) and *ldaclose*
close a common object file

(2) functions that return header or symbol table information

ldahread(3X)
read the archive header of a member of an archive file
ldfhread(3X)
read the file header of a common object file
ldshread(3X) and *ldnshread*
read a section header of a common object file
ldtbread(3X)
read a symbol table entry of a common object file
ldgetname(3X)
retrieve a symbol name from a symbol table entry or from the string table
ldgetaux(3X)
retrieve a pointer into the aux table for the specified ldptr
ldgetsymstr(3X)
create a type string (for example, C declarations) for the specified symbol
ldgetpd(3X)
retrieve a procedure descriptor
ldgetrfd(3X)
retrieve a relative file table entry

(3) functions that position an object file at (seek to) the start of the section, relocation, or line number information for a particular section

ldohseek(3X)
seek to the optional file header of a common object file
ldsseek(3X) and *ldnsseek*
seek to a section of a common object file
ldrseek(3X) and *ldnrseek*
seek to the relocation information for a section of a common object file
ldlseek(3X) and *ldnlseek*
seek to the line number information for a section of a common object file
ldtbseek(3X)
seek to the symbol table of a common object file

(4) miscellaneous functions

ldtbindex(3X)
return the index of a particular common object file symbol table entry
ranhashinit(3X)
initialize the tables and constants so that the archive hash and lookup routines can work
ranhash(3X)
give a string return the hash index for it
ranlookup(3X)
return an archive hash bucket that is empty or matches the string argument
disassembler(3X)
print MIPS assembly instructions
ldreadst(3X)

cause section of the the symbol table to be read

These functions are described in detail in the manual pages identified for each function.

Ldopen and *ldaopen* both return pointers to a **LDFILE** structure.

MACROS

Additional access to an object file is provided through a set of macros defined in **<ldfcn.h>**. These macros parallel the standard input/output file reading and manipulating functions. They translate a reference of the **LDFILE** structure into a reference to its file descriptor field.

The following macros are provided:

GETC(ldptr)
FGETC(ldptr)
GETW(ldptr)
UNGETC(c, ldptr)
FGETS(s, n, ldptr)
FREAD((char *) ptr, sizeof (*ptr), nitems, ldptr)
FSEEK(ldptr, offset, ptrname)
FTELL(ldptr)
REWIND(ldptr)
FEOF(ldptr)
FERROR(ldptr)
FILENO(ldptr)
SETBUF(ldptr, buf)
STROFFSET(ldptr)

The STROFFSET macro calculates the address of the local symbol's string table in an object file. See the manual entries for the corresponding standard input/output library functions for details on the use of these macros. (The functions are identified as 3S in Section 3 of this manual.)

The program must be loaded with the object file access routine library **libmld.a**.

WARNINGS

The macro **FSEEK** defined in the header file **<ldfcn.h>** translates into a call to the standard input/output function *fseek*(3S). **FSEEK** should not be used to seek from the end of an archive file since the end of an archive file cannot be the same as the end of one of its object file members.

SEE ALSO

ar(1), fopen(3S), fseek(3S), ldahread(3X), ldclose(3X), ldfhread(3X), ldgetname(3X), ldlread(3X), ldlseek(3X), ldohseek(3X), ldopen(3X), ldrseek(3X), ldlseek(3X), ldshread(3X), ldtbindex(3X), ldtbread(3X), ldtbseek(3X). **COFF** in the *MIPS Languages Programmer Guide*.

NAME

limits – header files for implementation-specific constants

SYNOPSIS

#include < limits.h>
#include < float.h>
#include < sys/limits.h>

DESCRIPTION

The header file < *limits.h* > specifies the sizes of intergral types as required by the proposed ANSI C standard. The header file < *float.h* > specifies the characteristics of floating types as required by the proposed ANSI C standard. The constants that refer to long doubles that should appear in < *float.h* > are not specified because MIPS does not implement long doubles. The header file < *sys/limits.h* > is a list of magnitude limitations imposed by a specific implementation of the operating system. All values in the files are specified in decimal. The file < *limits.h* > contains:

```
#define CHAR_BIT    8           /* # of bits in a "char" */
#define SCHAR_MIN   (-128)      /* min integer value of a "signed char" */
#define SCHAR_MAX   (+127)      /* max integer value of a "signed char" */
#define UCHAR_MAX   255         /* max integer value of an "unsigned char" */
#define CHAR_MIN    0           /* min integer value of a "char" */
#define CHAR_MAX    255         /* max integer value of a "char" */
#define SHRT_MIN    (-32768)    /* min decimal value of a "short" */
#define SHRT_MAX    (+32767)    /* max decimal value of a "short" */
#define USHRT_MAX   65535       /* max decimal value of an "unsigned short" */
#define INT_MIN     (-2147483648)   /* min decimal value of an "int" */
#define INT_MAX     (+2147483647)   /* max decimal value of a "int" */
#define UINT_MAX    4294967295  /* max decimal value of an "unsigned int" */
#define LONG_MIN    (-2147483648)   /* min decimal value of a "long" */
#define LONG_MAX    (+2147483647)   /* max decimal value of a "long" */
#define ULONG_MAX   4294967295  /* max decimal value of an "unsigned long" */

#define USI_MAX     4294967295  /* max decimal value of an "unsigned" */
#define WORD_BIT    32          /* # of bits in a "word" or "int" */
```

The file < *float.h* > contains:

```
#define FLT_RADIX       2       /* radix of exponent representation */
#define FLT_ROUNDS      1       /* addition rounds (>0 implemention-defined) */
/* number of base-FLT_RADIX digits in the floating point mantissa */
#define FLT_MANT_DIG    24
#define DBL_MANT_DIG    53
/* minimum positive floating-point number x such that 1.0 + x ≠ 1.0 */
#define FLT_EPSILON     1.19209290e-07
#define DBL_EPSILON     2.2204460492503131e-16
/* number of decimal digits of precision */
#define FLT_DIG         6
#define DBL_DIG         15
/* minimum negitive integer such that FLT_RADIX raised to that power minus 1
is a normalized floating point number */
#define FLT_MIN_EXP     -125
#define DBL_MIN_EXP     -1021
/* minimum normalized positive floating-point number */
#define FLT_MIN         1.17549435e-38
#define DBL_MIN         2.225073858507201e-308
```

```
/* minimum negative integer such that 10 raised to that power is in the range of
normalized floating-point numbers */
#define FLT_MIN_10_EXP   -37
#define DBL_MIN_10_EXP   -307
/* maximum integer such that FLT_RADIX raised to that power minus 1 is a
representable finite floating-point number */
#define FLT_MAX_EXP      +128
#define DBL_MAX_EXP      +1024
/* maximum representable finite floating-point number */
#define FLT_MAX          3.40282347e+38
#define DBL_MAX          1.797693134862316e+308
/* maximum integer such that 10 raised to that power is in the range of representable
finite floating-point numbers */
#define FLT_MAX_10_EXP   +38
#define DBL_MAX_10_EXP   +308
```

The file < *sys/limits.h* > contains:

```
#define ARG_MAX     5120      /* max length of arguments to exec */
#define CHILD_MAX   25        /* max # of processes per user id */
#define CLK_TCK     100       /* # of clock ticks per second */
#define FCHR_MAX    1048576   /* max size of a file in bytes */
#define LINK_MAX    32767     /* max # of links to a single file */
#define NAME_MAX    14        /* max # of characters in a file name */
#define OPEN_MAX    20        /* max # of files a process can have open */
#define PASS_MAX    8         /* max # of characters in a password */
#define PATH_MAX    256       /* max # of characters in a path name */
#define PID_MAX     30000     /* max value for a process ID */
#define PIPE_BUF    5120      /* max # bytes atomic in write to a pipe */
#define PIPE_MAX    5120      /* max # bytes written to a pipe in a write */
#define SHRT_MAX    32767     /* max decimal value of a "short" */
#define SHRT_MIN    -32767    /* min decimal value of a "short" */
#define STD_BLK     1024      /* # bytes in a physical I/O block */
#define SYS_NMLN    9         /* # of chars in uname-returned strings */
#define UID_MAX     30000     /* max value for a user or group ID */
```

NAME

linenum – line number entries in a MIPS object file

DESCRIPTION

The *cc* command generates an entry in the object file for each C source line on which a breakpoint is possible [when invoked with the **–g** option; see *cc*(1)]. Users can then reference line numbers when using the appropriate software test system [see *dbx*(1)]. The structure of these line number entries is described in the *MIPS Assembly Language Programmer's Guide* chapter 11 in the section entitled "Format of Symbol Table Entries" in that section's section on "Line Numbers".

SEE ALSO

cc(1), dbx(1), a.out(4).

NAME

magic – configuration for file command

SYNOPSIS

/etc/magic

DESCRIPTION

When *file(1)* is executed, it reads the file /etc/magic (or an alternate file if requested). This file, also called the "magic number file", contains information to help *file* decide what type of file it is looking at.

The name "magic" comes from the term "magic number", which refers to a (usually) unique combination of bytes that is used by either the operating system or system programs to recognize the file.

For example, the octal representation of the first two bytes of an old style archive file is 0177545. Thus, any program that needed to work with these files (such as a compiler, linker, or archiver) would check to make sure that this data was present. If it isn't. then the file isn't an old style archive.

The magic number file contains four types of lines: comments, specifications, and continuations. Blank lines are also allowed, and are ignored.

A comment is any line that has a '#' in the first column. All comment lines are ignored.

A specification line is used to describe a magic number. It consists of four fields, separated by tabs:

offset — This is the byte offset in the file where the data to be looked at is found. The number may be in decimal, octal (begins with a 0), or hexadecimal (begins with 0x).

type — This is the type of the data to be looked at. The type can be **byte** (single byte of data), **short** (short integer, usually 2 bytes, of data), **long** (long integer, usually 4 bytes, of data), or **string** (null-terminated string of bytes).

match — This field contains the value to be matched against the value in the file. If the *type* field is **string**, that value is compared literally. Otherwise, the value consists of an optional relational operator (**!** or **^** for not equal, **<** for less than, **>** for greater than, or **=** for equal, which is the default) and a numeric value (in decimal, octal, or hexadecimal, as with the *offset* field). In addition, if the field is a single 'x', any value is allowed (useful for printing version numbers or strings).

output — This field, which consists of the rest of the line, is the string to be printed if the value in the file matches the *match* field value. This may contain a *printf(3s)*-style '%' specifier to print the value. This should be a string or integer specifier (depending on the *type* field).

Normally, the first field printed for a file is preceded by a tab, and all subsequent fields are preceded by a space. If the first character of the field is a backspace or the characters **\b**, leading spaces are supressed. This is useful for printing data in which the value is split across fields, such as multi-word version numbers.

A specification line is used by *file* as meaning "read the required number of bytes from the file, and if the value matches the required value, print the specified output".

Continuation lines are used for printing other information about a file of a certain type. A continuation line has the same format as a specification line, except that the offset is preceded by the character '>'. This type of line is used just like a specification line, but only if the specification preceding it matches. The output is printed preceded by a space (to separate it from previously printed output). Multiple continuation lines are allowed for a specification line, in which case all continuation lines are checked, in the order they appear in the magic file.

Once a matching specification is found and processed (including checking continuations), no other searches are made for that file.

The following magic file lines show how a specification and related continuations might work:

```
0      short   0173737    Joe's file type
>8     long    >0          - version %d
>8     long    0          - prerelease
>12    long    >0          (checksum 0%lo)
```

If a file begins with a short integer whose octal value is 0173737, *file* will print the text "Joe's file type". Then, the long integer found at location 8 in the file is checked to see if it is a positive integer, in which case the text "- version" followed by the number found is printed. Next, the long integer found at location 8 in the file is checked to see if it is a 0, in which case the text "-prerelease" is printed. Finally, the long integer found at location 12 in the file is checked to see if it is a positive integer, in which case the text "(checksum" is printed followed by the number found, which is printed in octal, followed by ")".

So, a file named *joefile* with a short 0173737 at location 0, a long 7 at location 8, and a long 04088 at location 12 would cause *file* to print the text:

```
joefile:  Joe's file type - version 7 (checksum 04088)
```

SEE ALSO

file(1).

NAME

master – master configuration database

DESCRIPTION

The *master* configuration database is a collection of files. Each file contains configuration information for a device or module that may be included in the system. A file is named with the module name to which it applies. This collection of files is maintained in a directory called **$ROOT3/usr/src/uts/mips/master.d** . Each individual file has an identical format. For convenience, this collection of files will be referred to as the *master* file, as though it was a single file. This will allow a reference to the *master* file to be understood to mean the *individual file* in the **master.d** directory that corresponds to the name of a device or module. The file is used by the *lboot*(1M) program to obtain device information to generate the device driver and configurable module files. *master* consists of two parts; they are separated by a line with a dollar sign ($) in column 1. Part 1 contains device information for both hardware and software devices, and loadable modules. Part 2 contains parameter declarations. Any line with an asterisk (*) in column 1 is treated as a comment.

Part 1, Description

Hardware devices, software drivers and loadable modules are defined with a line containing the following information. Field 1 must begin in the left most position on the line. Fields are separated by white space (tab or blank).

Field 1:	element characteristics:
o	specify only once
r	required device
b	block device
c	character device
t	initialize cdevsw[].d_ttys
j	file system
s	software driver
f	STREAMS driver
m	STREAMS module
x	not a driver; a loadable module
k	kernel module
Field 2:	handler prefix (14 chars. maximum)
Field 3:	software device, external major number list. If multiple major numbers are listed, separate them with ,. Be sure not to use spaces in the list. The reason you may need a list of major numbers is if you have a device that can suport multiple major numbers. An example would be the Interphase 4210 SCSI board. For RISC/os, each additional card placed in a system will have a different major number. This field can be a - if this is not a software driver or if you wish to have the major number assigned by *lboot*. *lboot* will only assign one major number, hence you can not use *lboot* to assign multiple major numbers for one driver.
Field 4:	number of sub-devices per device; - if none. This is an optional-second element that can be after the nubmer of sub-devices, and are separated by a ,. The element is the number of controllers per major number. Using a - for this field means that only one major

number will be used no matter how many controllers exist. Therefore, **3,4** would mean 3 sub-devices per controller, and 4 controllers per each major number. **-,4** would mean no sub-devices, and 4 controllers per major number. If the second elemenet denoting the number of controllers per major number is left off, it is the same as if you denote - for it.

Field 5: dependency list (optional); this is a comma separated list of other drivers or modules that must be present in the configuration if this module is to be included

For each module, two classes of information are required by *lboot*(1M): external routine references and variable definitions. Routine lines begin with white space white space and immediately follow the initial module specification line. These lines are free form, thus they may be continued arbitrarily between non-blank tokens as long as the first character of a line is white space. Variable definition lines begin after a line that contains a '$' in column one. Variable definitions follow C language conventions, with slight modifications.

Part 1, Routine Reference Lines

If the UNIX system kernel or other dependent module contains external references to a module, but the module is not configured, then these external references would be undefined. Therefore, the *routine reference* lines are used to provide the information necessary to generate appropriate dummy functions at boot time when the driver is not loaded.

Routine references are defined as follows:

Field 1: routine name ()

Field 2: the routine type: one of

{}	routine_name(){}
{nulldev}	routine_name(){nulldev();}
{nosys}	routine_name(){return nosys();}
{nodev}	routine_name(){return nodev();}
{false}	routine_name(){return 0;}
{true}	routine_name(){return 1;}
{fsnull}	routine_name(){return fsnull();}
{fsstray}	routine_name(){return fsstray();}
{nopkg}	routine_name(){nopkg();}
{noreach}	routine_name(){noreach();}

Part 2, Variables

Variables may be declared and (optionally) statically initialized on lines after a line whose first character is a dollar sign ('$'). Variable definitions follow standard C syntax for global declarations, with the following in-line substitutions:

##M	the internal major number assigned to the current module if it is a device driver; zero of this module is not a device driver
##E	the external major number assigned to the current module; either explicitly defined by the current master file entry, or assigned by *lboot*(1M)
##C	number of controllers present; this number is determined dynamically by *lboot*(1M) for hardware devices, or by the number provided in the system file for non-hardware drivers or modules
##D	number of devices per controller taken directly from the current master file entry
##P	number of controllers per major number. From the master-file.

##N
number of major numbers neede to support this device.

##I, ##X
these will give you the list of the internal major number and external major numbers respectively. The intended use is as follows. This example might appear in the declaration section of a master file:

```
int internal [##N] = ##I;
int external [##N] = ##X;
```

Since ##N is the numbers of major used, that is what you can use to declare the size of the arrays. For a 4 major number case, these lines might be translated in the master<suffix>.c file as

```
int internal [4] = {4,5,9,11};
int external [4] = {22,27,31,35};
```

Note that when multiple major numbers are used, ##E is analogous to the first external major-number in the array, but is not recommended as the way to obtain multi-major information. Similarly with ##M.

EXAMPLES

A sample *master* file for a shared memory module would be named "**shm**". The module is an optional loadable software module that can only be specified once. The module prefix is **shm**, and it has no major number associated with it. In addition, another module named "*ipc*" is necessary for the correct operation of this module.

```
*FLAG PREFIX SOFT #DEV DEPENDENCIES
ox   shm   -   -   ipc
                        shmsys(){nosys}
                        shmexec(){}
                        shmexit(){}
                        shmfork(){}
                        shmslp(){true}
                        shmtext(){}
$
#define SHMMAX 131072
#define SHMMIN 1
#define SHMMNI 100
#define SHMSEG 6
#define SHMALL 512

struct shmid_ds shmem[SHMMNI];
struct shminfo shminfo = {
    SHMMAX,
    SHMMIN,
    SHMMNI,
    SHMSEG,
    SHMALL,
};
```

This *master* file will cause routines named *shmsys shmexec etc., to be generated by the boot program if the* **shm** *driver is not loaded, and there is a reference to this routine from any other module loaded. When the driver is loaded, the structure array shmem* will be allocated, and the structure *shminfo* allocated and initialized. will be allocated and initialized as specified.

A sample *master* file for a VME disk driver would be named "**dkip The driver is a block and a character device, the driver prefix is dkip**, and the external major number is 4. The VME interrupt priority level and vector numbers are declared in the system file (see lboot(1M)).

```
*FLAG PREFIX SOFT #DEV DEPENDENCIES
bc   dkip 4  -   -   io

$$$
/* disk driver variable tables */
#include "sys/dvh.h"
#include "sys/dkipreg.h"
#include "sys/elog.h"

struct iotime dkipiotime[##C][DKIPUPC]; /* io statistics */
struct iobuf dkipctab[##C];                    /* controller queues */
struct iobuf dkiputab[##C][DKIPUPC];           /* drive queues */
int dkipmajor = ##E;                    /* external major # */
```

This *master* file will cause entries in the block and character device switch tables to be generated, if this module is loaded. Since this is a hardware device (implied by the block and character flags), VME interrupt structures will be generated, also, by the boot program. The declared arrays will all be sized to the number of controllers present, which is determined by the boot program, based on information in the system file *$ROOT/usr/src/uts/mips/master.d/system[.suffix].*

FILES

$ROOT/usr/src/uts/mips/master.d/*
$ROOT/usr/src/uts/mips/master.d/system[.suffix]

SEE ALSO

system(4), lboot(1M)

NAME

mntent – static information about filesystems

SYNOPSIS

#include <mntent.h>

DESCRIPTION

The file */etc/fstab* describes the file systems and swapping partitions used by the local machine. It is created by the system administrator using a text editor and processed by commands which mount, unmount, check consistency of, dump and restore file systems, and by the system in providing swap space.

It consists of a number of lines of the form:

fsname dir type opts freq passno

an example of which would be:

/dev/xy0a / efs rw, 1 2

The entries from this file are accessed using the routines in *getmntent*(3), which returns a structure of the following form:

```
struct mntent {
        char    *mnt_fsname;   /* file system name */
        char    *mnt_dir;      /* file system path prefix */
        char    *mnt_type;     /* nfs, efs, or ignore */
        char    *mnt_opts;     /* ro, quota, etc. */
        int     mnt_freq;      /* dump frequency, in days */
        int     mnt_passno;    /* pass number on parallel fsck */
};
```

The fields are separated by white space, and a '#' as the first non-white character indicates a comment.

The *mnt_type* field determines how the *mnt_fsname,* and *mnt_opts* fields will be interpreted.

Below is a list of the file system types currently supported and the way each of them interprets these fields.

efs

mnt_fsname Must be a block special device.

mnt_opts Valid opts are ro, rw.

nfs

mnt_fsname The path on the server of the directory to be served.

mnt_opts Valid opts are ro, rw, hard, soft.

If the *mnt_type* is specified as "ignore" the entry is ignored. This is useful to show disk partitions which are currently not used.

mnt_freq and *mnt_passno* are not supported.

/etc/fstab is only *read* by programs, and not written; it is the duty of the system administrator to properly create and maintain this file. The order of records in */etc/fstab* is important because *mount* and *umount* process the file sequentially; file systems must appear *after* file systems they are mounted within.

FILES

/etc/fstab

SEE ALSO

fsck(1M), getmntent(3), mount(1M), umount(1M)

ORIGIN

Sun Microsystems

NAME

/etc/mtab – mounted file system table

SYNOPSIS

#include <mntent.h>

DESCRIPTION

*mtab file mounted file system table file system*mounted table *Mtab* resides in the */etc* directory, and contains a table of filesystems currently mounted by the *mount* command. *Umount* removes entries from this file.

The file contains a line of information for each mounted filesystem, structurally identical to the contents of */etc/fstab*, described in *fstab*(4). There are a number of lines of the form:

fsname dir type opts freq passno

for example:

/dev/xy0a / efs rw,noquota 1 2

The file is accessed by programs using *getmntent*(3), and by the system administrator using a text editor.

NOTES

You should not change */etc/mtab* by hand. This confuses the system and does not achieve the desired result.

FILES

/etc/mtab

SEE ALSO

getmntent(3), fstab(4), mount(1m)

ORIGIN

Sun Microsystems

NAME

netgroup – list of network groups

DESCRIPTION

netgroup defines network wide groups, used for permission checking when doing remote mounts, remote logins, and remote shells. For remote mounts, the information in *netgroup* is used to classify machines; for remote logins and remote shells, it is used to classify users. Each line of the *netgroup* file defines a group and has the format

groupname member1 member2 ....

where member*i* is either another group name, or a triple:

(hostname, username, domainname)

Any of three fields can be empty, in which case it signifies a wild card. Thus

universal (,,)

defines a group to which everyone belongs. Field names that begin with something other than a letter, digit or underscore (such as "-") work in precisely the opposite fashion. For example, consider the following entries:

```
justmachines   (analytica,-,sun)
justpeople     (-,babbage,sun)
```

The machine *analytica* belongs to the group *justmachines* in the domain *sun*, but no users belong to it. Similarly, the user *babbage* belongs to the group *justpeople* in the domain *sun*, but no machines belong to it.

Network groups are contained in the yellow pages, and are accessed through these files:

/etc/yp/*domainname*/netgroup.dir
/etc/yp/*domainname*/netgroup.pag
/etc/yp/*domainname*/netgroup.byuser.dir
/etc/yp/*domainname*/netgroup.byuser.pag
/etc/yp/*domainname*/netgroup.byhost.dir
/etc/yp/*domainname*/netgroup.byhost.pag

These files can be created from */etc/netgroup* using *makedbm*(1M).

FILES

/etc/netgroup
/etc/yp/*domainname*/netgroup.dir
/etc/yp/*domainname*/netgroup.pag
/etc/yp/*domainname*/netgroup.byuser.dir
/etc/yp/*domainname*/netgroup.byuser.pag
/etc/yp/*domainname*/netgroup.byhost.dir
/etc/yp/*domainname*/netgroup.byhost.pag

SEE ALSO

makedbm(1M)

ORIGIN

Sun Microsystems

NAME

networks – network name data base

DESCRIPTION

The *networks* file contains information regarding the known networks which comprise the DARPA Internet. For each network a single line should be present with the following information:

official network name
network number
aliases

Items are separated by any number of blanks and/or tab characters. A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file. This file is normally created from the official network data base maintained at the Network Information Control Center (NIC), though local changes may be required to bring it up to date regarding unofficial aliases and/or unknown networks.

Network number may be specified in the conventional "." notation using the *inet_network*() routine from the Internet address manipulation library, *inet*(3N). Network names may contain any printable character other than a field delimiter, newline, or comment character.

FILES

/etc/networks

SEE ALSO

getnetent(3N)

BUGS

A name server should be used instead of a static file. A binary indexed file format should be available for fast access.

ORIGIN

4.3 BSD

NAME

passwd – password file

SYNOPSIS

/etc/passwd

DESCRIPTION

passwd file

The *passwd* file contains for each user the following information:

name	User's login name – contains no upper case characters and must not be greater than eight characters long.
password	encrypted password
numerical user ID	This is the user's ID in the system and it must be unique.
numerical group ID	This is the number of the group that the user belongs to.
user's real name	In some versions of UNIX, this field also contains the user's office, extension, home phone, and so on. For historical reasons this field is called the GCOS field.
initial working directory	The directory that the user is positioned in when they log in – this is known as the 'home' directory.
shell	program to use as Shell when the user logs in.

The user's real name field may contain '&', meaning insert the login name.

The password file is an ASCII file. Each field within each user's entry is separated from the next by a colon. Each user is separated from the next by a new-line. If the password field is null, no password is demanded; if the Shell field is null, */bin/sh* is used.

The *passwd* file can also have a line beginning with a plus (+), which means to incorporate entries from the Yellow Pages. There are three styles of + entries: all by itself, + means to insert the entire contents of the Yellow Pages password file at that point; +*name* means to insert the entry (if any) for *name* from the Yellow Pages at that point; +@*name* means to insert the entries for all members of the network group *name* at that point. If a + entry has a non-null password, directory, gecos, or shell field, they will overide what is contained in the Yellow Pages. The numerical user ID and group ID fields cannot be overridden.

EXAMPLE

Here is a sample */etc/passwd* file:

```
root:q.mJzTnu8icF.:0:10:superuser:/:/bin/csh
tut:6k/7KCFRPNVXg:508:10:Bill Tuthill:/usr2/tut:/bin/csh
+john:
+@documentation:no-login:
+:::Guest
```

In this example, there are specific entries for users *root tut*, in case the Yellow Pages are out of order. The user *john* will have his password entry in the Yellow Pages incorporated without change; anyone in the netgroup *documentation* will have their password field disabled, and anyone else will be able to log in with their usual password, shell, and home directory, but with a gecos field of *guest*.

The password file resides in the */etc* directory. Because of the encrypted passwords, it has general read permission and can be used, for example, to map numerical user ID's to names.

Appropriate precautions must be taken to lock the */etc/passwd* file against simultaneous changes if it is to be edited with a text editor;

FILES

/etc/passwd

SEE ALSO

getpwent(3), login(1), crypt(3), passwd(1), group(4)

ORIGIN

Sun Microsystems

NAME

passwd.conf – configuration database for the passwd command

SYNOPSIS

/etc/passwd.conf

DESCRIPTION

When the command *passwd(1)* is executed, it will look for the file */etc/passwd.conf*, which must have move 0644, and be owned by userid 0 (root) and group 0 (root) (these restrictions are to assure that the file has not been tampered with). If the file exists and the restrictions are met, the file is read for information telling *passwd* how passwords must be constructed. This file is not supplied with the system as shipped, and must be created by the systems administrator to be used. Without the file, the standard System V rules apply. Each line contains keyword/value pairs, separated by whitespace. Leading and trailing whitespace is ignored. If a word begins with a #, the rest of the line is ignored as a comment. The following are the valid keywords, values, and defaults:

length This sets the minimum length for a password. The value must be in the range 1-8, inclusive. The default is 6.

minimum This is a synonym for **length**.

shift This says whether or not to allow the password to be the same as, a circular shift of, or teh reverse of the username. If no value, "yes", or "1" is specified, the password is not restricted in this way. If the value "no" or "0" is specified, the password is restricted. By default, this restriction is on.

differ This sets the number of positions by which the new and old passwords must differ; that is, the minimum number of characters which must be different between the old and new passwords. The value must be in the range 0-7, inclusive, and if it is more than the number of characters in the password, will be reduced to that number. The default is 3.

diffpos This is a synonym for **differ**.

alpha This specifies the number of characters in the password that must be alphabetic characters. The value must be in the range 0-8, inclusive, and if it is more than the minimum password length minus the number of special characters, it will be reduced to that value. The default is 2.

special This specifies the number of characters in the password that must be non-alphabetic characters. The value must be in the range 0-8, inclusive, and may be reduced or force a reduction in the minimum number of alphabetic characters if it is too large (the alphabetic character count is always reduced first). The default is 1.

nonalpha This is a synonym for **special**.

insist This says whether or not to allow the user to force a password by being insistent (typing the same password in a number of times; see **numinsist** below for information). If no value, "yes", or "1" is specified, insistence causes acceptance. If the value "no" or "0" is specified, nonconforming passwords are rejected despite insistence. By default, nonconforming passwords are rejected.

numinsist This sets the number of times the same password must be types in a row before it will be accepted. This value is meaningless if **insist** is not set. The value must be in the range 1-25, inclusive. The default is 1.

tries This specifies the total number of attempts that can be made to set the

password. The value must be in the range 1-25, inclusive. The default is 3.

retypes This sets the number of times a password can be retyped incorrectly before the program gives up (note that this is **not** the number of attempts at retyping the password after a valid new password is entered, but the number of times a valid new password can be entered followed by an incorrect retyping). The value must be in the range 1-25, inclusive. The default is 2.

bsd This is a special entry that tells *passwd* to set the configuration values to behave like the BSD *passwd* command. This is equivalent to giving the following entries:

```
minlength   5
shift
differ      0
alpha       0
special     0
insist
numinsist   3
tries       3
retypes     2
```

If a keyword is given more than once, the last value seen in the file is used. Case is ignored in both keywords and values, so **BSD** is equivalent to **bsd**, and **Yes** is equivalent to **yes**. When *passwd* reads the file, any errors encountered will reset all values back to their default values and the configuration file ignored. If the user is the super-user, messages are printed to aid in fixing the file. Otherwise, errors are silently ignored.

SEE ALSO

passwd(1), passwd(20).

NAME

pnch – file format for card images

DESCRIPTION

The PNCH format is a convenient representation for files consisting of card images in an arbitrary code.

A PNCH file is a simple concatenation of card records. A card record consists of a single control byte followed by a variable number of data bytes. The control byte specifies the number (which must lie in the range 0-80) of data bytes that follow. The data bytes are 8-bit codes that constitute the card image. If there are fewer than 80 data bytes, it is understood that the remainder of the card image consists of trailing blanks.

NAME

profile – setting up an environment at login time

SYNOPSIS

/etc/profile
$HOME/.profile

DESCRIPTION

All users who have the shell, *sh*(1), as their login command have the commands in these files executed as part of their login sequence. */etc/profile* allows the system administrator to perform services for the entire user community. Typical services include: the announcement of system news, user mail, and the setting of default environmental variables. It is not unusual for */etc/profile* to execute special actions for the **root** login or the *su*(1) command. The file *$HOME/.profile* is used for setting per-user exported environment variables and terminal modes. The following example is typical (except for the comments):

```
# Make some environment variables global
export MAIL PATH TERM
# Set file creation mask
umask 027
# Tell me when new mail comes in
MAIL=/usr/mail/$LOGNAME
# Add my /bin directory to the shell search sequence
PATH=$PATH:$HOME/bin
# Set terminal type
while :
do      echo "terminal: \c"
        read TERM
        if [ -f ${TERMINFO:-/usr/lib/terminfo}/?/$TERM ]
        then break
        elif [ -f /usr/lib/terminfo/?/$TERM ]
        then break
        else echo "invalid term $TERM" 1>&2
        fi
done
# Initialize the terminal and set tabs
# The environmental variable TERM must have been exported
# before the "tput init" command is executed.
tput init
# Set the erase character to backspace
stty erase '^H' echo
```

FILES

$HOME/.profile user-specific environment
/etc/profile system-wide environment

SEE ALSO

terminfo(4), environ(5), term(5).
env(1), login(1), mail(1), sh(1), stty(1), su(1), tput(1) in the *User's Reference Manual.*
su(1M) in the *System Administrator's Reference Manual.*
User's Guide.
Chapter 10 in the *Programmer's Guide.*

NOTES

Care must be taken in providing system-wide services in */etc/profile*. Personal *.profile* files are better for serving all but the most global needs.

NAME

protocols – protocol name data base

DESCRIPTION

The *protocols* file contains information regarding the known protocols used in the DARPA Internet. For each protocol a single line should be present with the following information:

official protocol name
protocol number
aliases

Items are separated by any number of blanks and/or tab characters. A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file.

Protocol names may contain any printable character other than a field delimiter, newline, or comment character.

FILES

/etc/protocols

SEE ALSO

getprotoent(3N)

BUGS

A name server should be used instead of a static file.

ORIGIN

4.3 BSD

NAME

queuedefs – at/batch/cron queue description file

SYNOPSIS

/usr/lib/cron/queuedefs

DESCRIPTION

The *queuedefs* file describes the characteristics of the queues managed by *cron*(1M). Each non-comment line in this file describes one queue. The format of the lines are as follows: -

q.[*njob***j**][*nice***n**][*nwait***w**]

The fields in this line are:

q — The name of the queue. **a** is the default queue for jobs started by *at*(1); **b** is the default queue for jobs started by *batch*(1); **c** is the default queue for jobs run from a **crontab** file.

njob — The maximum number of jobs that can be run simultaneously in that queue; if more than *njob* jobs are ready to run, only the first *njob* jobs will be run, and the others will be run as jobs that are currently running terminate. The default value is 100.

nice — The *nice*(1) value to give to all jobs in that queue that are not run with a user ID of super-user. The default value is 2.

nwait — The number of seconds to wait before rescheduling a job that was deferred because more than *njob* jobs were running in that job's queue, or because more than 25 jobs were running in all the queues. The default value is 60.

Lines beginning with # are comments, and are ignored.

EXAMPLE

```
a.4j1n
b.2j2n90w
```

This file specifies that the **a** queue, for *at* jobs, can have up to 4 jobs running simultaneously; those jobs will be run with a *nice* value of 1. As no *nwait* value was given, if a job cannot be run because too many other jobs are running *cron* will wait 60 seconds before trying again to run it. The **b** queue, for *batch* jobs, can have up to 2 jobs running simultaneously; those jobs will be run with a *nice* value of 2. If a job cannot be run because too many other jobs are running, *cron* will wait 90 seconds before trying again to run it. All other queues can have up to 100 jobs running simultaneously; they will be run with a *nice* value of 2, and if a job cannot be run because too many other jobs are running *cron* will wait 60 seconds before trying again to run it.

FILES

/usr/lib/cron/queuedefs

SEE ALSO

cron(1M)

NAME

rcsfile – format of RCS file

DESCRIPTION

An RCS file is an ASCII file. Its contents is described by the grammar below. The text is free format, i.e., spaces, tabs and new lines have no significance except in strings. Strings are enclosed by '@'. If a string contains a '@', it must be doubled.

The meta syntax uses the following conventions: '|' (bar) separates alternatives; '{' and '}' enclose optinal phrases; '{' and '}*' enclose phrases that may be repeated zero or more times; '{' and '}+' enclose phrases that must appear at least once and may be repeated; '<' and '>' enclose nonterminals.

```
<rcstext>     ::=  <admin> {<delta>}* <desc> {<deltatext>}*

<admin>       ::=  head          {<num>};
                   access        {<id>}*;
                   symbols       {<id> : <num>}*;
                   locks         {<id> : <num>}*;
                   comment       {<string>};

<delta>       ::=  <num>
                   date          <num>;
                   author        <id>;
                   state         {<id>};
                   branches      {<num>}*;
                   next          {<num>};

<desc>        ::=  desc          <string>

<deltatext>   ::=  <num>
                   log           <string>
                   text          <string>


<num>         ::=  {<digit>{.}}+

<digit>       ::=  0 | 1 | ... | 9

<id>          ::=  <letter>{<idchar>}*

<letter>      ::=  A | B | ... | Z | a | b | ... | z

<idchar>      ::=  Any printing ASCII character except space,
                   tab, carriage return, new line, and <special>.

<special>     ::=  ; | : | , | @

<string>      ::=  @{any ASCII character, with '@' doubled}*@
```

Identifiers are case sensitive. Keywords are in lower case only. The sets of keywords and identifiers may overlap.

The <delta> nodes form a tree. All nodes whose numbers consist of a single pair (e.g., 2.3, 2.1, 1.3, etc.) are on the "trunk", and are linked through the "next" field in order of decreasing numbers. The "head" field in the <admin> node points to the head of that sequence (i.e., contains the highest pair).

All <delta> nodes whose numbers consist of 2n fields (n≥2) (e.g., 3.1.1.1, 2.1.2.2, etc.) are linked as follows. All nodes whose first (2n)-1 number fields are identical are linked through the "next" field in order of increasing numbers. For each such sequence, the <delta> node whose number is identical to the first 2(n-1) number fields of the deltas on that sequence is called the branchpoint. The "branches" field of a node contains a list of the numbers of the first nodes of all sequences for which it is a branchpoint. This list is ordered in increasing numbers.

Example:

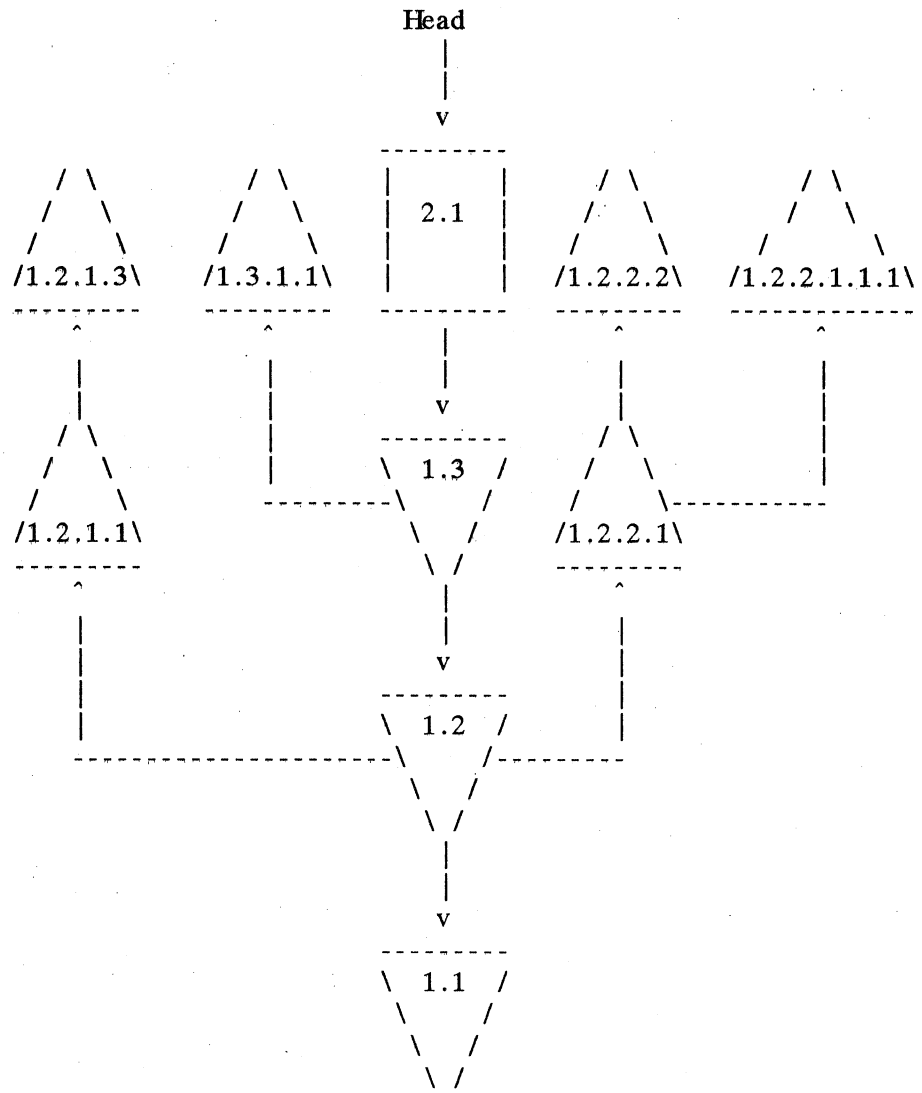


Fig. 1: A revision tree

IDENTIFICATION

Author: Walter F. Tichy, Purdue University, West Lafayette, IN, 47907.
Revision Number: 1.7 ; Release Date: 89/01/28 .
Copyright © 1982 by Walter F. Tichy.

SEE ALSO

ci (1), co (1), ident (1), rcs (1), rcsdiff (1), rcsintro (1), rcsmerge (1), rlog (1), sccstorcs (1M).

NAME

reloc – relocation information for a MIPS object file

SYNOPSIS

#include <reloc.h>

DESCRIPTION

Object files have one relocation entry for each relocatable reference in the text or data. If relocation information is present, it will be in the following format.

```
struct    reloc
{
        long            r_vaddr ;       /* (virtual) address of reference */
        long            r_symndx ;      /* index into symbol table */
        ushort          r_type ;        /* relocation type */
        unsigned        r_symndx:24,    /* index into symbol table */
                        r_reserved:3,
                        r_type:4,       /* relocation type */
                        r_extern:1;     /* if 1 symndx is an index into the external
                                           symbol table, else symndx is a section # */
} ;

/* Relocation types */
#define R_ABS           0
#define R_REFHALF       1
#define R_REFWORD       2
#define R_JMPADDR       3
#define R_REFHI         4
#define R_REFLO         5
#define R_GPREL         6
#define R_LITERAL       7

/* Section numbers */
#define R_SN_NULL       0
#define R_SN_TEXT       1
#define R_SN_RDATA      2
#define R_SN_DATA       3
#define R_SN_SDATA      4
#define R_SN_SBSS       5
#define R_SN_BSS        6
#define R_SN_INIT       7
#define R_SN_LIT8       8
#define R_SN_LIT4       9
```

The link editor reads each input section and performs relocation. The relocation entries direct how references found within the input section are treated.

If *r_extern* is zero then it is a local relocation entry and then *r_symndex* is a section number (R_SN_*). For these entries the starting address for the section referenced by the section number is used in place of an external symbol table entry's value. The assembler and loader always use local relocation entries if the item to be relocated is defined in the object file.

For every external relocation (except R_ABS) a signed constant is added to the symbol's virtual address that the relocation entry refers to. This constant is assembled at the address being relocated.

R_ABS	The reference is absolute and no relocation is necessary. The entry will be ignored.
R_REFHALF	A 16-bit reference to the symbol's virtual address.
R_REFWORD	A 32-bit reference to the symbol's virtual address.
R_JMPADDR	A 26-bit jump instruction reference to the symbol's virtual address.
R_REFHI	A reference to the high 16-bits of the symbol's virtual address. The next relocation entry must be the corresponding R_REFLO entry so the proper value of the constant to be added to the symbol's virtual address can be reconstructed.
R_REFLO	A reference to low 16-bits to the symbol's virtual address.
R_GPREL	A 16-bit offset to the symbol's virtual address from the global pointer register.
R_LITERAL	A 16-bit offset to the literal's virtual address from the global pointer register.

Relocation entries are generated automatically by the assembler and automatically used by the link editor. Link editor options exist for both preserving and removing the relocation entries from object files.

The number of relocation entries for a section is found in the *s_nreloc* field of the section header. This field is a 'C' language short and can overflow with large objects. If this field overflows the section header *s_flags* field has the S_NRELOC_OVFL bit set. In this case the true number of relocation entries is found in the *r_vaddr* field of the first relocation entry for that section. That relocation entry has a type of R_ABS so it is ignored when the relocation takes place. This is a kluge.

SEE ALSO

MIPS Assembly Language Programmer's Guide, Chapter 10 the section entitled "Section Relocation Information"
as(1), ld(1), a.out(4), syms(4), scnhdr(4).

NAME

rfmaster – Remote File Sharing name server master file

DESCRIPTION

The **rfmaster** file is an ASCII file that identifies the hosts that are responsible for providing primary and secondary domain name service for Remote File Sharing domains. This file contains a series of records, each terminated by a newline; a record may be extended over more than one line by escaping the newline character with a backslash ("\"). The fields in each record are separated by one or more tabs or spaces. Each record has three fields:

name type data The type field, which defines the meaning of the *name* and *data* fields, has three possible values:

p The **p** type defines the primary domain name server. For this type, *name* is the domain name and *data* is the full host name of the machine that is the primary name server. The full host name is specified as *domain.nodename*. There can be only one primary name server per domain.

s The **s** type defines a secondary name server for a domain. *Name* and *data* are the same as for the **p** type. The order of the **s** entries in the **rfmaster** file determines the order in which secondary name servers take over when the current domain name server fails.

a The **a** type defines a network address for a machine. *Name* is the full domain name for the machine and *data* is the network address of the machine. The network address can be in plain ASCII text or it can be preceded by a \x to be interpreted as hexadecimal notation. (See the documentation for the particular network you are using to determine the network addresses you need.) There are at least two lines in the **rfmaster** file per domain name server: one **p** and one **a** line, to define the primary and its network address. There should also be at least one secondary name server in each domain. This file is created and maintained on the primary domain name server. When a machine other than the primary tries to start Remote File Sharing, this file is read to determine the address of the primary. If **rfmaster** is missing, the **-p** option of **rfstart** must be used to identify the primary. After that, a copy of the primary's **rfmaster** file is automatically placed on the machine. Domains not served by the primary can also be listed in the **rfmaster** file. By adding primary, secondary, and address information for other domains on a network, machines served by the primary will be able to share resources with machines in other domains. A primary name server may be a primary for more than one domain. However, the secondaries must then also be the same for each domain served by the primary.

Example

An example of an **rfmaster** file is shown below. (The network address examples, *comp1.serve* and *comp2.serve*, are STARLAN network addresses.)

```
ccs          p     ccs.comp1
ccs          s     ccs.comp2
ccs.comp2    a     comp2.serve
ccs.comp1    a     comp1.serve
```

NOTE: If a line in the **rfmaster** file begins with a # character, the entire line will be treated as a comment.

FILES

/usr/nserve/rfmaster

SEE ALSO

rfstart(1M) in the *System Administrator's Reference Manual*.

NAME

rhosts – list of trusted hosts and users

DESCRIPTION

Each user may have a *.rhosts* file in his home directory. This file contains a list of users on other hosts in the network that are trusted in the following sense: when making requests to access the user's system with *rcp(1C)*, *rlogin(1C)*, or *rsh(1C)*, they are allowed to assume the user's identity without specifying a password. In other words, the remote user has exactly the same access privileges on the local system that the owner of the *.rhosts* file does and this access is granted without any attempt to verify the remote user's identity by requiring him to enter a password. The incoming request includes the user name that should be used on the local system. The *.rhosts* file owned by that local user acts as a logical extension to the *hosts.equiv(4)* file when deciding whether to grant permission for the incoming *rcp(1C)*, *rlogin(1C)*, or *rsh(1C)* request.

The *.rhosts* file has the same format as the *hosts.equiv(4)* file.

NOTES

The owner of the *.rhosts* file must be the user in whose home directory it resides. The contents of the file will be disregarded if it is owned by another user.

Special care should be taken in deciding the contents of the file */.rhosts*. Any host or user added to this file has the ability to become the superuser on the local system without entering the password. Note that */.rhosts* are not required.

FILES

$HOME/.rhosts

SEE ALSO

hosts.equiv(4)

ORIGIN

4.3 BSD

NAME

rmtab – remotely mounted file system table

DESCRIPTION

rmtab file rmtab resides in the directory */etc* and contains a record of all clients that have done remote mounts of file systems from this machine. Whenever a remote *mount* is done, an entry is made in the *rmtab* file of the machine serving up that file system. *umount* removes entries, if of a remotely mounted file system. *umount −a* broadcasts to all servers, and informs them that they should remove all entries from *rmtab* created by the sender of the broadcast message. By placing a *umount −a* command in */etc/rc.boot*, *rmtab* tables can be purged of entries made by a crashed host, which upon rebooting did not remount the same file systems it had before. The table is a series of lines of the form:

hostname:directory

This table is used only to preserve information between crashes, and is read only by *mountd*(1M) when it starts up. *mountd* keeps an in-core table, which it uses to handle requests from programs like *showmount*(1) and *shutdown*(1M).

FILES

/etc/rmtab

SEE ALSO

showmount(1), mountd(1M), mount(1M), umount(1M), shutdown(1M)

BUGS

Although the *rmtab* table is close to the truth, it is not always 100% accurate.

ORIGIN

Sun Microsystems

NAME

rpc – rpc program number data base

SYNOPSIS

/etc/rpc

DESCRIPTION

The *rpc* file contains user readable names that can be used in place of rpc program numbers. Each line has the following information:

name of server for the rpc program
rpc program number
aliases

Items are separated by any number of blanks and/or tab characters. A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file.

Here is an example of the */etc/rpc* file from the UNIX System.

```
#
#       rpc   1.2   86/01/07
#
rstatd          100001  rstat rup perfmeter
rusersd         100002  rusers
nfs             100003  nfsprog
ypserv          100004  ypprog
mountd          100005  mount showmount
ypbind          100007
walld           100008  rwall shutdown
yppasswdd       100009  yppasswd
sprayd          100012  spray
```

FILES

/etc/rpc

SEE ALSO

getrpcent(3N)

ORIGIN

Sun Microsystems

NAME

sccsfile – format of SCCS file

DESCRIPTION

An SCCS (Source Code Control System) file is an ASCII file. It consists of six logical parts: the *checksum*, the *delta table* (contains information about each delta), *user names* (contains login names and/or numerical group IDs of users who may add deltas), *flags* (contains definitions of internal keywords), *comments* (contains arbitrary descriptive information about the file), and the *body* (contains the actual text lines intermixed with control lines).

Throughout an SCCS file there are lines which begin with the **ASCII SOH** (start of heading) character (octal 001). This character is hereafter referred to as *the control character* and will be represented graphically as @. Any line described below which is not depicted as beginning with the control character is prevented from beginning with the control character.

Entries of the form
DDDDD represent a five-digit string (a number between 00000 and 99999).

Each logical part of an SCCS file is described in detail below.

checksum

The checksum is the first line of an SCCS file. The form of the line is:

@hDDDDD

The value of the checksum is the sum of all characters, except those of the first line. The **@h** provides a *magic number* of (octal) 064001.

delta table

The delta table consists of a variable number of entries of the form:

@s DDDDD/DDDDD/DDDDD
@d <type> <**SCCS ID**> yr/mo/da hr:mi:se <pgmr> **DDDDD D**
@i DDDDD ...
@x DDDDD ...
@g DDDDD ...
@m <**MR** number>
.
.
.
@c <comments> **...**
.
.
.
@e

The first line (**@s**) contains the number of lines inserted/deleted/unchanged, respectively. The second line (**@d**) contains the type of the delta (currently, normal: **D**, and removed: **R**), the SCCS ID of the delta, the date and time of creation of the delta, the login name corresponding to the real user ID at the time the delta was created, and the serial numbers of the delta and its predecessor, respectively.

The **@i**, **@x**, and **@g** lines contain the serial numbers of deltas included, excluded, and ignored, respectively. These lines are optional.

The **@m** lines (optional) each contain one **MR** number associated with the delta; the **@c** lines contain comments associated with the delta.

The **@e** line ends the delta table entry.

User names The list of login names and/or numerical group IDs of users who may add deltas to the file, separated by new-lines. The lines containing these login names and/or numerical group IDs are surrounded by the bracketing lines **@u** and **@U**. An empty list allows anyone to make a delta. Any line starting with a **!** prohibits the succeeding group or user from making deltas.

Flags Keywords used internally. [See *admin*(1) for more information on their use.] Each flag line takes the form:

@f <flag> <optional text>

The following flags are defined:

@f t <type of program>
@f v <program name>
@f i <keyword string>
@f b
@f m <module name>
@f f <floor>
@f c <ceiling>
@f d <default-sid>
@f n
@f j
@f l <lock-releases>
@f q <user defined>
@f z <reserved for use in interfaces>

The **t** flag defines the replacement for the **%Y%** identification keyword. The **v** flag controls prompting for **MR** numbers in addition to comments; if the optional text is present it defines an **MR** number validity checking program. The **i** flag controls the warning/error aspect of the "No id keywords" message. When the **i** flag is not present, this message is only a warning; when the **i** flag is present, this message will cause a "fatal" error (the file will not be gotten, or the delta will not be made). When the **b** flag is present the **−b** keyletter may be used on the *get* command to cause a branch in the delta tree. The **m** flag defines the first choice for the replacement text of the **%M%** identification keyword. The **f** flag defines the "floor" release; the release below which no deltas may be added. The **c** flag defines the "ceiling" release; the release above which no deltas may be added. The **d** flag defines the default SID to be used when none is specified on a *get* command. The **n** flag causes *delta* to insert a "null" delta (a delta that applies *no* changes) in those releases that are skipped when a delta is made in a *new* release (e.g., when delta 5.1 is made after delta 2.7, releases 3 and 4 are skipped). The absence of the **n** flag causes skipped releases to be completely empty. The **j** flag causes *get* to allow concurrent edits of the same base SID. The **l** flag defines a *list* of releases that are *locked* against editing [*get*(1) with the **−e** keyletter]. The **q** flag defines the replacement for the **%Q%** identification keyword. The **z** flag is used in certain specialized interface programs. *Comments* Arbitrary text is surrounded by the bracketing lines **@t** and **@T**. The comments section typically will contain a description of the file's purpose.

Body~~~~~

The body consists of text lines and control lines. Text lines do not begin with the control character, control lines do. There are three kinds of control lines: *insert*,~*delete*, and *end*, represented by:

@I DDDDD
@D DDDDD
@E DDDDD

respectively. The digit string is the serial number corresponding to the delta for the control line.

SEE ALSO

admin(1), delta(1), get(1), prs(1).

NAME

scnhdr – section header for a MIPS object file

SYNOPSIS

#include < scnhdr.h>

DESCRIPTION

Every MIPS object file has a table of section headers to specify the layout of the data within the file. Each section within an object file has its own header. The C structure appears below:

```
struct scnhdr
{
        char            s_name[8];  /* section name */
        long            s_paddr;    /* physical address, aliased s_nlib */
        long            s_vaddr;    /* virtual address */
        long            s_size;     /* section size */
        long            s_scnptr;   /* file ptr to raw data for section */
        long            s_relptr;   /* file ptr to relocation */
        long            s_lnnoptr;  /* file ptr to gp table */
        unsigned short  s_nreloc;   /* number of relocation entries */
        unsigned short  s_nlnno;    /* number of gp table entries */
        long            s_flags;    /* flags */
};
```

File pointers are byte offsets into the file; they can be used as the offset in a call to FSEEK [see *ldfcn*(4)]. If a section is initialized, the file contains the actual bytes. An uninitialized section is somewhat different. It has a size, symbols defined in it, and symbols that refer to it. But it can have no relocation entries or data. Consequently, an uninitialized section has no raw data in the object file, and the values for *s_scnptr, s_relptr,* and *s_nreloc* are zero.

The entries that refer to line numbers (*s_lnnoptr,* and *s_nlnno*) are not used for line numbers on MIPS machines. See the header file *sym.h* for the entries to get to the line number table. The entries that were for line numbers in the section header are used for gp tables on MIPS machines.

The number of relocation entries for a section is found in the *s_nreloc* field of the section header. This field being a 'C' language short and can overflow with large objects. If this field overflows the section header *s_flags* field has the S_NRELOC_OVFL bit set. In this case the true number of relocation entries is found in the *r_vaddr* field of the first relocation entry for that section. That relocation entry has a type of R_ABS so it is ignored when the relocation takes place. This is a kluge.

The gp table gives the section size corresponding to each applicable value of the compiler option **-G** *num* (always including 0), sorted by smallest size first. It is pointed to by the *s_lnnoptr* field in the section header and its number of entries (including the header) is in the *s_nlnno* field in the section header. This table only needs to exist for the *.sdata* and *.sbss* sections. If there is no "small" section then the gp table for it is attached to the coresponding "large" section so the information still gets to the link editor, *ld*(1). The C union for the gp table appears below.

```
union gp_table
{
        struct {
                long    current_g_value; /* actual value */
                long    unused;
        } header;
```

```
        struct {
                long    g_value; /* hypothetical value */
                long    bytes;   /* section size corresponding to hypothetical value */
        } entry;
};
```

Each gp table has one header structure that contains the actual value of the **−G** *num* option used to produce the object file. An entry must exist for every applicable value of the **−G** *num* option. The applicable values are all the sizes of the data items in that section.

For *.lib* sections the number of shared libraries is in the *s_nlib* field (an alias to *s_paddr*). The *.lib* section is made up of *s_nlib* descriptions of shared libraries. Each description of a shared library is a *libscn* structure followed by the path name to the shared library. The C structure appears below and is defined in *scnhdr.h* .

```
struct libscn
{
        long    size;           /* size of this entry (including target name) */
        long    offset;         /* offset from start of entry to target name */
        long    tsize;          /* text size in bytes, padded to DW boundary */
        long    dsize;          /* initialized data size */
        long    bsize;          /* uninitialized data */
        long    text_start;     /* base of text used for this library */
        long    data_start;     /* base of data used for this library */
        long    bss_start;      /* base of bss used for this library */
        /* pathname of target shared library */
};
```

SEE ALSO

ld(1), fseek(3S), a.out(4), reloc(4).

NAME

scr_dump – format of curses screen image file.

SYNOPSIS

scr_dump(file)

DESCRIPTION

The *curses*(3X) function *scr_dump*() will copy the contents of the screen into a file. The format of the screen image is as described below. The name of the tty is 20 characters long and the modification time (the *mtime* of the tty that this is an image of) is of the type *time_t*. All other numbers and characters are stored as *chtype* (see **<curses.h>**). No newlines are stored between fields.

```
<magic number:  octal 0433>
<name of tty>
<mod time of tty>
<columns> <lines>
<line length> <chars in line>   for each line on the screen
<line length> <chars in line>
  .
  .
  .
<labels?>                       1, if soft screen labels are present
<cursor row> <cursor column>
```

Only as many characters as are in a line will be listed. For example, if the *<line length>* is **0**, there will be no characters following *<line length>*. If *<labels?>* is TRUE, following it will be

```
<number of labels>
<label width>
<chars in label 1>
<chars in label 2>
  .
  .
  .
```

SEE ALSO

curses(3X).

NAME

sendmail.cf – sendmail configuration file

SYNOPSIS

/usr/lib/sendmail.cf

DESCRIPTION

The command */usr/lib/sendmail* reads */usr/lib/sendmail.cf* to obtain the site configuration information. This includes information such as which mailers are to be executed for various addressing styles, and hosts which are prepared to receive mail to be forwarded.

The form and content of this file is too complicated for this manual (it is also too complicated for most people to understand). The best place to start is with the *Sendmail Installation and Operations Guide* in the *System Administrator's Guide*.

SEE ALSO

sendmail(1M)

NAME

services – service name data base

DESCRIPTION

The *services* file contains information regarding the known services available in the DARPA Internet. For each service a single line should be present with the following information:

official service name
port number
protocol name
aliases

Items are separated by any number of blanks and/or tab characters. The port number and protocol name are considered a single *item*; a "/" is used to separate the port and protocol (e.g. "512/tcp"). A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file.

Service names may contain any printable character other than a field delimiter, newline, or comment character.

FILES

/etc/services

SEE ALSO

getservent(3N)

ORIGIN

4.3 BSD

NAME

su_people – special access database for su

SYNOPSIS

/etc/su_people

DESCRIPTION

When *su(1M)* is executed such that the user being subsituted is root (userid 0), the file */etc/su_people* is searched to see if the user executing the command or the user logged in originally (if these are different) is priveleged enough not to have to give the password (this is called having free access). This is done as a convenience, and should not be taken lightly.

In order to stop any possible security hazards with this feature, */etc/su_people* must have mode 0600 (read and write for owner only), owner 0 (root), and group 0 (root) or it will be ignored. In addition, if any syntax errors are found in the file, free access will be denied.

There are a number of different types of lines that can be placed in this file:

#text — Comment. This line is ignored.

username — The named user is allowed free access.

username hostname_list
: The named user is allowed free access on the hosts named in *hostname_list*, which is a list of hostnames separated by spaces, tabs, and/or commas.

username !hostname_list
: The named user is denied free access on the hosts named in *hostname_list*, which is a list of hostnames separated by spaces, tabs, and/or commas.

SEE ALSO

su(1M)

NAME

syms – MIPS symbol table

SYNOPSIS

#include < sym.h>
#include < symconst.h>

DESCRIPTION

The MIPS symbol table departs from the standard COFF symbol table. The symbol table consists of many tables unbundling information usually found in the one COFF symbol table. The symbol table should be viewed as a hand-crafted, network-style database designed for space and access efficiency.

The following structures or tables appear in the MIPS symbol table:

TABLE	CONTENTS
symbolic header	sizes and locations of all other tables.
file descriptors	per file locations for other tables.
procedure descriptors	frame information and location of procedure info.
local symbols	local type, local variable, and scoping info.
local strings	string space for local symbols.
line numbers	compacted by encoding, contains a line per instruction.
relative file desc.	indirection for inter-file symbol access.
optimzation symbols	to be defined.
auxiliary symbols	variable data type information for each local symbol.
external symbols	loader symbols (global text and data).
external strings	string space for external symbols.
dense numbers	(file, symbol) index pairs for compiler use.

External and local symbols contain the standard concept of a "symbol" as follows:

```
struct
{
        long        iss;        /* index into string space */
        long        value;      /* address, size, etc., depends on sc and st */
        unsigned    st: 6;      /* symbol type (e.g. local, param, etc.) */
        unsigned    sc: 5;      /* storage class (e.g. text, bss, etc.) */
        unsigned    reserved: 1;
        unsigned    index;      /* index to symbol or auxiliary tables */
};
```

SEE ALSO

The chapter on "The Symbol Table" in the *MIPS Assembly Language Programmer's Guide.*
ldfcn(4).

NAME

system – system configuration information table

DESCRIPTION

This file is used by the **lboot** or **mboot** program to obtain configuration information. This file generally contains information used to determine if specified hardware exists, a list of software drivers to include in the load, the assignment of system devices such as *pipedev* and *swapdev,* as well as instructions for manually overriding the drivers selected by the self-configuring boot process.

The syntax of the system file is given below. The parser for the **system** file is case sensitive. All upper case strings in the syntax below should be upper case in the **system** file as well. Nonterminal symbols are enclosed in angle brackets "<>" while optional arguments are enclosed in square brackets "[]". Ellipses "..." indicate optional repetition of the argument for that line.

<fname> ::= master file name from */master.d* directory
<func> ::= interrupt function name
<device> ::= special device name | DEV(<major>,<minor>)
<major> ::= <number>
<minor> ::= <number>
<number> ::= decimal, octal or hex literal

The lines listed below may appear in any order. Blank lines may be inserted at any point. Comment lines must begin with an asterisk. Entries for VECTOR, EXCLUDE and INCLUDE are cumulative. For all other entries, the last line to appear in the file is used – any earlier entries are ignored.

VECTOR: (Note: this is one line) module=<fname> [intr=<func>]
[vector=<number> ipl=<number> unit=<number> base=<number>]
[probe=<number> [probe_size=<number>]]

specifies hardware to conditionally load. If a probe address is specified, the boot program will read probe_size bytes (default 4) to determine if the hardware exists for the module. If so, the module is included. If a probe address is not specified, the hardware will be assumed to exist. The intr function specifies the name of the module's interrupt handler. If it is not specified, the prefix defined in the module's master file (see *master(4)*) is concatenated with the string "intr", and, if a routine with that name is found in the module's object (which resides in the boot directory, typically **$ROOT/usr/src/uts/mips/bootarea**), it is used as the interrupt routine. If the quadruplet (vector, ipl, unit, base) is specified, a VME interrupt structure is assigned, using the corresponding VME address "vector", priority level "ipl", unit "unit", and accessing the device beginning at memory location "base".

EXCLUDE: [<string>] ...

specifies drivers to exclude from the load even if the device is found via VECTOR information.

INCLUDE: [<string>[(<number>)]] ...

specifies software drivers or loadable modules to be included in the load. This is necessary to include the drivers for software "devices". The optional <number> (parenthesis required) specifies the number of "devices" to be controlled by the driver (defaults to 1). This number corresponds to the builtin variable *##c* which may be referred to by expressions in part two of the **master** file.

ROOTDEV: <device>

identifies the device containing the root file system.

SWAPDEV: <device> <number> <number>

identifies the device to be used as swap space, the block number the swap space starts at, and the number of swap blocks available.

PIPEDEV: <device>
identifies the device to be used for pipe space.

DUMPDEV: <device>
identifies the device to be used for kernel dumps.

USE: [<string>[(<number>)]] ...
If the driver is present, it is the same as INCLUDE. Behaves like EXCLUDE if the module or driver is not present in boot directory, typically **$ROOT/usr/src/uts/mips/bootarea.**

KERNEL: [<string>] ...
Specifies the module containing the heart of the operating system. It must be present in the system file.

LCOPTS
LDOPTS
are option strings given to *cc*(1) and *ld*(1) respectively, to compile the master.c file and link the operating system.

FILES

$ROOT/usr/src/uts/mips/master.d/system[.suffix]

SEE ALSO

master(4), lboot(1M)

NAME

tar – tape archive file format

DESCRIPTION

Tar, (the tape archive command) dumps several files into one, in a medium suitable for transportation.

A "tar tape" or file is a series of blocks. Each block is of size TBLOCK. A file on the tape is represented by a header block which describes the file, followed by zero or more blocks which give the contents of the file. At the end of the tape are two blocks filled with binary zeros, as an end-of-file indicator.

The blocks are grouped for physical I/O operations. Each group of *n* blocks (where *n* is set by the **b** keyletter on the *tar*(1) command line – default is 20 blocks) is written with a single system call; on nine-track tapes, the result of this write is a single tape record. The last group is always written at the full size, so blocks after the two zero blocks contain random data. On reading, the specified or default group size is used for the first read, but if that read returns less than a full tape block, the reduced block size is used for further reads.

The header block looks like:

```
#define TBLOCK      512
#define NAMSIZ      100

union hblock {
        char dummy[TBLOCK];
        struct header {
                char name[NAMSIZ];
                char mode[8];
                char uid[8];
                char gid[8];
                char size[12];
                char mtime[12];
                char chksum[8];
                char linkflag;
                char linkname[NAMSIZ];
        } dbuf;
};
```

name is a null-terminated string. The other fields are zero-filled octal numbers in ASCII. Each field (of width w) contains w-2 digits, a space, and a null, except *size* and *mtime* , which do not contain the trailing null and *chksum* which has a null followed by a space. *name* is the name of the file, as specified on the *tar* command line. Files dumped because they were in a directory which was named in the command line have the directory name as prefix and */filename* as suffix. *mode* is the file mode, with the top bit masked off. *uid* and *gid* are the user and group numbers which own the file. *size* is the size of the file in bytes. Links and symbolic links are dumped with this field specified as zero. *mtime* is the modification time of the file at the time it was dumped. *chksum* is an octal ASCII value which represents the sum of all the bytes in the header block. When calculating the checksum, the *chksum* field is treated as if it were all blanks. *linkflag* is NULL if the file is "normal" or a special file, ASCII '1' if it is an hard link, and ASCII '1' if it is a symbolic link. The name linked-to, if any, is in *linkname,* with a trailing null. Unused fields of the header are binary zeros (and are included in the checksum).

The first time a given i-node number is dumped, it is dumped as a regular file. The second and subsequent times, it is dumped as a link instead. Upon retrieval, if a link entry is retrieved, but not the file it was linked to, an error message is printed and the tape must be

manually re-scanned to retrieve the linked-to file.

The encoding of the header is designed to be portable across machines.

SEE ALSO

tar(1)

BUGS

Names or linknames longer than NAMSIZ produce error reports and cannot be dumped.

NAME

term – format of compiled term file.

SYNOPSIS

/usr/lib/terminfo/?/*

DESCRIPTION

Compiled *terminfo*(4) descriptions are placed under the directory */usr/lib/terminfo*. In order to avoid a linear search of a huge UNIX system directory, a two-level scheme is used: */usr/lib/terminfo/c/name* where *name* is the name of the terminal, and *c* is the first character of *name*. Thus, **att4425** can be found in the file */usr/lib/terminfo/a/att4425*. Synonyms for the same terminal are implemented by multiple links to the same compiled file.

The format has been chosen so that it will be the same on all hardware. An 8-bit byte is assumed, but no assumptions about byte ordering or sign extension are made. Thus, these binary *terminfo*(4) files can be transported to other hardware with 8-bit bytes.

Short integers are stored in two 8-bit bytes. The first byte contains the least significant 8 bits of the value, and the second byte contains the most significant 8 bits. (Thus, the value represented is 256*second+first.) The value **−1** is represented by **0377,0377**, and the value **−2** is represented by **0376,0377**; other negative values are illegal. Computers where this does not correspond to the hardware read the integers as two bytes and compute the result, making the compiled entries portable between machine types. The **−1** generally means that a capability is missing from this terminal. The **−2** means that the capability has been cancelled in the *terminfo*(4) source and also is to be considered missing.

The compiled file is created from the source file descriptions of the terminals (see the **−I** option of *infocmp*(1M)) by using the *terminfo*(4) compiler, *tic*(1M), and read by the routine **setupterm**( ). (See *curses*(3X).) The file is divided into six parts: the header, terminal names, boolean flags, numbers, strings, and string table.

The header section begins the file. This section contains six short integers in the format described below. These integers are (1) the magic number (octal **0432**); (2) the size, in bytes, of the names section; (3) the number of bytes in the boolean section; (4) the number of short integers in the numbers section; (5) the number of offsets (short integers) in the strings section; (6) the size, in bytes, of the string table.

The terminal names section comes next. It contains the first line of the *terminfo*(4) description, listing the various names for the terminal, separated by the bar (|) character (see *term*(5)). The section is terminated with an ASCII NUL character.

The boolean flags have one byte for each flag. This byte is either **0** or **1** as the flag is present or absent. The value of **2** means that the flag has been cancelled. The capabilities are in the same order as the file <**term.h**>.

Between the boolean section and the number section, a null byte will be inserted, if necessary, to ensure that the number section begins on an even byte. All short integers are aligned on a short word boundary.

The numbers section is similar to the boolean flags section. Each capability takes up two bytes, and is stored as a short integer. If the value represented is **−1** or **−2**, the capability is taken to be missing.

The strings section is also similar. Each capability is stored as a short integer, in the format above. A value of **−1** or **−2** means the capability is missing. Otherwise, the value is taken as an offset from the beginning of the string table. Special characters in ^X or \c notation are stored in their interpreted form, not the printing representation. Padding information ($<nn>) and parameter information (%x) are stored intact in uninterpreted form.

The final section is the string table. It contains all the values of string capabilities referenced in the string section. Each string is null terminated.

Note that it is possible for **setupterm**( ) to expect a different set of capabilities than are actually present in the file. Either the database may have been updated since **setupterm**( ) has been recompiled (resulting in extra unrecognized entries in the file) or the program may have been recompiled more recently than the database was updated (resulting in missing entries). The routine **setupterm**( ) must be prepared for both possibilities – this is why the numbers and sizes are included. Also, new capabilities must always be added at the end of the lists of boolean, number, and string capabilities.

As an example, an octal dump of the description for the AT&T Model 37 KSR is included:

```
37|tty37|AT&T model 37 teletype,
    hc, os, xon,
    bel=^G, cr=\r, cub1=\b, cud1=\n, cuu1=\E7, hd=\E9,
    hu=\E8, ind=\n,

0000000 032 001     \0 032  \0 013  \0 021 001   3 \0   3   7   |   t
0000020   t   y   3   7   |   A   T   &   T       m   o   d   e   l
0000040   3   7       t   e   l   e   t   y   p   e \0 \0 \0 \0 \0
0000060 \0 \0 \0 001 \0 \0 \0 \0 \0 \0 \0 001 \0 \0 \0 \0
0000100 001 \0 \0 \0 \0 \0 377 377 377 377 377 377 377 377 377 377
0000120 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377   &  \0
0000140     \0 377 377 377 377 377 377 377 377 377 377 377 377 377 377
0000160 377 377   "  \0 377 377 377 377   (  \0 377 377 377 377 377 377
0000200 377 377   0  \0 377 377 377 377 377 377 377 377   -  \0 377 377
0000220 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377
*
0000520 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377   $  \0
0000540 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377   *  \0
0000560 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377
*
0001160 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377   3   7
0001200   |   t   t   y   3   7   |   A   T   &   T       m   o   d   e
0001220   l       3   7       t   e   l   e   t   y   p   e \0  \r \0
0001240  \n \0  \n \0 007  \0  \b \0 033   8 \0 033   9 \0 033   7
0001260  \0 \0
0001261
```

Some limitations: total compiled entries cannot exceed 4096 bytes; all entries in the name field cannot exceed 128 bytes.

FILES

/usr/lib/terminfo/?/*	compiled terminal description database
/usr/include/term.h	*terminfo*(4) header file

SEE ALSO

curses(3X), terminfo(4), term(5).
infocmp(1M) in the *System Administrator's Reference Manual*.
Chapter 10 of the *Programmer's Guide*.

NAME

terminfo – terminal capability data base

SYNOPSIS

/usr/lib/terminfo/?/*

DESCRIPTION

terminfo is a compiled database (see *tic*(1M)) describing the capabilities of terminals. Terminals are described in *terminfo* source descriptions by giving a set of capabilities which they have, by describing how operations are performed, by describing padding requirements, and by specifying initialization sequences. This database is used by applications programs, such as *vi*(1) and *curses*(3X), so they can work with a variety of terminals without changes to the programs. To obtain the source description for a terminal, use the **-I** option of *infocmp*(1M).

Entries in *terminfo* source files consist of a number of comma-separated fields. White space after each comma is ignored. The first line of each terminal description in the *terminfo* database gives the name by which *terminfo* knows the terminal, separated by bar (|) characters. The first name given is the most common abbreviation for the terminal (this is the one to use to set the environment variable **TERM** in *$HOME/.profile*; see *profile*(4)), the last name given should be a long name fully identifying the terminal, and all others are understood as synonyms for the terminal name. All names but the last should contain no blanks and must be unique in the first 14 characters; the last name may contain blanks for readability. Terminal names (except for the last, verbose entry) should be chosen using the following conventions. The particular piece of hardware making up the terminal should have a root name chosen, for example, for the AT&T 4425 terminal, **att4425**. Modes that the hardware can be in, or user preferences, should be indicated by appending a hyphen and an indicator of the mode. See *term*(5) for examples and more information on choosing names and synonyms.

CAPABILITIES

In the table below, the **Variable** is the name by which the C programmer (at the *terminfo* level) accesses the capability. The **Capname** is the short name for this variable used in the text of the database. It is used by a person updating the database and by the *tput*(1) command when asking what the value of the capability is for a particular terminal. The **Termcap Code** is a two-letter code that corresponds to the old *termcap* capability name. Capability names have no hard length limit, but an informal limit of 5 characters has been adopted to keep them short. Whenever possible, names are chosen to be the same as or similar to the ANSI X3.64-1979 standard. Semantics are also intended to match those of the specification. All string capabilities listed below may have padding specified, with the exception of those used for input. Input capabilities, listed under the **Strings** section in the table below, have names beginning with **key_**. The following indicators may appear at the end of the **Description** for a variable.

(G)	indicates that the string is passed through **tparm**( ) with parameters (parms) as given ($\#_i$).
(*)	indicates that padding may be based on the number of lines affected.
($\#_i$)	indicates the i^{th} parameter.

Variable	Cap-name	Termcap Code	Description
Booleans:			
auto_left_margin	bw	bw	**cub**1 wraps from column 0 to last column
auto_right_margin	am	am	Terminal has automatic margins
no_esc_ctlc	xsb	xb	Beehive (f1=escape, f2=ctrl C)
ceol_standout_glitch	xhp	xs	Standout not erased by overwriting (hp)
eat_newline_glitch	xenl	xn	Newline ignored after 80 cols (*Concept*)
erase_overstrike	eo	eo	Can erase overstrikes with a blank
generic_type	gn	gn	Generic line type (e.g. dialup, switch).
hard_copy	hc	hc	Hardcopy terminal
hard_cursor	chts	HC	Cursor is hard to see.
has_meta_key	km	km	Has a meta key (shift, sets parity bit)
has_status_line	hs	hs	Has extra "status line"
insert_null_glitch	in	in	Insert mode distinguishes nulls
memory_above	da	da	Display may be retained above the screen
memory_below	db	db	Display may be retained below the screen
move_insert_mode	mir	mi	Safe to move while in insert mode
move_standout_mode	msgr	ms	Safe to move in standout modes
needs_xon_xoff	nxon	nx	Padding won't work, xon/xoff required
non_rev_rmcup	nrrmc	NR	**smcup** does not reverse **rmcup**
no_pad_char	npc	NP	Pad character doesn't exist
over_strike	os	os	Terminal overstrikes on hard-copy terminal
prtr_silent	mc5i	5i	Printer won't echo on screen.
status_line_esc_ok	eslok	es	Escape can be used on the status line
dest_tabs_magic_smso	xt	xt	Destructive tabs, magic **smso** char (t1061)
tilde_glitch	hz	hz	Hazeltine; can't print tildes(~)
transparent_underline	ul	ul	Underline character overstrikes
xon_xoff	xon	xo	Terminal uses xon/xoff handshaking
Numbers:			
columns	cols	co	Number of columns in a line
init_tabs	it	it	Tabs initially every # spaces.
label_height	lh	lh	Number of rows in each label
label_width	lw	lw	Number of cols in each label
lines	lines	li	Number of lines on screen or page
lines_of_memory	lm	lm	Lines of memory if > **lines**; **0** means varies
magic_cookie_glitch	xmc	sg	Number blank chars left by **smso** or **rmso**
num_labels	nlab	Nl	Number of labels on screen (start at 1)
padding_baud_rate	pb	pb	Lowest baud rate where padding needed
virtual_terminal	vt	vt	Virtual terminal number (UNIX system)
width_status_line	wsl	ws	Number of columns in status line
Strings:			
acs_chars	acsc	ac	Graphic charset pairs aAbBcC - def=vt100+
back_tab	cbt	bt	Back tab
bell	bel	bl	Audible signal (bell)
carriage_return	cr	cr	Carriage return (*)
change_scroll_region	csr	cs	Change to lines #1 thru #2 (vt100) (G)
char_padding	rmp	rP	Like **ip** but when in replace mode
clear_all_tabs	tbc	ct	Clear all tab stops

clear_margins	mgc	MC	Clear left and right soft margins
clear_screen	clear	cl	Clear screen and home cursor (*)
clr_bol	el1	cb	Clear to beginning of line, inclusive
clr_eol	el	ce	Clear to end of line
clr_eos	ed	cd	Clear to end of display (*)
column_address	hpa	ch	Horizontal position absolute (G)
command_character	cmdch	CC	Term. settable cmd char in prototype
cursor_address	cup	cm	Cursor motion to row #1 col #2 (G)
cursor_down	cud1	do	Down one line
cursor_home	home	ho	Home cursor (if no **cup**)
cursor_invisible	civis	vi	Make cursor invisible
cursor_left	cub1	le	Move cursor left one space.
cursor_mem_address	mrcup	CM	Memory relative cursor addressing (G)
cursor_normal	cnorm	ve	Make cursor appear normal (undo **vs/vi**)
cursor_right	cuf1	nd	Non-destructive space (cursor right)
cursor_to_ll	ll	ll	Last line, first column (if no **cup**)
cursor_up	cuu1	up	Upline (cursor up)
cursor_visible	cvvis	vs	Make cursor very visible
delete_character	dch1	dc	Delete character (*)
delete_line	dl1	dl	Delete line (*)
dis_status_line	dsl	ds	Disable status line
down_half_line	hd	hd	Half-line down (forward 1/2 linefeed)
ena_acs	enacs	eA	Enable alternate char set
enter_alt_charset_mode	smacs	as	Start alternate character set
enter_am_mode	smam	SA	Turn on automatic margins
enter_blink_mode	blink	mb	Turn on blinking
enter_bold_mode	bold	md	Turn on bold (extra bright) mode
enter_ca_mode	smcup	ti	String to begin programs that use **cup**
enter_delete_mode	smdc	dm	Delete mode (enter)
enter_dim_mode	dim	mh	Turn on half-bright mode
enter_insert_mode	smir	im	Insert mode (enter);
enter_protected_mode	prot	mp	Turn on protected mode
enter_reverse_mode	rev	mr	Turn on reverse video mode
enter_secure_mode	invis	mk	Turn on blank mode (chars invisible)
enter_standout_mode	smso	so	Begin standout mode
enter_underline_mode	smul	us	Start underscore mode
enter_xon_mode	smxon	SX	Turn on xon/xoff handshaking
erase_chars	ech	ec	Erase #1 characters (G)
exit_alt_charset_mode	rmacs	ae	End alternate character set
exit_am_mode	rmam	RA	Turn off automatic margins
exit_attribute_mode	sgr0	me	Turn off all attributes
exit_ca_mode	rmcup	te	String to end programs that use **cup**
exit_delete_mode	rmdc	ed	End delete mode
exit_insert_mode	rmir	ei	End insert mode;
exit_standout_mode	rmso	se	End standout mode
exit_underline_mode	rmul	ue	End underscore mode
exit_xon_mode	rmxon	RX	Turn off xon/xoff handshaking
flash_screen	flash	vb	Visible bell (may not move cursor)
form_feed	ff	ff	Hardcopy terminal page eject (*)
from_status_line	fsl	fs	Return from status line
init_1string	is1	i1	Terminal initialization string
init_2string	is2	is	Terminal initialization string

init_3string	is3	i3	Terminal initialization string
init_file	if	if	Name of initialization file containing **is**
init_prog	iprog	iP	Path name of program for init.
insert_character	ich1	ic	Insert character
insert_line	il1	al	Add new blank line (*)
insert_padding	ip	ip	Insert pad after character inserted (*)
key_a1	ka1	K1	KEY_A1, 0534, Upper left of keypad
key_a3	ka3	K3	KEY_A3, 0535, Upper right of keypad
key_b2	kb2	K2	KEY_B2, 0536, Center of keypad
key_backspace	kbs	kb	KEY_BACKSPACE, 0407, Sent by backspace ky
key_beg	kbeg	@1	KEY_BEG, 0542, Sent by beg(inning) key
key_btab	kcbt	kB	KEY_BTAB, 0541, Sent by back-tab key
key_c1	kc1	K4	KEY_C1, 0537, Lower left of keypad
key_c3	kc3	K5	KEY_C3, 0540, Lower right of keypad
key_cancel	kcan	@2	KEY_CANCEL, 0543, Sent by cancel key
key_catab	ktbc	ka	KEY_CATAB, 0526, Sent by clear-all-tabs key
key_clear	kclr	kC	KEY_CLEAR, 0515, Sent by clear-screen or erase key
key_close	kclo	@3	KEY_CLOSE, 0544, Sent by close key
key_command	kcmd	@4	KEY_COMMAND, 0545, Sent by cmd (command) key
key_copy	kcpy	@5	KEY_COPY, 0546, Sent by copy key
key_create	kcrt	@6	KEY_CREATE, 0547, Sent by create key
key_ctab	kctab	kt	KEY_CTAB, 0525, Sent by clear-tab key
key_dc	kdch1	kD	KEY_DC, 0512, Sent by delete-character key
key_dl	kdl1	kL	KEY_DL, 0510, Sent by delete-line key
key_down	kcud1	kd	KEY_DOWN, 0402, Sent by terminal down-arrow key
key_eic	krmir	kM	KEY_EIC, 0514, Sent by **rmir** or **smir** in insert m
key_end	kend	@7	KEY_END, 0550, Sent by end key
key_enter	kent	@8	KEY_ENTER, 0527, Sent by enter/send key
key_eol	kel	kE	KEY_EOL, 0517, Sent by clear-to-end-of-line key
key_eos	ked	kS	KEY_EOS, 0516, Sent by clear-to-end-of-screen k
key_exit	kext	@9	KEY_EXIT, 0551, Sent by exit key
key_f0	kf0	k0	KEY_F(0), 0410, Sent by function key f0
key_f1	kf1	k1	KEY_F(1), 0411, Sent by function key f1
key_f2	kf2	k2	KEY_F(2), 0412, Sent by function key f2
key_f3	kf3	k3	KEY_F(3), 0413, Sent by function key f3
key_f4	kf4	k4	KEY_F(4), 0414, Sent by function key f4
key_f5	kf5	k5	KEY_F(5), 0415, Sent by function key f5
key_f6	kf6	k6	KEY_F(6), 0416, Sent by function key f6
key_f7	kf7	k7	KEY_F(7), 0417, Sent by function key f7
key_f8	kf8	k8	KEY_F(8), 0420, Sent by function key f8
key_f9	kf9	k9	KEY_F(9), 0421, Sent by function key f9
key_f10	kf10	k;	KEY_F(10), 0422, Sent by function key f10
key_f11	kf11	F1	KEY_F(11), 0423, Sent by function key f11
key_f12	kf12	F2	KEY_F(12), 0424, Sent by function key f12
key_f13	kf13	F3	KEY_F(13), 0425, Sent by function key f13
key_f14	kf14	F4	KEY_F(14), 0426, Sent by function key f14
key_f15	kf15	F5	KEY_F(15), 0427, Sent by function key f15
key_f16	kf16	F6	KEY_F(16), 0430, Sent by function key f16
key_f17	kf17	F7	KEY_F(17), 0431, Sent by function key f17
key_f18	kf18	F8	KEY_F(18), 0432, Sent by function key f18
key_f19	kf19	F9	KEY_F(19), 0433, Sent by function key f19
key_f20	kf20	FA	KEY_F(20), 0434, Sent by function key f20

key_f21	kf21	FB	KEY_F(21), 0435, Sent by function key f21
key_f22	kf22	FC	KEY_F(22), 0436, Sent by function key f22
key_f23	kf23	FD	KEY_F(23), 0437, Sent by function key f23
key_f24	kf24	FE	KEY_F(24), 0440, Sent by function key f24
key_f25	kf25	FF	KEY_F(25), 0441, Sent by function key f25
key_f26	kf26	FG	KEY_F(26), 0442, Sent by function key f26
key_f27	kf27	FH	KEY_F(27), 0443, Sent by function key f27
key_f28	kf28	FI	KEY_F(28), 0444, Sent by function key f28
key_f29	kf29	FJ	KEY_F(29), 0445, Sent by function key f29
key_f30	kf30	FK	KEY_F(30), 0446, Sent by function key f30
key_f31	kf31	FL	KEY_F(31), 0447, Sent by function key f31
key_f32	kf32	FM	KEY_F(32), 0450, Sent by function key f32
key_f33	kf33	FN	KEY_F(13), 0451, Sent by function key f13
key_f34	kf34	FO	KEY_F(34), 0452, Sent by function key f34
key_f35	kf35	FP	KEY_F(35), 0453, Sent by function key f35
key_f36	kf36	FQ	KEY_F(36), 0454, Sent by function key f36
key_f37	kf37	FR	KEY_F(37), 0455, Sent by function key f37
key_f38	kf38	FS	KEY_F(38), 0456, Sent by function key f38
key_f39	kf39	FT	KEY_F(39), 0457, Sent by function key f39
key_f40	kf40	FU	KEY_F(40), 0460, Sent by function key f40
key_f41	kf41	FV	KEY_F(41), 0461, Sent by function key f41
key_f42	kf42	FW	KEY_F(42), 0462, Sent by function key f42
key_f43	kf43	FX	KEY_F(43), 0463, Sent by function key f43
key_f44	kf44	FY	KEY_F(44), 0464, Sent by function key f44
key_f45	kf45	FZ	KEY_F(45), 0465, Sent by function key f45
key_f46	kf46	Fa	KEY_F(46), 0466, Sent by function key f46
key_f47	kf47	Fb	KEY_F(47), 0467, Sent by function key f47
key_f48	kf48	Fc	KEY_F(48), 0470, Sent by function key f48
key_f49	kf49	Fd	KEY_F(49), 0471, Sent by function key f49
key_f50	kf50	Fe	KEY_F(50), 0472, Sent by function key f50
key_f51	kf51	Ff	KEY_F(51), 0473, Sent by function key f51
key_f52	kf52	Fg	KEY_F(52), 0474, Sent by function key f52
key_f53	kf53	Fh	KEY_F(53), 0475, Sent by function key f53
key_f54	kf54	Fi	KEY_F(54), 0476, Sent by function key f54
key_f55	kf55	Fj	KEY_F(55), 0477, Sent by function key f55
key_f56	kf56	Fk	KEY_F(56), 0500, Sent by function key f56
key_f57	kf57	Fl	KEY_F(57), 0501, Sent by function key f57
key_f58	kf58	Fm	KEY_F(58), 0502, Sent by function key f58
key_f59	kf59	Fn	KEY_F(59), 0503, Sent by function key f59
key_f60	kf60	Fo	KEY_F(60), 0504, Sent by function key f60
key_f61	kf61	Fp	KEY_F(61), 0505, Sent by function key f61
key_f62	kf62	Fq	KEY_F(62), 0506, Sent by function key f62
key_f63	kf63	Fr	KEY_F(63), 0507, Sent by function key f63
key_find	kfnd	@0	KEY_FIND, 0552, Sent by find key
key_help	khlp	%1	KEY_HELP, 0553, Sent by help key
key_home	khome	kh	KEY_HOME, 0406, Sent by home key
key_ic	kich1	kI	KEY_IC, 0513, Sent by ins-char/enter ins-mode key
key_il	kil1	kA	KEY_IL, 0511, Sent by insert-line key
key_left	kcub1	kl	KEY_LEFT, 0404, Sent by terminal left-arrow key
key_ll	kll	kH	KEY_LL, 0533, Sent by home-down key
key_mark	kmrk	%2	KEY_MARK, 0554, Sent by mark key
key_message	kmsg	%3	KEY_MESSAGE, 0555, Sent by message key

key_move	kmov	%4	KEY_MOVE, 0556, Sent by move key
key_next	knxt	%5	KEY_NEXT, 0557, Sent by next-object key
key_npage	knp	kN	KEY_NPAGE, 0522, Sent by next-page key
key_open	kopn	%6	KEY_OPEN, 0560, Sent by open key
key_options	kopt	%7	KEY_OPTIONS, 0561, Sent by options key
key_ppage	kpp	kP	KEY_PPAGE, 0523, Sent by previous-page key
key_previous	kprv	%8	KEY_PREVIOUS, 0562, Sent by previous-object key
key_print	kprt	%9	KEY_PRINT, 0532, Sent by print or copy key
key_redo	krdo	%0	KEY_REDO, 0563, Sent by redo key
key_reference	kref	&1	KEY_REFERENCE, 0564, Sent by ref(erence) key
key_refresh	krfr	&2	KEY_REFRESH, 0565, Sent by refresh key
key_replace	krpl	&3	KEY_REPLACE, 0566, Sent by replace key
key_restart	krst	&4	KEY_RESTART, 0567, Sent by restart key
key_resume	kres	&5	KEY_RESUME, 0570, Sent by resume key
key_right	kcuf1	kr	KEY_RIGHT, 0405, Sent by terminal right-arrow key
key_save	ksav	&6	KEY_SAVE, 0571, Sent by save key
key_sbeg	kBEG	&9	KEY_SBEG, 0572, Sent by shifted beginning key
key_scancel	kCAN	&0	KEY_SCANCEL, 0573, Sent by shifted cancel key
key_scommand	kCMD	*1	KEY_SCOMMAND, 0574, Sent by shifted command key
key_scopy	kCPY	*2	KEY_SCOPY, 0575, Sent by shifted copy key
key_screate	kCRT	*3	KEY_SCREATE, 0576, Sent by shifted create key
key_sdc	kDC	*4	KEY_SDC, 0577, Sent by shifted delete-char key
key_sdl	kDL	*5	KEY_SDL, 0600, Sent by shifted delete-line key
key_select	kslt	*6	KEY_SELECT, 0601, Sent by select key
key_send	kEND	*7	KEY_SEND, 0602, Sent by shifted end key
key_seol	kEOL	*8	KEY_SEOL, 0603, Sent by shifted clear-line key
key_sexit	kEXT	*9	KEY_SEXIT, 0604, Sent by shifted exit key
key_sf	kind	kF	KEY_SF, 0520, Sent by scroll-forward/down key
key_sfind	kFND	*0	KEY_SFIND, 0605, Sent by shifted find key
key_shelp	kHLP	#1	KEY_SHELP, 0606, Sent by shifted help key
key_shome	kHOM	#2	KEY_SHOME, 0607, Sent by shifted home key
key_sic	kIC	#3	KEY_SIC, 0610, Sent by shifted input key
key_sleft	kLFT	#4	KEY_SLEFT, 0611, Sent by shifted left-arrow key
key_smessage	kMSG	%a	KEY_SMESSAGE, 0612, Sent by shifted message key
key_smove	kMOV	%b	KEY_SMOVE, 0613, Sent by shifted move key
key_snext	kNXT	%c	KEY_SNEXT, 0614, Sent by shifted next key
key_soptions	kOPT	%d	KEY_SOPTIONS, 0615, Sent by shifted options key
key_sprevious	kPRV	%e	KEY_SPREVIOUS, 0616, Sent by shifted prev key
key_sprint	kPRT	%f	KEY_SPRINT, 0617, Sent by shifted print key
key_sr	kri	kR	KEY_SR, 0521, Sent by scroll-backward/up key
key_sredo	kRDO	%g	KEY_SREDO, 0620, Sent by shifted redo key
key_sreplace	kRPL	%h	KEY_SREPLACE, 0621, Sent by shifted replace key
key_sright	kRIT	%i	KEY_SRIGHT, 0622, Sent by shifted right-arrow key
key_srsume	kRES	%j	KEY_SRSUME, 0623, Sent by shifted resume key
key_ssave	kSAV	!1	KEY_SSAVE, 0624, Sent by shifted save key
key_ssuspend	kSPD	!2	KEY_SSUSPEND, 0625, Sent by shifted suspend key
key_stab	khts	kT	KEY_STAB, 0524, Sent by set-tab key
key_sundo	kUND	!3	KEY_SUNDO, 0626, Sent by shifted undo key
key_suspend	kspd	&7	KEY_SUSPEND, 0627, Sent by suspend key
key_undo	kund	&8	KEY_UNDO, 0630, Sent by undo key
key_up	kcuu1	ku	KEY_UP, 0403, Sent by terminal up-arrow key
keypad_local	rmkx	ke	Out of "keypad-transmit" mode

keypad_xmit	smkx	ks	Put terminal in "keypad-transmit" mode
lab_f0	lf0	l0	Labels on function key f0 if not f0
lab_f1	lf1	l1	Labels on function key f1 if not f1
lab_f2	lf2	l2	Labels on function key f2 if not f2
lab_f3	lf3	l3	Labels on function key f3 if not f3
lab_f4	lf4	l4	Labels on function key f4 if not f4
lab_f5	lf5	l5	Labels on function key f5 if not f5
lab_f6	lf6	l6	Labels on function key f6 if not f6
lab_f7	lf7	l7	Labels on function key f7 if not f7
lab_f8	lf8	l8	Labels on function key f8 if not f8
lab_f9	lf9	l9	Labels on function key f9 if not f9
lab_f10	lf10	la	Labels on function key f10 if not f10
label_off	rmln	LF	Turn off soft labels
label_on	smln	LO	Turn on soft labels
meta_off	rmm	mo	Turn off "meta mode"
meta_on	smm	mm	Turn on "meta mode" (8th bit)
newline	nel	nw	Newline (behaves like **cr** followed by **lf**)
pad_char	pad	pc	Pad character (rather than null)
parm_dch	dch	DC	Delete #1 chars (G*)
parm_delete_line	dl	DL	Delete #1 lines (G*)
parm_down_cursor	cud	DO	Move cursor down #1 lines. (G*)
parm_ich	ich	IC	Insert #1 blank chars (G*)
parm_index	indn	SF	Scroll forward #1 lines. (G)
parm_insert_line	il	AL	Add #1 new blank lines (G*)
parm_left_cursor	cub	LE	Move cursor left #1 spaces (G)
parm_right_cursor	cuf	RI	Move cursor right #1 spaces. (G*)
parm_rindex	rin	SR	Scroll backward #1 lines. (G)
parm_up_cursor	cuu	UP	Move cursor up #1 lines. (G*)
pkey_key	pfkey	pk	Prog funct key #1 to type string #2
pkey_local	pfloc	pl	Prog funct key #1 to execute string #2
pkey_xmit	pfx	px	Prog funct key #1 to xmit string #2
plab_norm	pln	pn	Prog label #1 to show string #2
print_screen	mc0	ps	Print contents of the screen
prtr_non	mc5p	pO	Turn on the printer for #1 bytes
prtr_off	mc4	pf	Turn off the printer
prtr_on	mc5	po	Turn on the printer
repeat_char	rep	rp	Repeat char #1 #2 times (G*)
req_for_input	rfi	RF	Send next input char (for ptys)
reset_1string	rs1	r1	Reset terminal completely to sane modes
reset_2string	rs2	r2	Reset terminal completely to sane modes
reset_3string	rs3	r3	Reset terminal completely to sane modes
reset_file	rf	rf	Name of file containing reset string
restore_cursor	rc	rc	Restore cursor to position of last sc
row_address	vpa	cv	Vertical position absolute (G)
save_cursor	sc	sc	Save cursor position.
scroll_forward	ind	sf	Scroll text up
scroll_reverse	ri	sr	Scroll text down
set_attributes	sgr	sa	Define the video attributes #1-#9 (G)
set_left_margin	smgl	ML	Set soft left margin
set_right_margin	smgr	MR	Set soft right margin
set_tab	hts	st	Set a tab in all rows, current column.
set_window	wind	wi	Current window is lines #1-#2 cols #3-#4 (G)

tab	ht	ta	Tab to next 8 space hardware tab stop.
to_status_line	tsl	ts	Go to status line, col #1 (G)
underline_char	uc	uc	Underscore one char and move past it
up_half_line	hu	hu	Half-line up (reverse 1/2 linefeed)
xoff_character	xoffc	XF	X-off character
xon_character	xonc	XN	X-on character

SAMPLE ENTRY

The following entry, which describes the *Concept*–100 terminal, is among the more complex entries in the *terminfo* file as of this writing.

```
concept100|c100|concept|c104|c100-4p|concept 100,
    am, db, eo, in, mir, ul, xenl,
    cols#80, lines#24, pb#9600, vt#8,
    bel=^G, blank=\EH, blink=\EC, clear=^L$<2*>,
    cnorm=\Ew, cr=^M$<9>, cub1=^H, cud1=^J,
    cuf1=\E=, cup=\Ea%p1%'␣'%+%c%p2%'␣'%+%c,
    cuu1=\E;, cvvis=\EW, dch1=\E^A$<16*>, dim=\EE,
    dl1=\E^B$<3*>, ed=\E^C$<16*>, el=\E^U$<16>,
    flash=\Ek$<20>\EK, ht=\t$<8>, il1=\E^R$<3*>,
    ind=^J, .ind=^J$<9>, ip=$<16*>,
    is2=\EU\Ef\E7\E5\E8\El\ENH\EK\E\0\Eo&\0\Eo\47\E,
    kbs=^h, kcub1=\E>, kcud1=\E<, kcuf1=\E=, kcuu1=\E;,
    kf1=\E5, kf2=\E6, kf3=\E7, khome=\E?,
    prot=\EI, rep=\Er%p1%c%p2%'␣'%+%c$<.2*>,
    rev=\ED, rmcup=\Ev\s\s\s\s$<6>\Ep\r\n,
    rmir=\E\0, rmkx=\Ex, rmso=\Ed\Ee, rmul=\Eg,
    rmul=\Eg, sgr0=\EN\0, smcup=\EU\Ev\s\s8p\Ep\r,
    smir=\E^P, smkx=\EX, smso=\EE\ED, smul=\EG,
```

Entries may continue onto multiple lines by placing white space at the beginning of each line except the first. Lines beginning with "#" are taken as comment lines. Capabilities in *terminfo* are of three types: boolean capabilities which indicate that the terminal has some particular feature, numeric capabilities giving the size of the terminal or particular features, and string capabilities, which give a sequence which can be used to perform particular terminal operations.

Types of Capabilities

All capabilities have names. For instance, the fact that the *Concept* has *automatic margins* (i.e., an automatic return and linefeed when the end of a line is reached) is indicated by the capability **am**. Hence the description of the *Concept* includes **am**. Numeric capabilities are followed by the character '#' and then the value. Thus **cols**, which indicates the number of columns the terminal has, gives the value **80** for the *Concept*. The value may be specified in decimal, octal or hexadecimal using normal C conventions.

Finally, string-valued capabilities, such as **el** (clear to end of line sequence) are given by the two- to five-character capname, an '=', and then a string ending at the next following comma. A delay in milliseconds may appear anywhere in such a capability, enclosed in **$<..>** brackets, as in **el=\EK$<3>**, and padding characters are supplied by **tputs**( ) (see *curses*(3X)) to provide this delay. The delay can be either a number, e.g., **20**, or a number followed by an '*' (i.e., **3***), a '**/**' (i.e., **5/**), or both (i.e., **10*/**). A '*' indicates that the padding required is proportional to the number of lines affected by the operation, and the amount given is the per-affected-unit padding required. (In the case of insert character, the factor is still the number of lines affected. This is always one unless the terminal has **in** and the software uses it.) When a '*' is specified, it is sometimes useful to give a delay of the form **3.5** to specify a delay per unit to tenths of milliseconds. (Only one decimal place is allowed.) A '/' indicates that the

padding is mandatory. Otherwise, if the terminal has **xon** defined, the padding information is advisory and will only be used for cost estimates or when the terminal is in raw mode. Mandatory padding will be transmitted regardless of the setting of **xon**.

A number of escape sequences are provided in the string valued capabilities for easy encoding of characters there. Both **\E** and **\e** map to an ESCAPE character, ^*x* maps to a control–*x* for any appropriate *x*, and the sequences **\n, \l, \r, \t, \b, \f,** and \s give a newline, linefeed, return, tab, backspace, formfeed, and space, respectively. Other escapes include: \^ for caret (^); \\ for backslash (\); \, for comma (,); \: for colon (:); and **\0** for null. (**\0** will actually produce **\200**, which does not terminate a string but behaves as a null character on most terminals.) Finally, characters may be given as three octal digits after a backslash (e.g., \123).

Sometimes individual capabilities must be commented out. To do this, put a period before the capability name. For example, see the second **ind** in the example above. Note that capabilities are defined in a left-to-right order and, therefore, a prior definition will override a later definition.

Preparing Descriptions

The most effective way to prepare a terminal description is by imitating the description of a similar terminal in *terminfo* and to build up a description gradually, using partial descriptions with **vi**(1) to check that they are correct. Be aware that a very unusual terminal may expose deficiencies in the ability of the **terminfo** file to describe it or the inability of **vi**(1) to work with that terminal. To test a new terminal description, set the environment variable **TERMINFO** to a pathname of a directory containing the compiled description you are working on and programs will look there rather than in */usr/lib/terminfo*. To get the padding for insert-line correct (if the terminal manufacturer did not document it) a severe test is to comment out **xon**, edit a large file at 9600 baud with **vi**(1), delete 16 or so lines from the middle of the screen, then hit the **u** key several times quickly. If the display is corrupted, more padding is usually needed. A similar test can be used for insert-character.

Basic Capabilities

The number of columns on each line for the terminal is given by the **cols** numeric capability. If the terminal has a screen, then the number of lines on the screen is given by the **lines** capability. If the terminal wraps around to the beginning of the next line when it reaches the right margin, then it should have the **am** capability. If the terminal can clear its screen, leaving the cursor in the home position, then this is given by the **clear** string capability. If the terminal overstrikes (rather than clearing a position when a character is struck over) then it should have the **os** capability. If the terminal is a printing terminal, with no soft copy unit, give it both **hc** and **os**. (**os** applies to storage scope terminals, such as Tektronix 4010 series, as well as hard-copy and APL terminals.) If there is a code to move the cursor to the left edge of the current row, give this as **cr**. (Normally this will be carriage return, control M.) If there is a code to produce an audible signal (bell, beep, etc) give this as **bel**. If the terminal uses the xon-xoff flow-control protocol, like most terminals, specify **xon**.

If there is a code to move the cursor one position to the left (such as backspace) that capability should be given as **cub1**. Similarly, codes to move to the right, up, and down should be given as **cuf1**, **cuu1**, and **cud1**. These local cursor motions should not alter the text they pass over; for example, you would not normally use "**cuf1**=\s" because the space would erase the character moved over.

A very important point here is that the local cursor motions encoded in *terminfo* are undefined at the left and top edges of a screen terminal. Programs should never attempt to backspace around the left edge, unless **bw** is given, and should never attempt to go up locally off the top. In order to scroll text up, a program will go to the bottom left corner of the screen and send the **ind** (index) string.

To scroll text down, a program goes to the top left corner of the screen and sends the **ri** (reverse index) string. The strings **ind** and **ri** are undefined when not on their respective corners of the screen.

Parameterized versions of the scrolling sequences are **indn** and **rin** which have the same semantics as **ind** and **ri** except that they take one parameter, and scroll that many lines. They are also undefined except at the appropriate edge of the screen.

The **am** capability tells whether the cursor sticks at the right edge of the screen when text is output, but this does not necessarily apply to a **cuf1** from the last column. The only local motion which is defined from the left edge is if **bw** is given, then a **cub1** from the left edge will move to the right edge of the previous row. If **bw** is not given, the effect is undefined. This is useful for drawing a box around the edge of the screen, for example. If the terminal has switch selectable automatic margins, the *terminfo* file usually assumes that this is on; i.e., **am**. If the terminal has a command which moves to the first column of the next line, that command can be given as **nel** (newline). It does not matter if the command clears the remainder of the current line, so if the terminal has no **cr** and **lf** it may still be possible to craft a working **nel** out of one or both of them.

These capabilities suffice to describe hardcopy and screen terminals. Thus the model 33 teletype is described as

33 |tty33 |tty |model 33 teletype, bel=^G, cols#72, cr=^M, cud1=^J, hc, ind=^J, os,

while the Lear Siegler ADM-3 is described as

adm3 |lsi adm3, am, bel=^G, clear=^Z, cols#80, cr=^M, cub1=^H, cud1=^J, ind=^J, lines#24,

Parameterized Strings

Cursor addressing and other strings requiring parameters in the terminal are described by a parameterized string capability, with **printf**(3S)-like escapes (**%x**) in it. For example, to address the cursor, the **cup** capability is given, using two parameters: the row and column to address to. (Rows and columns are numbered from zero and refer to the physical screen visible to the user, not to any unseen memory.) If the terminal has memory relative cursor addressing, that can be indicated by **mrcup**.

The parameter mechanism uses a stack and special **%** codes to manipulate it in the manner of a Reverse Polish Notation (postfix) calculator. Typically a sequence will push one of the parameters onto the stack and then print it in some format. Often more complex operations are necessary. Binary operations are in postfix form with the operands in the usual order. That is, to get x-5 one would use **%gx%{5}%-**.

The **%** encodings have the following meanings:

%%	outputs '%'
%[[:]*flags*][*width*[.*precision*]][**doxXs**]	as in printf, flags are [**-+#**] and space
%c	print pop() gives %c
%p[1-9]	push i^{th} parm
%P[a-z]	set variable [a-z] to pop()
%g[a-z]	get variable [a-z] and push it
%'*c*'	push char constant *c*
%{*nn*}	push decimal constant *nn*
%l	push strlen(pop())
%+ %- %* %/ %m	arithmetic (%m is mod): push(pop() op pop())

%& %| %^	bit operations: push(pop() op pop())
%= %> %<	logical operations: push(pop() op pop())
%A %O	logical operations: and, or
%! %~	unary operations: push(op pop())
%i	(for ANSI terminals) add 1 to first parm, if one parm present, or first two parms, if more than one parm present

%? expr %t thenpart %e elsepart %;
if-then-else, %e elsepart is optional;
else-if's are possible ala Algol 68:
%? c_1 %t b_1 %e c_2 %t b_2 %e c_3 %t b_3 %e c_4 %t b_4 %e b_5%;
c_i are conditions, b_i are bodies.

If the "–" flag is used with "%[doxXs]", then a colon (:) must be placed between the "%" and the "–" to differentiate the flag from the binary "%–" operator, .e.g "%:–16.16s".

Consider the Hewlett-Packard 2645, which, to get to row 3 and column 12, needs to be sent **\E&a12c03Y** padded for 6 milliseconds. Note that the order of the rows and columns is inverted here, and that the row and column are zero-padded as two digits. Thus its **cup** capability is "**cup**=\E&a%p2%2.2dc%p1%2.2dY$<6>".

The Micro-Term ACT-IV needs the current row and column sent preceded by a **^T**, with the row and column simply encoded in binary, "**cup**=^T%p1%c%p2%c". Terminals which use "%c" need to be able to backspace the cursor (**cub1**), and to move the cursor up one line on the screen (**cuu1**). This is necessary because it is not always safe to transmit **\n**, **^D**, and **\r**, as the system may change or discard them. (The library routines dealing with *terminfo* set tty modes so that tabs are never expanded, so **\t** is safe to send. This turns out to be essential for the Ann Arbor 4080.)

A final example is the LSI ADM-3a, which uses row and column offset by a blank character, thus "**cup**=\E=%p1%'\s'%+%c%p2%'\s'%+%c". After sending "\E=", this pushes the first parameter, pushes the ASCII value for a space (32), adds them (pushing the sum on the stack in place of the two previous values), and outputs that value as a character. Then the same is done for the second parameter. More complex arithmetic is possible using the stack.

Cursor Motions

If the terminal has a fast way to home the cursor (to very upper left corner of screen) then this can be given as **home**; similarly a fast way of getting to the lower left-hand corner can be given as **ll**; this may involve going up with **cuu1** from the home position, but a program should never do this itself (unless **ll** does) because it can make no assumption about the effect of moving up from the home position. Note that the home position is the same as addressing to (0,0): to the top left corner of the screen, not of memory. (Thus, the **\EH** sequence on Hewlett-Packard terminals cannot be used for **home** without losing some of the other features on the terminal.)

If the terminal has row or column absolute-cursor addressing, these can be given as single parameter capabilities **hpa** (horizontal position absolute) and **vpa** (vertical position absolute). Sometimes these are shorter than the more general two-parameter sequence (as with the Hewlett-Packard 2645) and can be used in preference to **cup**. If there are parameterized local motions (e.g., move *n* spaces to the right) these can be given as **cud**, **cub**, **cuf**, and **cuu** with a single parameter indicating how many spaces to move. These are primarily useful if the terminal does not have **cup**, such as the Tektronix 4025.

Area Clears

If the terminal can clear from the current position to the end of the line, leaving the cursor where it is, this should be given as **el**. If the terminal can clear from the beginning of the line

to the current position inclusive, leaving the cursor where it is, this should be given as **el1**. If the terminal can clear from the current position to the end of the display, then this should be given as **ed**. **ed** is only defined from the first column of a line. (Thus, it can be simulated by a request to delete a large number of lines, if a true **ed** is not available.)

Insert/delete line

If the terminal can open a new blank line before the line where the cursor is, this should be given as **il1**; this is done only from the first position of a line. The cursor must then appear on the newly blank line. If the terminal can delete the line which the cursor is on, then this should be given as **dl1**; this is done only from the first position on the line to be deleted. Versions of **il1** and **dl1** which take a single parameter and insert or delete that many lines can be given as **il** and **dl**. If the terminal has a settable destructive scrolling region (like the VT100) the command to set this can be described with the **csr** capability, which takes two parameters: the top and bottom lines of the scrolling region. The cursor position is, alas, undefined after using this command. It is possible to get the effect of insert or delete line using this command – the **sc** and **rc** (save and restore cursor) commands are also useful. Inserting lines at the top or bottom of the screen can also be done using **ri** or **ind** on many terminals without a true insert/delete line, and is often faster even on terminals with those features. To determine whether a terminal has destructive scrolling regions or non-destructive scrolling regions, create a scrolling region in the middle of the screen, place data on the bottom line of the scrolling region, move the cursor to the top line of the scrolling region, and do a reverse index (**ri**) followed by a delete line (**dl1**) or index (**ind**). If the data that was originally on the bottom line of the scrolling region was restored into the scrolling region by the **dl1** or **ind**, then the terminal has non-destructive scrolling regions. Otherwise, it has destructive scrolling regions. Do not specify **csr** if the terminal has non-destructive scrolling regions, unless **ind**, **ri**, **indn**, **rin**, **dl**, and **dl1** all simulate destructive scrolling.

If the terminal has the ability to define a window as part of memory, which all commands affect, it should be given as the parameterized string **wind**. The four parameters are the starting and ending lines in memory and the starting and ending columns in memory, in that order.

If the terminal can retain display memory above, then the **da** capability should be given; if display memory can be retained below, then **db** should be given. These indicate that deleting a line or scrolling a full screen may bring non-blank lines up from below or that scrolling back with **ri** may bring down non-blank lines.

Insert/Delete Character

There are two basic kinds of intelligent terminals with respect to insert/delete character operations which can be described using *terminfo*. The most common insert/delete character operations affect only the characters on the current line and shift characters off the end of the line rigidly. Other terminals, such as the *Concept* 100 and the Perkin Elmer Owl, make a distinction between typed and untyped blanks on the screen, shifting upon an insert or delete only to an untyped blank on the screen which is either eliminated, or expanded to two untyped blanks. You can determine the kind of terminal you have by clearing the screen and then typing text separated by cursor motions. Type "**abc def**" using local cursor motions (not spaces) between the **abc** and the **def**. Then position the cursor before the **abc** and put the terminal in insert mode. If typing characters causes the rest of the line to shift rigidly and characters to fall off the end, then your terminal does not distinguish between blanks and untyped positions. If the **abc** shifts over to the **def** which then move together around the end of the current line and onto the next as you insert, you have the second type of terminal, and should give the capability **in**, which stands for "insert null". While these are two logically separate attributes (one line versus multiline insert mode, and special treatment of untyped spaces) we have seen no terminals whose insert mode cannot be described with the single attribute.

terminfo can describe both terminals which have an insert mode and terminals which send a simple sequence to open a blank position on the current line. Give as **smir** the sequence to get into insert mode. Give as **rmir** the sequence to leave insert mode. Now give as **ich1** any sequence needed to be sent just before sending the character to be inserted. Most terminals with a true insert mode will not give **ich1**; terminals which send a sequence to open a screen position should give it here. (If your terminal has both, insert mode is usually preferable to **ich1**. Do not give both unless the terminal actually requires both to be used in combination.) If post-insert padding is needed, give this as a number of milliseconds padding in **ip** (a string option). Any other sequence which may need to be sent after an insert of a single character may also be given in **ip**. If your terminal needs both to be placed into an 'insert mode' and a special code to precede each inserted character, then both **smir/rmir** and **ich1** can be given, and both will be used. The **ich** capability, with one parameter, *n*, will repeat the effects of **ich1** *n* times.

If padding is necessary between characters typed while not in insert mode, give this as a number of milliseconds padding in **rmp**.

It is occasionally necessary to move around while in insert mode to delete characters on the same line (e.g., if there is a tab after the insertion position). If your terminal allows motion while in insert mode you can give the capability **mir** to speed up inserting in this case. Omitting **mir** will affect only speed. Some terminals (notably Datamedia's) must not have **mir** because of the way their insert mode works.

Finally, you can specify **dch1** to delete a single character, **dch** with one parameter, *n*, to delete *n* characters, and delete mode by giving **smdc** and **rmdc** to enter and exit delete mode (any mode the terminal needs to be placed in for **dch1** to work).

A command to erase *n* characters (equivalent to outputting *n* blanks without moving the cursor) can be given as **ech** with one parameter.

Highlighting, Underlining, and Visible Bells

If your terminal has one or more kinds of display attributes, these can be represented in a number of different ways. You should choose one display form as *standout mode* (see *curses*(3X)), representing a good, high contrast, easy-on-the-eyes, format for highlighting error messages and other attention getters. (If you have a choice, reverse-video plus half-bright is good, or reverse-video alone; however, different users have different preferences on different terminals.) The sequences to enter and exit standout mode are given as **smso** and **rmso**, respectively. If the code to change into or out of standout mode leaves one or even two blank spaces on the screen, as the TVI 912 and Teleray 1061 do, then **xmc** should be given to tell how many spaces are left.

Codes to begin underlining and end underlining can be given as **smul** and **rmul** respectively. If the terminal has a code to underline the current character and move the cursor one space to the right, such as the Micro-Term MIME, this can be given as **uc**.

Other capabilities to enter various highlighting modes include **blink** (blinking), **bold** (bold or extra-bright), **dim** (dim or half-bright), **invis** (blanking or invisible text), **prot** (protected), **rev** (reverse-video), **sgr0** (turn off all attribute modes), **smacs** (enter alternate-character-set mode), and **rmacs** (exit alternate-character-set mode). Turning on any of these modes singly may or may not turn off other modes. If a command is necessary before alternate character set mode is entered, give the sequence in **enacs** (enable alternate-character-set mode).

If there is a sequence to set arbitrary combinations of modes, this should be given as **sgr** (set attributes), taking nine parameters. Each parameter is either **0** or non-zero, as the corresponding attribute is on or off. The nine parameters are, in order: standout, underline, reverse, blink, dim, bold, blank, protect, alternate character set. Not all modes need be supported by **sgr**, only those for which corresponding separate attribute commands exist. (See

the example at the end of this section.)

Terminals with the "magic cookie" glitch (**xmc**) deposit special "cookies" when they receive mode-setting sequences, which affect the display algorithm rather than having extra bits for each character. Some terminals, such as the Hewlett-Packard 2621, automatically leave standout mode when they move to a new line or the cursor is addressed. Programs using standout mode should exit standout mode before moving the cursor or sending a newline, unless the **msgr** capability, asserting that it is safe to move in standout mode, is present.

If the terminal has a way of flashing the screen to indicate an error quietly (a bell replacement), then this can be given as **flash**; it must not move the cursor. A good flash can be done by changing the screen into reverse video, pad for 200 ms, then return the screen to normal video.

If the cursor needs to be made more visible than normal when it is not on the bottom line (to make, for example, a non-blinking underline into an easier to find block or blinking underline) give this sequence as **cvvis**. The boolean **chts** should also be given. If there is a way to make the cursor completely invisible, give that as **civis**. The capability **cnorm** should be given which undoes the effects of either of these modes.

If the terminal needs to be in a special mode when running a program that uses these capabilities, the codes to enter and exit this mode can be given as **smcup** and **rmcup**. This arises, for example, from terminals like the *Concept* with more than one page of memory. If the terminal has only memory relative cursor addressing and not screen relative cursor addressing, a one screen-sized window must be fixed into the terminal for cursor addressing to work properly. This is also used for the Tektronix 4025, where **smcup** sets the command character to be the one used by **terminfo**. If the **smcup** sequence will not restore the screen after an **rmcup** sequence is output (to the state prior to outputting **rmcup**), specify **nrrmc**.

If your terminal generates underlined characters by using the underline character (with no special codes needed) even though it does not otherwise overstrike characters, then you should give the capability **ul**. For terminals where a character overstriking another leaves both characters on the screen, give the capability **os**. If overstrikes are erasable with a blank, then this should be indicated by giving **eo**.

Example of highlighting: assume that the terminal under question needs the following escape sequences to turn on various modes.

tparm parameter	attribute	escape sequence
	none	\E[0m
p1	standout	\E[0;4;7m
p2	underline	\E[0;3m
p3	reverse	\E[0;4m
p4	blink	\E[0;5m
p5	dim	\E[0;7m
p6	bold	\E[0;3;4m
p7	invis	\E[0;8m
p8	protect	not available
p9	altcharset	^O (off) ^N(on)

Note that each escape sequence requires a **0** to turn off other modes before turning on its own mode. Also note that, as suggested above, *standout* is set up to be the combination of *reverse* and *dim*. Also, since this terminal has no *bold* mode, *bold* is set up as the combination of *reverse* and *underline*. In addition, to allow combinations, such as *underline+blink*, the sequence to use would be **\E[0;3;5m**. The terminal doesn't have *protect* mode, either, but

that cannot be simulated in any way, so **p8** is ignored. The *altcharset* mode is different in that it is either ^**O** or ^**N** depending on whether it is off or on. If all modes were to be turned on, the sequence would be **\E[0;3;4;5;7;8m^N**.

Now look at when different sequences are output. For example, **;3** is output when either **p2** or **p6** is true, that is, if either *underline* or *bold* modes are turned on. Writing out the above sequences, along with their dependencies, gives the following:

sequence	when to output	terminfo translation
\E[0	always	\E[0
;3	if p2 or p6	%?%p2%p6%|%t;3%;
;4	if p1 or p3 or p6	%?%p1%p3%|%p6%|%t;4%;
;5	if p4	%?%p4%t;5%;
;7	if p1 or p5	%?%p1%p5%|%t;7%;
;8	if p7	%?%p7%t;8%;
m	always	m
^N or ^O	if p9 ^N, else ^O	%?%p9%t^N%e^O%;

Putting this all together into the **sgr** sequence gives:

sgr=\E[0%?%p2%p6%|%t;3%;%?%p1%p3%|%p6%|%t;4%;%?%p5%t;5%;%?%p1%p5%
|%t;7%;%?%p7%t;8%;m%?%p9%t^N%e^O%;,

Keypad

If the terminal has a keypad that transmits codes when the keys are pressed, this information can be given. Note that it is not possible to handle terminals where the keypad only works in local (this applies, for example, to the unshifted Hewlett-Packard 2621 keys). If the keypad can be set to transmit or not transmit, give these codes as **smkx** and **rmkx**. Otherwise the keypad is assumed to always transmit.

The codes sent by the left arrow, right arrow, up arrow, down arrow, and home keys can be given as **kcub1, kcuf1, kcuu1, kcud1,** and **khome** respectively. If there are function keys such as f0, f1, ..., f63, the codes they send can be given as **kf0, kf1, ..., kf63**. If the first 11 keys have labels other than the default f0 through f10, the labels can be given as **lf0, lf1, ..., lf10**. The codes transmitted by certain other special keys can be given: **kll** (home down), **kbs** (backspace), **ktbc** (clear all tabs), **kctab** (clear the tab stop in this column), **kclr** (clear screen or erase key), **kdch1** (delete character), **kdl1** (delete line), **krmir** (exit insert mode), **kel** (clear to end of line), **ked** (clear to end of screen), **kich1** (insert character or enter insert mode), **kil1** (insert line), **knp** (next page), **kpp** (previous page), **kind** (scroll forward/down), **kri** (scroll backward/up), **khts** (set a tab stop in this column). In addition, if the keypad has a 3 by 3 array of keys including the four arrow keys, the other five keys can be given as **ka1, ka3, kb2, kc1,** and **kc3**. These keys are useful when the effects of a 3 by 3 directional pad are needed. Further keys are defined above in the capabilities list.

Strings to program function keys can be given as **pfkey, pfloc,** and **pfx**. A string to program their soft-screen labels can be given as **pln**. Each of these strings takes two parameters: the function key number to program (from 0 to 10) and the string to program it with. Function key numbers out of this range may program undefined keys in a terminal-dependent manner. The difference between the capabilities is that **pfkey** causes pressing the given key to be the same as the user typing the given string; **pfloc** causes the string to be executed by the terminal in local mode; and **pfx** causes the string to be transmitted to the computer. The capabilities **nlab, lw** and **lh** define how many soft labels there are and their width and height. If there are commands to turn the labels on and off, give them in **smln** and **rmln**. **smln** is normally output after one or more **pln** sequences to make sure that the change becomes visible.

Tabs and Initialization

If the terminal has hardware tabs, the command to advance to the next tab stop can be given as **ht** (usually control I). A "backtab" command which moves leftward to the next tab stop can be given as **cbt**. By convention, if the teletype modes indicate that tabs are being expanded by the computer rather than being sent to the terminal, programs should not use **ht** or **cbt** even if they are present, since the user may not have the tab stops properly set. If the terminal has hardware tabs which are initially set every *n* spaces when the terminal is powered up, the numeric parameter **it** is given, showing the number of spaces the tabs are set to. This is normally used by **tput init** (see *tput*(1)) to determine whether to set the mode for hardware tab expansion and whether to set the tab stops. If the terminal has tab stops that can be saved in nonvolatile memory, the *terminfo* description can assume that they are properly set. If there are commands to set and clear tab stops, they can be given as **tbc** (clear all tab stops) and **hts** (set a tab stop in the current column of every row).

Other capabilities include: **is1**, **is2**, and **is3**, initialization strings for the terminal; **iprog**, the path name of a program to be run to initialize the terminal; and **if**, the name of a file containing long initialization strings. These strings are expected to set the terminal into modes consistent with the rest of the *terminfo* description. They must be sent to the terminal each time the user logs in and be output in the following order: run the program **iprog**; output **is1**; output **is2**; set the margins using **mgc**, **smgl** and **smgr**; set the tabs using **tbc** and **hts**; print the file **if**; and finally output **is3**. This is usually done using the **init** option of *tput*(1); see *profile*(4). Most initialization is done with **is2**. Special terminal modes can be set up without duplicating strings by putting the common sequences in **is2** and special cases in **is1** and **is3**. Sequences that do a harder reset from a totally unknown state can be given as **rs1**, **rs2**, **rf**, and **rs3**, analogous to **is1**, **is2**, **is3**, and **if**. (The method using files, **if** and **rf**, is used for a few terminals, from */usr/lib/tabset/**; however, the recommended method is to use the initialization and reset strings.) These strings are output by **tput reset**, which is used when the terminal gets into a wedged state. Commands are normally placed in **rs1**, **rs2**, **rs3**, and **rf** only if they produce annoying effects on the screen and are not necessary when logging in. For example, the command to set a terminal into 80-column mode would normally be part of **is2**, but on some terminals it causes an annoying glitch on the screen and is not normally needed since the terminal is usually already in 80-column mode.

If a more complex sequence is needed to set the tabs than can be described by using **tbc** and **hts**, the sequence can be placed in **is2** or **if**.

If there are commands to set and clear margins, they can be given as **mgc** (clear all margins), **smgl** (set left margin), and **smgr** (set right margin).

Delays

Certain capabilities control padding in the *tty*(7) driver. These are primarily needed by hard-copy terminals, and are used by **tput init** to set tty modes appropriately. Delays embedded in the capabilities **cr**, **ind**, **cub1**, **ff**, and **tab** can be used to set the appropriate delay bits to be set in the tty driver. If **pb** (padding baud rate) is given, these values can be ignored at baud rates below the value of **pb**.

Status Lines

If the terminal has an extra "status line" that is not normally used by software, this fact can be indicated. If the status line is viewed as an extra line below the bottom line, into which one can cursor address normally (such as the Heathkit h19's 25th line, or the 24th line of a VT100 which is set to a 23-line scrolling region), the capability **hs** should be given. Special strings that go to a given column of the status line and return from the status line can be given as **tsl** and **fsl**. (**fsl** must leave the cursor position in the same place it was before **tsl**. If necessary, the **sc** and **rc** strings can be included in **tsl** and **fsl** to get this effect.) The capability **tsl** takes one parameter, which is the column number of the status line the cursor is to be moved to.

If escape sequences and other special commands, such as tab, work while in the status line, the flag **eslok** can be given. A string which turns off the status line (or otherwise erases its contents) should be given as **dsl**. If the terminal has commands to save and restore the position of the cursor, give them as **sc** and **rc**. The status line is normally assumed to be the same width as the rest of the screen, e.g., **cols**. If the status line is a different width (possibly because the terminal does not allow an entire line to be loaded) the width, in columns, can be indicated with the numeric parameter **wsl**.

Line Graphics

If the terminal has a line drawing alternate character set, the mapping of glyph to character would be given in **acsc**. The definition of this string is based on the alternate character set used in the DEC VT100 terminal, extended slightly with some characters from the AT&T 4410v1 terminal.

glyph name	vt100+ character
arrow pointing right	+
arrow pointing left	,
arrow pointing down	.
solid square block	0
lantern symbol	I
arrow pointing up	-
diamond	`
checker board (stipple)	a
degree symbol	f
plus/minus	g
board of squares	h
lower right corner	j
upper right corner	k
upper left corner	l
lower left corner	m
plus	n
scan line 1	o
horizontal line	q
scan line 9	s
left tee (├)	t
right tee (┤)	u
bottom tee (┴)	v
top tee (┬)	w
vertical line	x
bullet	~

The best way to describe a new terminal's line graphics set is to add a third column to the above table with the characters for the new terminal that produce the appropriate glyph when the terminal is in the alternate character set mode. For example,

glyph name	vt100+ char	new tty char
upper left corner	l	R
lower left corner	m	F
upper right corner	k	T
lower right corner	j	G
horizontal line	q	,
vertical line	x	.

Now write down the characters left to right, as in "**acsc**=lRmFkTjGq\,x.".

Miscellaneous

If the terminal requires other than a null (zero) character as a pad, then this can be given as **pad**. Only the first character of the **pad** string is used. If the terminal does not have a pad character, specify **npc**.

If the terminal can move up or down half a line, this can be indicated with **hu** (half-line up) and **hd** (half-line down). This is primarily useful for superscripts and subscripts on hardcopy terminals. If a hardcopy terminal can eject to the next page (form feed), give this as **ff** (usually control L).

If there is a command to repeat a given character a given number of times (to save time transmitting a large number of identical characters) this can be indicated with the parameterized string **rep**. The first parameter is the character to be repeated and the second is the number of times to repeat it. Thus, **tparm(repeat_char, 'x', 10)** is the same as **xxxxxxxxxx.**

If the terminal has a settable command character, such as the Tektronix 4025, this can be indicated with **cmdch**. A prototype command character is chosen which is used in all capabilities. This character is given in the **cmdch** capability to identify it. The following convention is supported on some UNIX systems: If the environment variable **CC** exists, all occurrences of the prototype character are replaced with the character in **CC**.

Terminal descriptions that do not represent a specific kind of known terminal, such as **switch**, **dialup**, **patch**, and **network**, should include the **gn** (generic) capability so that programs can complain that they do not know how to talk to the terminal. (This capability does not apply to **virtual** terminal descriptions for which the escape sequences are known.) If the terminal is one of those supported by the UNIX system virtual terminal protocol, the terminal number can be given as **vt**. A line-turn-around sequence to be transmitted before doing reads should be specified in **rfi**.

If the terminal uses xon/xoff handshaking for flow control, give **xon**. Padding information should still be included so that routines can make better decisions about costs, but actual pad characters will not be transmitted. Sequences to turn on and off xon/xoff handshaking may be given in **smxon** and **rmxon**. If the characters used for handshaking are not ^S and ^Q, they may be specified with **xonc** and **xoffc**.

If the terminal has a "meta key" which acts as a shift key, setting the 8th bit of any character transmitted, this fact can be indicated with **km**. Otherwise, software will assume that the 8th bit is parity and it will usually be cleared. If strings exist to turn this "meta mode" on and off, they can be given as **smm** and **rmm**.

If the terminal has more lines of memory than will fit on the screen at once, the number of lines of memory can be indicated with **lm**. A value of **lm**#0 indicates that the number of lines is not fixed, but that there is still more memory than fits on the screen.

Media copy strings which control an auxiliary printer connected to the terminal can be given as **mc0**: print the contents of the screen, **mc4**: turn off the printer, and **mc5**: turn on the printer. When the printer is on, all text sent to the terminal will be sent to the printer. A

variation, **mc5p**, takes one parameter, and leaves the printer on for as many characters as the value of the parameter, then turns the printer off. The parameter should not exceed 255. If the text is not displayed on the terminal screen when the printer is on, specify **mc5i** (silent printer). All text, including **mc4**, is transparently passed to the printer while an **mc5p** is in effect.

Special Cases

The working model used by *terminfo* fits most terminals reasonably well. However, some terminals do not completely match that model, requiring special support by *terminfo*. These are not meant to be construed as deficiencies in the terminals; they are just differences between the working model and the actual hardware. They may be unusual devices or, for some reason, do not have all the features of the *terminfo* model implemented. Terminals which can not display tilde (~) characters, such as certain Hazeltine terminals, should indicate **hz**. Terminals which ignore a linefeed immediately after an **am** wrap, such as the *Concept* 100, should indicate **xenl**. Those terminals whose cursor remains on the right-most column until another character has been received, rather than wrapping immediately upon receiving the right-most character, such as the VT100, should also indicate **xenl**. If **el** is required to get rid of standout (instead of writing normal text on top of it), **xhp** should be given. Those Teleray terminals whose tabs turn all characters moved over to blanks, should indicate **xt** (destructive tabs). This capability is also taken to mean that it is not possible to position the cursor on top of a "magic cookie" therefore, to erase standout mode, it is instead necessary to use delete and insert line. Those Beehive Superbee terminals which do not transmit the escape or control–C characters, should specify **xsb**, indicating that the f1 key is to be used for escape and the f2 key for control–C.

Similar Terminals

If there are two very similar terminals, one can be defined as being just like the other with certain exceptions. The string capability **use** can be given with the name of the similar terminal. The capabilities given before **use** override those in the terminal type invoked by **use**. A capability can be canceled by placing *xx*@ to the left of the capability definition, where *xx* is the capability. For example, the entry

```
att4424-2|Teletype 4424 in display function group ii,
        rev@, sgr@, smul@, use=att4424,
```

defines an AT&T 4424 terminal that does not have the **rev**, **sgr**, and **smul** capabilities, and hence cannot do highlighting. This is useful for different modes for a terminal, or for different user preferences. More than one **use** capability may be given.

FILES

/usr/lib/terminfo/?/*	compiled terminal description database
/usr/lib/.COREterm/?/*	subset of compiled terminal description database
/usr/lib/tabset/*	tab settings for some terminals, in a format appropriate to be output to the terminal (escape sequences that set margins and tabs)

SEE ALSO

curses(3X), printf(3S), term(5).
captoinfo(1M), infocmp(1M), tic(1M), tty(7) in the *System Administrator's Reference Manual* .
tput(1) in the *User's Reference Manual*.
Chapter 10 of the *Programmer's Guide*.

WARNING

As described in the "Tabs and Initialization" section above, a terminal's initialization strings, **is1**, **is2**, and **is3**, if defined, must be output before a *curses*(3X) program is run. An available mechanism for outputting such strings is **tput init** (see *profile*(4)).

Tampering with entries in */usr/lib/.COREterm/?/** or */usr/lib/terminfo/?/** (for example, changing or removing an entry) can affect programs such as *vi*(1) that expect the entry to be present and correct. In particular, removing the description for the "dumb" terminal will cause unexpected problems.

NOTE

The *termcap* database (from earlier releases of UNIX System V) may not be supplied in future releases.

NAME

timezone – set default system time zone

SYNOPSIS

/etc/TIMEZONE

DESCRIPTION

This file is obsolete. It used to be executed by */etc/profile* and "dotted" into other files that needed to know the time. Now, *login(1)* initializes the timezone variable TZ as follows: from the environment, *environ(5)*, if it exists, or from the file */etc/TZ* if that exists, or from a default built into *login(1)*.

NAME

tpd – format of MIPS boot tape directories

SYNTAX

#include <saio/tpd.h>

DESCRIPTION

Boot tapes that can be read by the MIPS prom monitor contain a directory that allows the commands and records on the tape to load by name rather than by physical record number. Boot tape are produced by the command *mkboottape*(1M).

A boot tape consists of a number of physical tape files, where each file may optionally contain a *tpd* directory describing the contents of that physical file. The prom monitor provides a syntax for referencing a named record on a boot tape (see the MIPS PROM MONITOR documentation).

All binary values in the tpd directory are 2's complement, big-endian regardless of target machine. The tpd checksum is calculated by first zeroing the td_cksum field, then 2's complement summing all 32 bit words in the struct tp_dir and then assigning the 2's complement of the cksum to td_cksum. Thus the checksum is verified by resumming the header and verifing the sum to be zero. Each tape record is TP_BLKSIZE bytes long, trailing bytes in a tape block (after the end of the directory or an end of file) are unspecified (although they should be zero). All files start on a tape block boundry, the start of a particular file may be found by skipping backward to the beginning of the file containing the tape directory and then skipping forward tp_lbn records.

The format of the *tpd* directory is:

```
#define TP_NAMESIZE 16
#define TP_NENTRIES 20
#define TP_BLKSIZE  512
#define TP_MAGIC    0xaced1234

#ifdef LANGUAGE_C
/*
 * tape directory entry
 */
struct tp_entry {
        char            te_name[TP_NAMESIZE]; /* file name */
        unsigned        te_lbn;         /* tp record num */
                                        /* 0 is tp_dir */
        unsigned        te_nbytes;      /* file byte count */
};

/*
 * boot tape directory block
 * WARNING: must not be larger than 512 bytes!
 */
struct tp_dir {
        unsigned        td_magic;
        unsigned        td_cksum;       /* csum of tp_dir */
        unsigned        td_spare1;
        unsigned        td_spare2;
        unsigned        td_spare3;
        unsigned        td_spare4;
        unsigned        td_spare5;
```

```
        unsigned        td_spare6;
        struct tp_entry td_entry[TP_NENTRIES];      /* directory */
};

union tp_header {
        char    th_block[TP_BLKSIZE];
        struct  tp_dir th_td;
};
#endif LANGUAGE_C
```

te_name is a null-terminated string. The *td_magic* field contains TP_MAGIC to help verify the presence of a header.

SEE ALSO

mkboottape(1M)
MIPS PROM MONITOR manual

NAME

unistd – file header for symbolic constants

SYNOPSIS

#include <unistd.h>

DESCRIPTION

The header file *<unistd.h>* lists the symbolic constants and structures not already defined or declared in some other header file.

```
/* Symbolic constants for the "access" routine: */

#define R_OK     4      /*Test for Read permission */
#define W_OK     2      /*Test for Write permission */
#define X_OK     1      /*Test for eXecute permission */
#define F_OK     0      /*Test for existence of File */

#define F_ULOCK 0       /*Unlock a previously locked region */
#define F_LOCK   1      /*Lock a region for exclusive use */
#define F_TLOCK 2       /*Test and lock a region for exclusive use */
#define F_TEST   3      /*Test a region for other processes locks */

/*Symbolic constants for the "lseek" routine: */

#define SEEK_SET0       /* Set file pointer to "offset" */
#define SEEK_CUR        1/* Set file pointer to current plus "offset" */
#define SEEK_END        2/* Set file pointer to EOF plus "offset" */

/*Pathnames:*/

#define GF_PATH  /etc/group /*Pathname of the group file */
#define PF_PATH  /etc/passwd/*Pathname of the passwd file */
```

NAME

utmp, wtmp – utmp and wtmp entry formats

SYNOPSIS

```
#include <sys/types.h>
#include <utmp.h>
```

DESCRIPTION

These files, which hold user and accounting information for such commands as *who*(1), *write*(1), and *login*(1), have the following structure as defined by **<utmp.h>**:

```
#define   UTMP_FILE     "/etc/utmp"
#define   WTMP_FILE     "/etc/wtmp"
#define   ut_name       ut_user

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) */
        short   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. */
        time_t  ut_time;            /* time entry was made */
};

/* Definitions for ut_type */
#define EMPTY           0
#define RUN_LVL         1
#define BOOT_TIME       2
#define OLD_TIME        3
#define NEW_TIME        4
#define INIT_PROCESS    5       /* Process spawned by "init" */
#define LOGIN_PROCESS   6       /* A "getty" process waiting for login */
#define USER_PROCESS    7       /* A user process */
#define DEAD_PROCESS    8
#define ACCOUNTING      9
#define UTMAXTYPE       ACCOUNTING  /* Largest legal value of ut_type */

/* Special strings or formats used in the "ut_line" field when */
/* accounting for something other than a process */
/* No string for the ut_line field can be more than 11 chars + */
/* a NULL in length */
#define RUNLVL_MSG "run-level %c"
#define BOOT_MSG   "system boot"
#define OTIME_MSG  "old time"
#define NTIME_MSG  "new time"
```

FILES

/etc/utmp
/etc/wtmp

SEE ALSO

getut(3C).
login(1), who(1), write(1) in the *User's Reference Manual*.

NAME

uuencode – format of an encoded uuencode file

DESCRIPTION

Files output by *uuencode(1)* consist of a header line, followed by a number of body lines, and a trailer line. *uudecode(1)* will ignore any lines preceding the header or following the trailer. Lines preceding a header must not, of course, look like a header.

The header line is distinguished by having the first 6 characters "begin ". The word *begin* is followed by a mode (in octal), and a string which names the remote file. A space separates the three items in the header line.

The body consists of a number of lines, each at most 62 characters long (including the trailing newline). These consist of a character count, followed by encoded characters, followed by a newline. The character count is a single printing character, and represents an integer, the number of bytes the rest of the line represents. Such integers are always in the range from 0 to 63 and can be determined by subtracting the character space (octal 40) from the character.

Groups of 3 bytes are stored in 4 characters, 6 bits per character. All are offset by a space to make the characters printing. The last line may be shorter than the normal 45 bytes. If the size is not a multiple of 3, this fact can be determined by the value of the count on the last line. Extra garbage will be included to make the character count a multiple of 4. The body is terminated by a line with a count of zero. This line consists of one ASCII space.

The trailer line consists of "end" on a line by itself.

SEE ALSO

mail(1), uuencode(1), uudecode(1), uucp(1).

NAME

ascii – map of ASCII character set

DESCRIPTION

ascii is a map of the ASCII character set, giving both octal and hexadecimal equivalents of each character, to be printed as needed. It contains:

000 nul	001 soh	002 stx	003 etx	004 eot	005 enq	006 ack	007 bel
010 bs	011 ht	012 nl	013 vt	014 np	015 cr	016 so	017 si
020 dle	021 dc1	022 dc2	023 dc3	024 dc4	025 nak	026 syn	027 etb
030 can	031 em	032 sub	033 esc	034 fs	035 gs	036 rs	037 us
040 sp	041 !	042 "	043 #	044 $	045 %	046 &	047 '
050 (	051)	052 *	053 +	054 ,	055 -	056 .	057 /
060 0	061 1	062 2	063 3	064 4	065 5	066 6	067 7
070 8	071 9	072 :	073 ;	074 <	075 =	076 >	077 ?
100 @	101 A	102 B	103 C	104 D	105 E	106 F	107 G
110 H	111 I	112 J	113 K	114 L	115 M	116 N	117 O
120 P	121 Q	122 R	123 S	124 T	125 U	126 V	127 W
130 X	131 Y	132 Z	133 [	134 \	135]	136 ^	137 _
140 `	141 a	142 b	143 c	144 d	145 e	146 f	147 g
150 h	151 i	152 j	153 k	154 l	155 m	156 n	157 o
160 p	161 q	162 r	163 s	164 t	165 u	166 v	167 w
170 x	171 y	172 z	173 {	174 |	175 }	176 ~	177 del

00 nul	01 soh	02 stx	03 etx	04 eot	05 enq	06 ack	07 bel
08 bs	09 ht	0a nl	0b vt	0c np	0d cr	0e so	0f si
10 dle	11 dc1	12 dc2	13 dc3	14 dc4	15 nak	16 syn	17 etb
18 can	19 em	1a sub	1b esc	1c fs	1d gs	1e rs	1f us
20 sp	21 !	22 "	23 #	24 $	25 %	26 &	27 '
28 (	29)	2a *	2b +	2c ,	2d -	2e .	2f /
30 0	31 1	32 2	33 3	34 4	35 5	36 6	37 7
38 8	39 9	3a :	3b ;	3c <	3d =	3e >	3f ?
40 @	41 A	42 B	43 C	44 D	45 E	46 F	47 G
48 H	49 I	4a J	4b K	4c L	4d M	4e N	4f O
50 P	51 Q	52 R	53 S	54 T	55 U	56 V	57 W
58 X	59 Y	5a Z	5b [	5c \	5d]	5e ^	5f _
60	61 a	62 b	63 c	64 d	65 e	66 f	67 g
68 h	69 i	6a j	6b k	6c l	6d m	6e n	6f o
70 p	71 q	72 r	73 s	74 t	75 u	76 v	77 w
78 x	79 y	7a z	7b {	7c |	7d }	7e ~	7f del

NAME

disktab – disk description file

SYNOPSIS

#include <disktab.h>

DESCRIPTION

disktab is a simple date base which describes disk geometries and disk partition characteristics. Entries in *disktab* consist of a number of ':' separated fields. The first entry for each disk gives the names which are known for the disk, separated by '|' characters. The last name given should be a long name fully identifying the disk.

The following list indicates the normal values stored for each disk entry.

Name	Type	Description
ns	num	Number of sectors per track
nt	num	Number of tracks per cylinder
nc	num	Total number of cylinders on the disk
ba	num	Block size for partition 'a' (bytes)
bd	num	Block size for partition 'd' (bytes)
be	num	Block size for partition 'e' (bytes)
bf	num	Block size for partition 'f' (bytes)
bg	num	Block size for partition 'g' (bytes)
bh	num	Block size for partition 'h' (bytes)
fa	num	Fragment size for partition 'a' (bytes)
fd	num	Fragment size for partition 'd' (bytes)
fe	num	Fragment size for partition 'e' (bytes)
ff	num	Fragment size for partition 'f' (bytes)
fg	num	Fragment size for partition 'g' (bytes)
fh	num	Fragment size for partition 'h' (bytes)
pa	num	Size of partition 'a' in sectors
pb	num	Size of partition 'b' in sectors
pc	num	Size of partition 'c' in sectors
pd	num	Size of partition 'd' in sectors
pe	num	Size of partition 'e' in sectors
pf	num	Size of partition 'f' in sectors
pg	num	Size of partition 'g' in sectors
ph	num	Size of partition 'h' in sectors
se	num	Sector size in bytes
sf	bool	supports bad144-style bad sector forwarding
so	bool	partition offsets in sectors
ty	str	Type of disk (e.g. removable, winchester)

disktab entries may be automatically generated with the *diskpart* program.

FILES

/etc/disktab

SEE ALSO

newfs(1FFS)

BUGS

This file shouldn't exist, the information should be stored on each disk pack.

NAME

environ – user environment

DESCRIPTION

An array of strings called the "environment" is made available by *exec*(2) when a process begins. By convention, these strings have the form "name=value". The following names are used by various commands:

PATH The sequence of directory prefixes that *sh*(1), *time*(1), *nice*(1), *nohup*(1), etc., apply in searching for a file known by an incomplete path name. The prefixes are separated by colons (**:**). *login*(1) sets **PATH=:/bin:/usr/bin**.

HOME Name of the user's login directory, set by *login*(1) from the password file *passwd*(4).

PERROR_FMT The format string for system-related error messages printed by the *perror(3)* subroutine. See *perror(3)* for details.

TERM The kind of terminal for which output is to be prepared. This information is used by commands, such as *mm*(1) or *tplot*(1G), which may exploit special capabilities of that terminal.

TZ Time zone information. The format is **xxx***n***zzz** where **xxx** is standard local time zone abbreviation, *n* is the difference in hours from GMT, and **zzz** is the abbreviation for the daylight-saving local time zone, if any; for example, **EST5EDT**.

Further names may be placed in the environment by the *export* command and "name=value" arguments in *sh*(1), or by *exec*(2). It is unwise to conflict with certain shell variables that are frequently exported by **.profile** files: **MAIL, PS1, PS2, IFS.**

SEE ALSO

exec(2).
env(1), login(1), sh(1), nice(1), nohup(1), time(1), tplot(1G) in the *User's Reference Manual.*
mm(1) in the *DOCUMENTER'S WORKBENCH* Software Release 2.0 Technical Discussion and Reference Manual.

NAME

fcntl – file control options

SYNOPSIS

#include <fcntl.h>

DESCRIPTION

The *fcntl*(2) function provides for control over open files. This include file describes *requests* and *arguments* to *fcntl* and *open*(2).

```
/* Flag values accessible to open(2) and fcntl(2) */
/*  (The first three can only be set by open) */
#define O_RDONLY 0
#define O_WRONLY 1
#define O_RDWR   2
#define O_NDELAY 04      /* Non-blocking I/O */
#define O_APPEND 010     /* append (writes guaranteed at the end) */
#define O_SYNC   020     /* synchronous write option */

/* Flag values accessible only to open(2) */
#define O_CREAT  00400   /* open with file create (uses third open arg)*/
#define O_TRUNC  01000   /* open with truncation */
#define O_EXCL   02000   /* exclusive open */

/* fcntl(2) requests */
#define F_DUPFD  0       /* Duplicate fildes */
#define F_GETFD  1       /* Get fildes flags */
#define F_SETFD  2       /* Set fildes flags */
#define F_GETFL  3       /* Get file flags */
#define F_SETFL  4       /* Set file flags */
#define F_GETLK  5       /* Get file lock */
#define F_SETLK  6       /* Set file lock */
#define F_SETLKW 7       /* Set file lock and wait */
#define F_CHKFL  8       /* Check legality of file flag changes */

/* file segment locking control structure */
struct flock {
        short l_type;
        short l_whence;
        long  l_start;
        long  l_len;        /* if 0 then until EOF */
        short l_sysid;      /* returned with F_GETLK*/
        short l_pid;        /* returned with F_GETLK*/
}

/* file segment locking types */
#define F_RDLCK 01    /* Read lock */
#define F_WRLCK 02    /* Write lock */
#define F_UNLCK 03    /* Remove locks */
```

SEE ALSO

fcntl(2), open(2).

NAME

intro – introduction to miscellany

DESCRIPTION

This section describes miscellaneous facilities such as macro packages, character set tables, etc.

NAME

math – math functions and constants

SYNOPSIS

#include <math.h>

DESCRIPTION

This file contains declarations of all the functions in the Math Library (described in Section 3M), as well as various functions in the C Library (Section 3C) that return floating-point values. It defines the constants used as error-return values:

HUGE	The maximum value of a single-precision floating-point number. The following mathematical constants are defined for user convenience:
M_E	The base of natural logarithms (e).
M_LOG2E	The base-2 logarithm of e.
M_LOG10E	The base-10 logarithm of e.
M_LN2	The natural logarithm of 2.
M_LN10	The natural logarithm of 10.
M_PI	π, the ratio of the circumference of a circle to its diameter.
M_PI_2	$\pi/2$.
M_PI_4	$\pi/4$.
M_1_PI	$1/\pi$.
M_2_PI	$2/\pi$.
M_2_SQRTPI	$2/\sqrt{\pi}$.
M_SQRT2	The positive square root of 2.
M_SQRT1_2	The positive square root of 1/2. For the definitions of various machine-dependent "constants," see the description of the *<values.h>* header file.

SEE ALSO

intro(3), values(5).

NAME

regexp – regular expression compile and match routines

SYNOPSIS

#define INIT <declarations>
#define GETC() <getc code>
#define PEEKC() <peekc code>
#define UNGETC(c) <ungetc code>
#define RETURN(pointer) <return code>
#define ERROR(val) <error code>

#include <regexp.h>

char *compile (instring, expbuf, endbuf, eof)
char *instring, *expbuf, *endbuf;
int eof;

int step (string, expbuf)
char *string, *expbuf;

extern char *loc1, *loc2, *locs;

extern int circf, sed, nbra;

DESCRIPTION

This page describes general-purpose regular expression matching routines in the form of *ed*(1), defined in **<regexp.h>** . Programs such as *ed*(1), *sed*(1), *grep*(1), *bs*(1), *expr*(1), etc., which perform regular expression matching use this source file. In this way, only this file need be changed to maintain regular expression compatibility.

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 character in the regular expression pattern. Successive calls to GETC( ) should return successive characters of the regular expression.
PEEKC( )	Return the next character in the regular expression. Successive calls to PEEKC( ) should return the same character [which should also be the next character returned by GETC( )].
UNGETC(*c*)	Cause the argument *c* to be returned by the next call to GETC( ) [and PEEKC( )]. No more that one character of pushback is ever needed and this character is guaranteed to be the last character 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 which have memory allocation to manage.
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 which 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 place where the compiled regular expression will be placed.

The parameter *endbuf* is one more than the highest address where the compiled regular expression may 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 will be 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 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).

There are other functions in this file which perform actual regular expression matching, one of which is the function *step*. 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 which was obtained by a call of the function *compile*.

The function *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* will point to the first character of *string* and *loc2* will point 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 then *step* will try 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*.

The function *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 indicating a match or until the end of *string* is reached. If one wants to constrain *string* to the beginning of the line in all cases, *step* need not be called; simply call *advance*.

When *advance* encounters a * or \{ \} sequence in the regular expression, it will advance its pointer to the string to be matched as far as possible and will recursively call itself trying to match the rest of the string to the rest of the regular expression. As long as there is no match, *advance* will back 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* will break out of the loop that backs up and will return 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 like **s/y*//g** do not loop forever.

The additional external variables *sed* and *nbra* are used for special purposes.

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

ed(1), expr(1), grep(1), sed(1) in the *User's Reference Manual*.

NAME

resolver – configuration file for name server routines

DESCRIPTION

The resolver configuration file contains information that is read by the resolver routines the first time they are invoked in a process. The file is designed to be human readable and contains a list of name-value pairs that provide various types of resolver information.

The different configuration options are:

nameserver followed by the Internet address (in dot notation) of a name server that the resolver should query. At least one name server should be listed. Up to MAXNS (currently 3) name servers may be listed, in that case the resolver library queries tries them in the order listed. (The algorithm used is to try a name server, and if the query times out, try the next, until out of name servers, then repeat trying all the name servers until a maximum number of retries are made).

domain followed by a domain name, that is the default domain to append to names that do not have a dot in them. This defaults to the domain set by the domainname(1) command.

address followed by an Internet address (in dot notation) of any preferred networks. The list of addresses returned by the resolver will be sorted to put any addresses on this network before any others.

The name value pair must appear on a single line, and the keyword (e.g. **nameserver**) must start the line. The value follows the keyword, separated by white space.

FILES

/etc/resolv.conf

SEE ALSO

domainname(1), gethostent(3N), named(8C)

NAME

stat – data returned by stat system call

SYNOPSIS

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

DESCRIPTION

The system calls *stat*, *lstat*, and *fstat* return data whose structure is defined by this include file. The encoding of the field *st_mode* is defined in this file also.

Structure of the result of stat

```
struct  stat
{
        dev_t   st_dev;
        ushort  st_ino;
        ushort  st_mode;
        short   st_nlink;
        ushort  st_uid;
        ushort  st_gid;
        dev_t   st_rdev;
        off_t   st_size;
        time_t  st_atime;
        time_t  st_mtime;
        time_t  st_ctime;
};

#define S_IFMT   0170000 /* type of file */
#define S_IFDIR  0040000 /* directory */
#define S_IFCHR  0020000 /* character special */
#define S_IFBLK  0060000 /* block special */
#define S_IFREG  0100000 /* regular */
#define S_IFIFO  0010000 /* fifo */
#define S_IFLNK  0120000 /* symbolic link */
#define S_ISUID  04000   /* set user id on execution */
#define S_ISGID  02000   /* set group id on execution */
#define S_ISVTX  01000   /* save swapped text even after use */
#define S_IREAD  00400   /* read permission, owner */
#define S_IWRITE 00200   /* write permission, owner */
#define S_IEXEC  00100   /* execute/search permission, owner */
#define S_ENFMT  S_ISGID /* record locking enforcement flag */
#define S_IRWXU  00700   /* read,write, execute: owner */
#define S_IRUSR  00400   /* read permission: owner */
#define S_IWUSR  00200   /* write permission: owner */
#define S_IXUSR  00100   /* execute permission: owner */
#define S_IRWXG  00070   /* read, write, execute: group */
#define S_IRGRP  00040   /* read permission: group */
#define S_IWGRP  00020   /* write permission: group */
#define S_IXGRP  00010   /* execute permission: group */
#define S_IRWXO  00007   /* read, write, execute: other */
#define S_IROTH  00004   /* read permission: other */
#define S_IWOTH  00002   /* write permission: other */
#define S_IXOTH  00001   /* execute permission: other */
```

SEE ALSO

stat(2), types(5).

NAME

term – conventional names for terminals

DESCRIPTION

These names are used by certain commands (e.g., *man*(1), *tabs*(1), *tput*(1), *vi*(1) and *curses*(3X)) and are maintained as part of the shell environment in the environment variable **TERM** (see *sh*(1), *profile*(4), and *environ*(5)).

Entries in *terminfo*(4) source files consist of a number of comma-separated fields. (To obtain the source description for a terminal, use the **–I** option of *infocmp*(1M).) White space after each comma is ignored. The first line of each terminal description in the *terminfo*(4) database gives the names by which *terminfo*(4) knows the terminal, separated by bar (|) characters. The first name given is the most common abbreviation for the terminal (this is the one to use to set the environment variable **TERMINFO** in *$HOME/.profile*; see *profile*(4)), the last name given should be a long name fully identifying the terminal, and all others are understood as synonyms for the terminal name. All names but the last should contain no blanks and must be unique in the first 14 characters; the last name may contain blanks for readability. Terminal names (except for the last, verbose entry) should be chosen using the following conventions. The particular piece of hardware making up the terminal should have a root name chosen, for example, for the AT&T 4425 terminal, **att4425**. This name should not contain hyphens, except that synonyms may be chosen that do not conflict with other names. Up to 8 characters, chosen from [a–z0–9], make up a basic terminal name. Names should generally be based on original vendors, rather than local distributors. A terminal acquired from one vendor should not have more than one distinct basic name. Terminal sub-models, operational modes that the hardware can be in, or user preferences, should be indicated by appending a hyphen and an indicator of the mode. Thus, an AT&T 4425 terminal in 132 column mode would be **att4425–w**. The following suffixes should be used where possible:

Suffix	Meaning	Example
–w	Wide mode (more than 80 columns)	att4425–w
–am	With auto. margins (usually default)	vt100–am
–nam	Without automatic margins	vt100–nam
–n	Number of lines on the screen	aaa–60
–na	No arrow keys (leave them in local)	c100–na
–np	Number of pages of memory	c100–4p
–rv	Reverse video	att4415–rv

To avoid conflicts with the naming conventions used in describing the different modes of a terminal (e.g., **–w**), it is recommended that a terminal's root name not contain hyphens. Further, it is good practice to make all terminal names used in the *terminfo*(4) database unique. Terminal entries that are present only for inclusion in other entries via the **use=** facilities should have a '+' in their name, as in **4415+nl**.

Some of the known terminal names may include the following (for a complete list, type: **ls -C /usr/lib/terminfo/?**):

2621,hp2621	Hewlett-Packard 2621 series
2631	Hewlett-Packard 2631 line printer
2631–c	Hewlett-Packard 2631 line printer - compressed mode
2631–e	Hewlett-Packard 2631 line printer - expanded mode
2640,hp2640	Hewlett-Packard 2640 series
2645,hp2645	Hewlett-Packard 2645 series
3270	IBM Model 3270
33,tty33	AT&T Teletype Model 33 KSR
35,tty35	AT&T Teletype Model 35 KSR
37,tty37	AT&T Teletype Model 37 KSR
4000a	Trendata 4000a

4014,tek4014	TEKTRONIX 4014
40,tty40	AT&T Teletype Dataspeed 40/2
43,tty43	AT&T Teletype Model 43 KSR
4410,5410	AT&T 4410/5410 terminal in 80-column mode - version 2
4410-nfk,5410-nfk	AT&T 4410/5410 without function keys - version 1
4410-nsl,5410-nsl	AT&T 4410/5410 without pln defined
4410-w,5410-w	AT&T 4410/5410 in 132-column mode
4410v1,5410v1	AT&T 4410/5410 terminal in 80-column mode - version 1
4410v1-w,5410v1-w	AT&T 4410/5410 terminal in 132-column mode - version 1
4415,5420	AT&T 4415/5420 in 80-column mode
4415-nl,5420-nl	AT&T 4415/5420 without changing labels
4415-rv,5420-rv	AT&T 4415/5420 80 columns in reverse video
4415-rv-nl,5420-rv-nl	AT&T 4415/5420 reverse video without changing labels
4415-w,5420-w	AT&T 4415/5420 in 132-column mode
4415-w-nl,5420-w-nl	AT&T 4415/5420 in 132-column mode without changing labels
4415-w-rv,5420-w-rv	AT&T 4415/5420 132 columns in reverse video
4415-w-rv-nl,5420-w-rv-nl	AT&T 4415/5420 132 columns reverse video without changing labels
4418,5418	AT&T 5418 in 80-column mode
4418-w,5418-w	AT&T 5418 in 132-column mode
4420	AT&T Teletype Model 4420
4424	AT&T Teletype Model 4424
4424-2	AT&T Teletype Model 4424 in display function group ii
4425,5425	AT&T 4425/5425
4425-fk,5425-fk	AT&T 4425/5425 without function keys
4425-nl,5425-nl	AT&T 4425/5425 without changing labels in 80-column mode
4425-w,5425-w	AT&T 4425/5425 in 132-column mode
4425-w-fk,5425-w-fk	AT&T 4425/5425 without function keys in 132-column mode
4425-nl-w,5425-nl-w	AT&T 4425/5425 without changing labels in 132-column mode
4426	AT&T Teletype Model 4426S
450	DASI 450 (same as Diablo 1620)
450	DASI 450 in 12-pitch mode
500,att500	AT&T-IS 500 terminal
510,510a	AT&T 510/510a in 80-column mode
513bct,att513	AT&T 513 bct terminal
5320	AT&T 5320 hardcopy terminal
5420_2	AT&T 5420 model 2 in 80-column mode
5420_2-w	AT&T 5420 model 2 in 132-column mode
5620,dmd	AT&T 5620 terminal 88 columns
5620-24,dmd-24	AT&T Teletype Model DMD 5620 in a 24x80 layer
5620-34,dmd-34	AT&T Teletype Model DMD 5620 in a 34x80 layer
610,610bct	AT&T 610 bct terminal in 80-column mode
610-w,610bct-w	AT&T 610 bct terminal in 132-column mode
7300,pc7300,unix_pc	AT&T UNIX PC Model 7300
735,ti	Texas Instruments TI 735 and TI 725
745	Texas Instruments TI 745
dumb	generic name for terminals that lack reverse line-feed and other special escape sequences

hp	Hewlett-Packard (same as 2645)
lp	generic name for a line printer
pt505	AT&T Personal Terminal 505 (22 lines)
pt505–24	AT&T Personal Terminal 505 (24-line mode)
sync	generic name for synchronous Teletype Model 4540-compatible terminals

Commands whose behavior depends on the type of terminal should accept arguments of the form **–T***term* where *term* is one of the names given above; if no such argument is present, such commands should obtain the terminal type from the environment variable **TERM**, which, in turn, should contain *term*.

FILES

/usr/lib/terminfo/?/* compiled terminal description database

SEE ALSO

curses(3X), profile(4), terminfo(4), environ(5).
man(1), sh(1), stty(1), tabs(1), tput(1), tplot(1G), vi(1) in the *User's Reference Manual.*
infocmp(1M) in the *System Administrator's Reference Manual*.
Chapter 10 of the *Programmer's Guide.*

NOTES

Not all programs follow the above naming conventions.

NAME

types – primitive system data types

SYNOPSIS

#include <sys/types.h>

DESCRIPTION

The data types defined in the include file are used in UNIX system code; some data of these types are accessible to user code:

```
typedef  struct { int r[1]; } *physadr;
typedef  long               daddr_t;
typedef  char *             caddr_t;
typedef unsigned char       unchar;
typedef unsigned short      ushort;
typedef  unsigned int       uint;
typedef  unsigned long      ulong;
typedef  ushort             ino_t;
typedef  short              cnt_t;
typedef  long               time_t;
typedef  int                label_t[10];
typedef  short              dev_t;
typedef  long               off_t;
typedef  long               paddr_t;
typedef int                 key_t;
typedef unsigned char       use_t;
typedef short               sysid_t;
typedef short               index_t;
typedef short               lock_t;
typedef unsigned int        size_t;
```

The form *daddr_t* is used for disk addresses except in an i-node on disk, see *fs*(4). Times are encoded in seconds since 00:00:00 GMT, January 1, 1970. The major and minor parts of a device code specify kind and unit number of a device and are installation-dependent. Offsets are measured in bytes from the beginning of a file. The *label_t* variables are used to save the processor state while another process is running.

SEE ALSO

fs(4).

NAME

values – machine-dependent values

SYNOPSIS

#include <values.h>

DESCRIPTION

This file contains a set of manifest constants, conditionally defined for particular processor architectures. The model assumed for integers is binary representation (one's or two's complement), where the sign is represented by the value of the high-order bit.

BITS(*type*)	The number of bits in a specified type (e.g., int).
HIBITS	The value of a short integer with only the high-order bit set (in most implementations, 0x8000).
HIBITL	The value of a long integer with only the high-order bit set (in most implementations, 0x80000000).
HIBITI	The value of a regular integer with only the high-order bit set (usually the same as HIBITS or HIBITL).
MAXSHORT	The maximum value of a signed short integer (in most implementations, 0x7FFF ≡ 32767).
MAXLONG	The maximum value of a signed long integer (in most implementations, 0x7FFFFFFF ≡ 2147483647).
MAXINT	The maximum value of a signed regular integer (usually the same as MAXSHORT or MAXLONG).
MAXFLOAT, LN_MAXFLOAT	The maximum value of a single-precision floating-point number, and its natural logarithm.
MAXDOUBLE, LN_MAXDOUBLE	The maximum value of a double-precision floating-point number, and its natural logarithm.
MINFLOAT, LN_MINFLOAT	The minimum positive value of a single-precision floating-point number, and its natural logarithm.
MINDOUBLE, LN_MINDOUBLE	The minimum positive value of a double-precision floating-point number, and its natural logarithm.
FSIGNIF	The number of significant bits in the mantissa of a single-precision floating-point number.
DSIGNIF	The number of significant bits in the mantissa of a double-precision floating-point number.

SEE ALSO

intro(3), math(5).

NAME

varargs – handle variable argument list

SYNOPSIS

#include <varargs.h> va_alist va_dcl void va_start(pvar)
va_list pvar; *type* **va_arg(pvar,** *type*)
va_list pvar; void va_end(pvar)
va_list pvar;

DESCRIPTION

This set of macros allows portable procedures that accept variable argument lists to be written. Routines that have variable argument lists [such as *printf*(3S)] but do not use *varargs* are inherently nonportable, as different machines use different argument-passing conventions.

va_alist is used as the parameter list in a function header.

va_dcl is a declaration for *va_alist*. No semicolon should follow *va_dcl*.

va_list is a type defined for the variable used to traverse the list.

va_start is called to initialize *pvar* to the beginning of the list.

va_arg will return the next argument in the list pointed to by *pvar*. *type* is the type the argument is expected to be. Different types can be mixed, but it is up to the routine to know what type of argument is expected, as it cannot be determined at runtime.

va_end is used to clean up.

Multiple traversals, each bracketed by *va_start ... va_end,* are possible.

EXAMPLE

This example is a possible implementation of *execl*(2).

```
#include <varargs.h>
#define MAXARGS    100

/*      execl is called by
                execl(file, arg1, arg2, ..., (char *)0);
*/
execl(va_alist)
va_dcl
{
        va_list ap;
        char *file;
        char *args[MAXARGS];
        int argno = 0;

        va_start(ap);
        file = va_arg(ap, char *);
        while ((args[argno++] = va_arg(ap, char *)) != (char *)0)
                ;
        va_end(ap);
        return execv(file, args);
}
```

SEE ALSO

exec(2), printf(3S), vprintf(3S).

NOTES

It is up to the calling routine to specify how many arguments there are, since it is not always possible to determine this from the stack frame. For example, *execl* is passed a zero pointer to signal the end of the list. *printf* can tell how many arguments are there by the format.
It is non-portable to specify a second argument of *char*, *short*, or *float* to *va_arg*, since arguments seen by the called function are not *char*, *short*, or *float*. C converts *char* and *short* arguments to *int* and converts *float* arguments to *double* before passing them to a function.